Linux Komut Satırı

Yedinci İnternet Sürümü

William Shotts

Bir LinuxCommand.org Kitabı

Telif Hakkı ©2026, William Shotts

Bu eser, Creative Commons Atıf-Gayri Ticari-Türev Eserler Yok 3.0 Amerika Birleşik Devletleri Lisansı altında lisanslanmıştır. Bu lisansın bir kopyasını görmek için yukarıdaki bağlantıyı ziyaret edin veya Creative Commons, PO Box 1866, Mountain View, CA 94042 adresine mektup gönderin.

Bu kitabın bir versiyonu, No Starch Press tarafından basılı olarak da yayınlanmıştır. Kitabın kopyaları, kaliteli kitapların satıldığı her yerden satın alınabilir. No Starch Press, popüler e-okuyucular için elektronik formatlar da sunmaktadır. Şu adresten ulaşabilirsiniz: https://www.nostarch.com.

Linux® Linus Torvalds'ın tescilli ticari markasıdır. Diğer tüm ticari markalar ilgili sahiplerine aittir.

Bu kitap, eski işletim sistemi kullanıcılarının geleceğe geçişine yardımcı olmaya adanmış bir Linux eğitimi ve tanıtım sitesi olan LinuxCommand.org projesinin bir parçasıdır. LinuxCommand.org projesine https://linuxcommand.org adresinden ulaşabilirsiniz.

Sürüm Geçmişi

Sürüm	Tarih	Açıklama
25.12	5 Ocak 2026	Yedinci İnternet Sürümü
24.11	1 Kasım 2024	Altıncı İnternet Sürümü
19.01A	28 Ocak 2019	Beşinci İnternet Sürümü (Düzeltilmiş İçindekiler)
19.01	17 Ocak 2019	Beşinci İnternet Sürümü.
17.10	19 Ekim 2017	Dördüncü İnternet Sürümü.
16.07	28 Temmuz 2016	Üçüncü İnternet Sürümü.
13.07	6 Temmuz 2013	İkinci İnternet Sürümü.
09.12	14 Aralık 2009	İlk İnternet Baskısı.

ÇEVİRMEN NOTU

Bu belge, William Shotts tarafından yazılan
“The Linux Command Line (7th Internet Edition)” adlı eserin
Türkçe çevirisidir.

Orijinal eser © William Shotts’a aittir.
Bu çeviri ticari amaç güdülmeden, eğitim amacıyla hazırlanmıştır.

Bu çalışma, Creative Commons
Atıf-GayriTicari-Türev Eserler Yok 3.0 (CC BY-NC-ND 3.0)
lisansı kapsamında paylaşılmaktadır:

https://creativecommons.org/licenses/by-nc-nd/3.0/

Türkçe çeviri:
Recep Şenel
2026

İçindekiler

Giriş ..... ***xvi***

Komut Satırını Neden Kullanmalı? xvi
Bu Kitabın Konusu xvii
Bu Kitabı Kimler Okumalı? xvii
Bu Kitapta Neler Var xviii
Bu Kitabı Nasıl Okumalı xix
Ön Koşullar xix
Neden "GNU/Linux" Demiyorum xix
Yedinci İnternet Sürümündeki Yenilikler xx
Teşekkür xx
Yedinci İnternet Sürümü xxi
Önceki Baskılar xxi
Görüşlerinizi Bekliyoruz! xxi
Daha Fazla Okuma xxi

Bölüm 1 – Kabuğu Öğrenmek **1**

1 – Kabuk Nedir? ***2***

Terminal Emülatörleri 2
İlk Tuş Vuruşlarınızı Yapma 2
Komut Geçmişi 3
İmleç Hareketi 3
Fare ve Odak Hakkında Birkaç Söz 3
Basit Komutları Deneyin 4
Terminal Oturumunu Sonlandırma 5
Perdenin Arkasındaki Konsol 5
Özet 5
Daha Fazla Okuma 6

2 – Gezinme ***7***

Dosya Sistemi Ağacını Anlamak 7
Geçerli Çalışma Dizini 7
Bir Dizinin İçeriğini Listeleme 8
Geçerli Çalışma Dizinini Değiştirme 9
Mutlak Yol Adları 9
Göreceli Yol Adları 9
Bazı Yararlı Kısayollar 11
Dosya Adları Hakkında Önemli Bilgiler 11

Özet ... 12

3 – Sistemi Keşfetmek ... ***13***

ls ile Daha Fazla Eğlence ... 13
Seçenekler ve Argümanlar ... 14
Uzun Biçimine Daha Yakından Bakış ... 15
file ile Dosya Türünü Belirleme ... 16
less ile Dosya İçeriğini Görüntüleme ... 17
"Metin" Nedir? ... 17
Az, çoktur ... 19
Rehberli Tur ... 19
Sembolik Bağlantılar ... 23
Sabit Bağlantılar ... 24
Özet ... 24
Daha Fazla Okuma ... 25

4 – Dosya ve Dizinleri İşleme ... ***26***

Joker Karakterler ... 26
Karakter Aralıkları ... 28
Nokta Dosyaları ... 28
Joker karakterler GUI'de de çalışır ... 29
mkdir – Dizin Oluşturma ... 29
cp – Dosya ve Dizinleri Kopyalama ... 30
Yararlı Seçenekler ve Örnekler ... 30
mv – Dosyaları Taşıma ve Yeniden Adlandırma ... 31
Yararlı Seçenekler ve Örnekler ... 32
rm – Dosyaları ve Dizinleri Silme ... 33
Yararlı Seçenekler ve Örnekler ... 33
rm ile Dikkatli Olun! ... 34
ln – Bağlantılar Oluşturma ... 34
Sabit Bağlantılar ... 35
Sembolik Bağlantılar ... 35
Bir Oyun Alanı Oluşturalım ... 35
Dizin Oluşturma ... 36
Dosyaları Kopyalama ... 36
Dosyaları Taşıma ve Yeniden Adlandırma ... 37
Sabit Bağlantılar Oluşturma ... 38
Sembolik Bağlantılar Oluşturma ... 39
Dosya ve Dizinleri Silme ... 41
GUI ile Sembolik Bağlantılar Oluşturma ... 42
Özet ... 42
Daha Fazla Okuma ... 43

5 – Komutlarla Çalışma ... ***44***

Komutlar Tam Olarak Nedir? ... 44
Komutları Tanımlama ... 45
type – Bir Komutun Türünü Görüntüleme ... 45
which – Yürütülebilir Dosyanın Konumunu Görüntüleme ... 45
Komutun Belgelerini Alma ... 46
help – Kabuk Yerleşik Komutları için Yardım Alma ... 46
--help – Kullanım Bilgilerini Görüntüleme ... 47
man – Bir Programın Kılavuz Sayfasını Görüntüleme ... 48

apropos – Uygun Komutları Görüntüle 49
whatis – Tek Satırlık Kılavuz Sayfası Açıklamalarını Görüntüleme 50
Hepsinden Daha Acımasız Man Sayfası 50
info – Bir Programın Bilgi Girişini Görüntüleme 50
README ve Diğer Program Belgeleri Dosyaları 52
alias ile Kendi Komutlarımızı Oluşturma 52
Özet 54
Daha Fazla Okuma 54

6 – Yönlendirme ***56***

Standart Giriş, Çıkış ve Hata 56
Standart Çıktıyı Yönlendirme 57
Grup Komutları 59
Standart Hata Yönlendirme 59
Standart Çıktı ve Standart Hata Bir Dosyaya Yönlendirme 60
İstenmeyen Çıktıları Silme 61
Unix Kültüründe /dev/null 61
Standart Girişi Yeniden Yönlendirme 61
cat – Dosyaları Birleştirme 61
Boru Hatları 63
> ve | Arasındaki Fark 64
Filtreler 65
uniq - Tekrarlanan Satırları Raporla veya Atla 65
wc – Satır, Kelime ve Bayt Sayısını Yazdır 66
grep – Desenle Eşleşen Satırları Yazdır 66
head / tail – Dosyaların İlk / Son Kısmını Yazdırma 67
tee – Stdin'den okur ve Stdout'a ve dosyalara yazdırır 69
Özet 69
Linux Hayal Gücüyle İlgilidir 70

7 – Dünyayı Kabuğun Gördüğü Gibi Görmek ***71***

Genişleme 71
Yol Adı Genişletme 72
Gizli Dosyaların Yol Adı Genişletme 73
Tilde Genişletme 73
Aritmetik Genişletme 74
Köşeli Parantez Genişletme 75
Parametre Genişletme 76
Komut Değiştirme 77
Tırnak İşareti 7
Çift Tırnak 79
Tek Tırnak 81
Kaçış Karakterleri 81
Ters eğik çizgi kaçış dizileri 82
Özet 83
Daha Fazla Okuma 83

8 – Gelişmiş Klavye Püf Noktaları ***84***

Komut Satırı Düzenleme 84
İmleç Hareketi 84
Metin Değiştirme 85
Metni Kesme ve Yapıştırma (Silme ve Çekme) 85

Meta Tuşu 86
Tamamlama 86
Programlanabilir Tamamlama 88
Geçmişi Kullanma 88
Geçmişi Arama 89
Geçmişi Genişletme 91
komut dosyası 92
Özet 92
Daha Fazla Okuma 92

9 – İzinler ***93***

Kullanıcılar, Grup Üyeleri ve Diğerleri 94
Okuma, Yazma ve Yürütme 95
chmod – Dosya Modunu Değiştirme 97
Sekizlik Nedir? 98
GUI ile Dosya Modunu Ayarlama 10
umask – Varsayılan İzinleri Ayarlama 102
Bazı Özel İzinler 104
Kimlik Değişikliği 105
su – İkame Kullanıcı ve Grup Kimlikleri ile Kabuk Çalıştırma 105
sudo – Başka Bir Kullanıcı Olarak Komut Çalıştırma 107
Modern Linux Dağıtımları ve sudo 107
chown – Dosya Sahibini ve Grubunu Değiştirme 108
chgrp – Grup Sahipliğini Değiştirme 109
Ayrıcalıklarımızı Kullanma 110
Şifrenizi Değiştirme 111
Özet 113
Daha Fazla Okuma 113

10 – Süreçler ***114***

Bir Süreç Nasıl Çalışır? 114
Süreçleri Görüntüleme 115
top ile Süreçleri Dinamik Olarak Görüntüleme 118
İşlemleri Kontrol Etme 120
Bir Süreci Kesintiye Uğratmak 121
İşlemi Arka Plana Alma 122
İşlemi Ön Plana Geri Getirme 123
Bir Süreci Durdurma (Askıya Alma) 123
İşlem Önceliklerini Değiştirme 124
Sinyaller 125
kill ile Süreçlere Sinyal Gönderme 126
Bir Süreci Askıya Alma Koruması Yapma 129
killall ile Birden Fazla İşleme Sinyal Gönderme 129
Sistemi Kapatma 130

İşlemlerle İlgili Diğer Komutlar 130
Özet 131

Bölüm 2 – Yapılandırma ve Ortam **133**

11 – Ortam ***134***

Ortamda Neler Depolanır? 134
Çevreyi İncelemek 134
Bazı İlginç Değişkenler 136
Çevre Nasıl Oluşur? 137
Başlangıç Dosyasında Neler Var? 139
Alt Süreçlerin Ortamlarını Nasıl Miras Aldığını Keşfetmek 140
Geçici Ortamla Bir Programı Başlatma 142
Ortamı Değiştirme 143
Hangi Dosyaları Değiştirmeliyiz? 143
Metin Editörleri 143
Metin Düzenleyiciyi Kullanma 144
Yorumlar Neden Önemlidir? 147
Değişikliklerimizi Etkinleştirme 148
Kaynak Hakkında Biraz Daha Fazla Bilgi 148
Özet 148
Daha Fazla Okuma 149

12 – vi(m) için Kolay Başlangıç 150

Neden vi'yi Öğrenmeliyiz? 150
Biraz Arka Plan Bilgisi 151
vi'yi Başlatma ve Durdurma 151
Uyumluluk Modu 152
Düzenleme Modları 153
Ekleme Moduna Girme 154
Çalışmamızı Kaydetme 154
İmleci Hareket Ettirme 155
Temel Düzenleme 156
Metin Ekleme 156
Satır Açma 157
Metin Silme 158
Metni Kesme, Kopyalama ve Yapıştırma 160
Satırları Birleştirme 161
Arama ve Değiştirme 16
Satır İçinde Arama 161
Tüm Dosyada Arama 162
Global Arama ve Değiştirme 162
Birden Çok Dosyayı Düzenleme 164
Dosyalar Arasında Geçiş Yapma 165
Düzenleme için Ek Dosyalar Açma 166
Bir Dosyadan Diğerine İçerik Kopyalama 166
Bir Dosyanın Tamamını Başka Bir Dosyaya Ekleme 167
Çalışmamızı Kaydetme 168
Bash da vi yapar. 169
Özet 170
Daha Fazla Okuma 170

13 – Komut Satırını Özelleştirme 171

İstemlerin Yapısı 171
Alternatif Komut İstemi Tasarımlarını Deneme 173
Renk Ekleme 174
Terminal Karışıklığı 175
İmleci Taşıma 177

Komut Satırını Kaydetme 178
Özet....179
Daha Fazla Okuma 179

Bölüm 3 – Yaygın Görevler ve Temel Araçlar 181

14 – Paket Yönetimi 182

Paketleme Sistemleri 182
Paket Sistemi Nasıl Çalışır? 183
Paket Dosyaları 183
Depolar 183
Bağımlılıklar 184
Yüksek ve Düşük Seviyeli Paket Araçları 184
Yaygın Paket Yönetimi Görevleri 185
Depoda Paket Bulma 185
Depodan Paket Yükleme 186
Paket Dosyasından Paket Yükleme 186
Paket Kaldırma 187
Depodan Paketleri Güncelleme 187
Paket Dosyasından Paketi Yükseltme 188
Yüklü Paketleri Listeleme 188
Bir Paketin Yüklü Olup Olmadığını Belirleme 188
Yüklü Bir Paket Hakkında Bilgi Görüntüleme 189
Bir Dosyayı Yükleyen Paketi Bulma 189
Dağıtımdan Bağımsız Paket Biçimleri 190
Özet....191
Linux Yazılım Kurulumu Efsanesi 191
Daha Fazla Okuma 192

15 – Depolama Ortamları 193

Depolama Aygıtlarını Takma ve Çıkarma 193
Takılı Dosya Sistemlerinin Listesini Görüntüleme 195
Bağlantıyı Kesmenin Önemi 198
Aygıt Adlarını Belirleme 199
Yeni Dosya Sistemleri Oluşturma 203
parted ile Bölümleri İşleme 203
mkfs ile Yeni Dosya Sistemi Oluşturma 207
Dosya Sistemlerini Test Etme ve Onarma 209
fsck nedir? 21
Verileri Doğrudan Aygıtlara ve Aygıtlardan Taşıma 210
CD-ROM Görüntüleri Oluşturma 211
CD-ROM'un Görüntü Kopyasını Oluşturma 211
Dosya Koleksiyonundan Görüntü Oluşturma 211
Başka Bir Adla Program. 212
CD-ROM Görüntüleri Yazma 212
ISO Görüntüsünü Doğrudan Bağlama 212
Yeniden Yazılabilir CD-ROM'u Boşaltma 213
Görüntü Yazma 213
Verileri Doğrulama 213
Özet....214
Daha Fazla Okuma 215

16 – Ağ oluşturma ***216***

Ağı İnceleme ve İzleme 217
ping 217
traceroute 218
ip 219
Ağ Üzerinden Dosya Aktarımı 221
ftp 221
lftp – Daha İyi Bir ftp 223
curl - Bir URL'yi aktarın 224
wget - Etkileşimli olmayan ağ indiricisi 224
Uzak Ana Bilgisayarlarla Güvenli İletişim 225
ssh 225
SSH ile Tünelleme 229
scp ve sftp 229
Windows için bir SSH istemcisi mi? 231
Özet 231
Daha Fazla Okuma 231

17 – Dosya Arama ***232***

locate – Dosyaları Kolay Yoldan Bulma 232
Locate Veritabanı Nereden Geliyor? 234
bul – Dosyaları Zor Yoldan Bulma 234
Testler 235
Operatörler 237
Önceden Tanımlanmış Eylemler 240
Kullanıcı Tanımlı Eylemler 242
Verimliliği Artırma 243
xargs 244
Komik Dosya Adlarıyla Başa Çıkma 244
Oyun Alanına Dönüş 245
Seçenekler 247
Özet 248
Daha Fazla Okuma 248

18 – Arşivleme ve Yedekleme ***249***

Dosyaları Sıkıştırma 249
gzip 250
bzip2 252
Sıkıştırıcı Kompulsif Olmayın 253
Dosyaları Arşivleme 253
tar 254
zip 259
Dosya ve Dizinleri Senkronize Etme 261
Ağ Üzerinden rsync Kullanımı 264
Özet 265
Daha Fazla Okuma 265

19 – Düzenli İfadeler ***267***

Düzenli İfadeler Nedir? 267
grep 267
Meta karakterler ve Literaler 270

Herhangi Bir Karakter 270
Bağlantı Noktaları....271
Çapraz Bulmaca Yardımcısı....272
Parantez İfadeleri ve Karakter Sınıfları....272
Olumsuzlama....273
Geleneksel Karakter Aralıkları....273
POSIX Karakter Sınıfları....274
Geleneksel Karşılaştırma Sırasına Geri Dönüş....277
POSIX Temel ve Genişletilmiş Düzenli İfadeler....278
POSIX....279
Alternatif....279
Nicelik belirteçleri....280
? - Bir öğeyi sıfır veya bir kez eşleştir....280
* - Bir öğeyi sıfır veya daha fazla kez eşleştirin....281
+ - Bir öğeyi bir veya daha fazla kez eşleştirin....282
{ } - Bir öğeyi belirli sayıda eşleştir....282
Düzenli İfadeleri Kullanma....283
grep ile Telefon Listesini Doğrulama....283
find ile Çirkin Dosya Adlarını Bulma....285
Locate ile Dosya Arama....285
less ve vim ile metin arama....286
Özet....287
Daha Fazla Okuma....288

20 – Metin İşleme*....*289

Metin Uygulamaları....289
Belgeler....290
Web Sayfaları....290
E-posta....290
Yazıcı Çıktısı....290
Program Kaynak Kodu....290
Eski Dostları Yeniden Ziyaret Etmek....291
cat....291
MS-DOS Metni ve Unix Metni Karşılaştırması....292
sıralama....293
uniq....300
Dilimleme ve Küp Küp Doğrama....302
kesmek....302
Sekmeleri Genişletme....305
yapıştır....306
birleştir....307
tac....310
rev....310
Metin Karşılaştırma....311
comm....311
diff....312
yama....315
Anında Düzenleme....316
tr....316
ROT13: Çok da Gizli Olmayan Şifre Çözücü Yüzük....317
sed....318
sed'i sevenler şunu da sever.....327
aspell....327

Özet....331
Daha Fazla Okuma....331
Ekstra Kredi....332

21 – Çıktı Biçimlendirme....333

Basit Biçimlendirme Araçları....333
nl – Sayı Çizgileri....333
fold – Her satırı belirli bir uzunlukta sarmala....337
fmt – Basit bir metin biçimlendirici....338
pr – Yazdırma için Metni Biçimlendir....341
printf – Verileri Biçimlendirme ve Yazdırma....342
Belge Biçimlendirme Sistemleri....346
groff....347
Özet....353
Daha Fazla Okuma....353

22 – Baskı....355

Baskının Kısa Tarihi....355
Karanlık Çağlarda Baskıcılık....355
Karakter Tabanlı Yazıcılar....356
Grafik Yazıcılar....357
Linux ile yazdırma....358
Yazdırma için Dosyaları Hazırlama....358
pr – Metin Dosyalarını Yazdırma için Dönüştürme....358
Yazıcıya Yazdırma İşini Gönderme....360
lpr – Dosyaları Yazdırma (Berkeley Stili)....360
lp – Dosyaları Yazdır (System V Stili)....361
Başka bir seçenek: a2ps....362
Yazdırma İşlerini İzleme ve Denetleme....365
lpstat – Yazdırma Sisteminin Durumunu Görüntüleme....365
lpq – Yazıcı Kuyruğu Durumunu Görüntüleme....366
lprm / cancel – Yazdırma İşlerini İptal Et....367
Özet....367
Daha Fazla Okuma....367

23 – Programları Derleme....368

Derleme Nedir?....368
Tüm Programlar Derlenir mi?....369
C Programını Derleme....370
Kaynak Kodunu Elde Etme....370
Kaynak Ağacını İnceleme....372
Programı Oluşturma....374
Programı Yükleme....378
Özet....379
Daha Fazla Okuma....379

Bölüm 4 – Kabuk Betikleri Yazma....381

24 – İlk Komut Dosyanızı Yazma....382

Kabuk Betikleri Nedir?....382
Shell Komut Dosyası Nasıl Yazılır?....382

Komut Dosyası Biçimi ... 383
Yürütülebilir İzinler ... 384
Komut Dosyası Konumu ... 384
Komut Dosyaları için Uygun Konumlar ... 386
Daha Fazla Biçimlendirme Püf Noktası ... 386
Uzun Seçenek Adları ... 386
Girinti ve Satır Devamı ... 386
Vim'i Komut Dosyası Yazmak İçin Yapılandırma ... 387
Özet ... 388
Daha Fazla Okuma ... 388

25 – Bir Projeye Başlamak ... ***389***

İlk Aşama: Minimal Belge ... 389
İkinci Aşama: Az Miktarda Veri Ekleme ... 391
Değişkenler ve Sabitler ... 392
Değişkenlere ve Sabitlere Değer Atama ... 395
Burada Belgeler ... 397
Özet ... 400
Daha Fazla Okuma ... 400

26 – Yukarıdan Aşağıya Tasarım ... ***401***

Kabuk İşlevleri ... 40
Yerel Değişkenler ... 405
Kabuk İşlevleri ve Yeniden Yönlendirme ... 406
Komut Dosyalarını Çalışır Halde Tutma ... 408
.bashrc Dosyanızdaki Kabuk İşlevleri ... 411
Özet ... 411
Daha Fazla Okuma ... 411

27 – Akış Kontrolü: if ile Dallanma ... ***412***

if ... 412
Çıkış Durumu ... 413
Test kullanımı ... 415
Dosya İfadeleri ... 415
Dize İfadeleri ... 418
Tamsayı İfadeleri ... 419
Testin Daha Modern Bir Versiyonu ... 421
(( )) - Tamsayılar için Tasarlanmıştır ... 42
İfadelerin birleştirilmesi ... 423
Taşınabilirlik, küçük zihinlerin hobgoblinidir. ... 426
Kontrol Operatörleri: Dallanmanın Başka Bir Yolu ... 426
Özet ... 427
Daha Fazla Okuma ... 428

28 – Klavye Girişini Okuma ... ***429***

read – Standart Girişten Değerleri Oku ... 430
Seçenekler ... 432
IFS ... 434
Pip ile okuyun ... 436
Girişi Doğrulama ... 437
Menüler ... 438

Özet....440
Ekstra Kredi....440
Daha Fazla Okuma....440

29 – Akış Kontrolü: while / until ile döngü oluşturma....442

Döngü....44
while....443
break ve devam....445
seç....447
kadar....450
Döngülerle Dosya Okuma....451
Özet....452
Daha Fazla Okuma....45

30 – Sorun Giderme....453

Sözdizimsel Hatalar....45
Eksik Tırnak İşaretleri....454
Eksik veya Beklenmedik Belirtiler....45
Beklenmeyen Genişlemeler....455
Mantıksal Hatalar....457
Savunma Programlama....458
set -e, set -u ve set -o pipefail....459
ShellCheck sizin dostunuzdur....460
Dosya Adlarına Dikkat Edin....460
Taşınabilir Dosya Adları....461
Girişi Doğrulama....461
Tasarım Zamanın Bir Fonksiyonudur....462
Test....462
Test Durumları....463
Hata Ayıklama....463
Sorunlu Alanı Bulma....464
İzleme....464
Yürütme Sırasında Değerleri İnceleme....467
Özet....467
Daha Fazla Okuma....467

31 – Akış Kontrolü: Case ile Dallanma....469

case....46
Desenler....471
Birden Fazla Eylem Gerçekleştirme....473
Özet....475
Daha Fazla Okuma....475

32 – Konumsal Parametreler....476

Komut Satırına Erişim....476
Argüman Sayısını Belirleme....477
shift – Birçok Argümana Erişim Sağlama....478
Basit Uygulamalar....480
Kabuk İşlevlerinde Konumsal Parametreleri Kullanma....480
Konumsal Parametreleri *Toplu Olarak* İşleme....481
Daha Tamamlanmış Bir Uygulama....484
getopts Seçeneği....485

Etkileşimli Mod 487
Dosya Çıktısı 488
Özet 489
Daha Fazla Okuma 492

33 – Akış Kontrolü: for ile Döngü ***493***

for: Geleneksel Kabuk Formu 493
Neden i? 496
için: C Dili Formu 496
Özet 498
Daha Fazla Okuma 499

34 – Diziler ve Sayılar ***500***

Parametre Genişletme 500
Temel Parametreler 500
Boş Değişkenleri Yönetmek için Genişletmeler 501
Değişken Adlarını Döndüren Genişletmeler 503
Dize İşlemleri 503
Büyük/Küçük Harf Dönüştürme 507
Aritmetik Değerlendirme ve Genişletme 509
Sayı Tabanları 509
Tekli İşlemciler 510
Basit Aritmetik 510
Atama 511
Bit İşlemleri 514
Mantık 515
Virgül İşleci 518

bc – Keyfi Hassasiyetli Hesap Makinesi Dili 518
bc Kullanımı 519
Örnek Bir Komut Dosyası 52
Özet 521
Ekstra Kredi 521
Daha Fazla Okuma 521

35 – Diziler ***523***

Diziler Nedir? 523
Dizi Oluşturma 523
Dizilere Değer Atama 524
Dizi Öğelerine Erişme 525
Dizi İşlemleri 527
Dizinin Tüm İçeriğini Çıktılamak 527
Dizi Öğelerinin Sayısını Belirleme 528
Bir Dizinin Kullandığı Alt İndeksleri Bulma 528
read -a ile Dizi Elemanlarını Atama 529
Dizinin Sonuna Öğeler Ekleme 52
Bir dosyayı diziye okuma 529
Diziyi Dilimleme 531
Diziyi Sıralama 532
Diziyi Silme 533
İlişkisel Diziler 534
Birden Çok Boyutu Simüle Etmek İçin İlişkisel Diziler Kullanma 536

Özet....537
Daha Fazla Okuma....537

36 – Exotica*....*538

Grup Komutları ve Alt Kabuklar....538
İşlem Değiştirme....543
eval ile Komut Oluşturma....545
eval ile Dikkatli Olun....547
Bir Wordle Yardımcısı....547
Tuzaklar....551
Geçici Dosyalar....554
Asenkron Yürütme....555
bekletme....555
Adlandırılmış Borular....557
Adlandırılmış Boru Kurma....557
Adlandırılmış Boruların Kullanılması....558
Özet....558
Daha Fazla Okuma....558

Dizin*....*560

Kolofon*....*573

Karen için

Giriş

Size bir hikaye anlatmak istiyorum.

Hayır, 1991 yılında Linus Torvalds'ın Linux çekirdeğinin ilk sürümünü nasıl yazdığına dair hikayeyi anlatmayacağım. Bu hikayeyi birçok Linux kitabında okuyabilirsiniz. Birkaç yıl önce Richard Stallman'ın, Unix benzeri ücretsiz bir işletim sistemi oluşturmak için GNU Projesi'ni nasıl başlattığına dair hikayeyi de anlatmayacağım. Bu da önemli bir hikaye, ancak diğer Linux kitaplarının çoğunda bu hikaye de yer alıyor.

Hayır, size bilgisayarınızın kontrolünü nasıl geri alacağınızın hikayesini anlatmak istiyorum.

1970'lerin sonunda üniversite öğrencisiyken bilgisayarlarla çalışmaya başladığımda, bir devrim yaşanıyordu. Mikroişlemcinin icadı, sizin ve benim gibi sıradan insanların bir bilgisayara sahip olmasını mümkün kılmıştı. Bugün birçok insan, sadece büyük şirketler ve büyük hükümetlerin tüm bilgisayarları işlettiği bir dünyanın nasıl olduğunu hayal etmekte zorlanıyor. Şöyle diyelim, pek bir şey yapamazdınız.

Bugün dünya çok farklı. Bilgisayarlar, küçük kol saatlerinden dev veri merkezlerine ve aradaki her şeye kadar her yerde. Her yerde bulunan bilgisayarlara ek olarak, bunları birbirine bağlayan her yerde bulunan bir ağ da var. Bu, kişisel güçlenme ve yaratıcı özgürlük açısından muhteşem bir yeni çağ yarattı, ancak son birkaç on yılda başka bir şey daha oluyor. Birkaç dev şirket, dünyadaki bilgisayarların çoğu üzerinde kontrolünü dayatıyor ve bu bilgisayarlarla neler yapabileceğinizi ve yapamayacağınızı belirliyor. Neyse ki, dünyanın her yerinden insanlar bu konuda bir şeyler yapıyorlar. Kendi yazılımlarını yazarak bilgisayarlarının kontrolünü elinde tutmak için mücadele ediyorlar. Linux'u geliştiriyorlar.

Birçok kişi Linux ile ilgili olarak "özgürlük"ten bahsediyor, ancak çoğu kişinin bu özgürlüğün gerçekte ne anlama geldiğini bildiğini sanmıyorum. Özgürlük, bilgisayarınızın ne yapacağına karar verme gücüdür ve bu özgürlüğe sahip olmanın tek yolu, bilgisayarınızın ne yaptığını bilmektir. Özgürlük, sırları olmayan, yeterince ilgilenirseniz her şeyi öğrenebileceğiniz bir bilgisayardır.

Neden Komut Satırı Kullanmalı?

Filmlerde, 30 saniyeden daha kısa sürede ultra güvenli askeri bilgisayara girebilen "süper hacker"ın bilgisayarın başına oturduğunda fareye hiç dokunmadığını fark ettiniz mi? Bunun nedeni, film yapımcılarının, biz insanların, bir bilgisayarda bir şeyi gerçekten yapmanın tek yolunun klavyeyi kullanmak olduğunu içgüdüsel olarak bildiğimizi fark etmeleridir.

klavyeyi kullanarak olduğunu içgüdüsel olarak bildiğimizi fark ettikleri içindir!

Günümüzde çoğu bilgisayar kullanıcısı yalnızca *grafik kullanıcı arayüzüne* (GUI) aşinadır ve satıcılar ve uzmanlar tarafından *komut satırı arayüzünün* (CLI) korkutucu bir şey olduğu öğretilmiştir. Bu talihsiz bir durumdur, çünkü iyi bir komut satırı arayüzü, yazılı kelimelerin insanlar için olduğu gibi, bilgisayar ile iletişim kurmanın harika bir yoludur. "Grafiksel kullanıcı arayüzleri kolay işleri kolaylaştırırken, komut satırı arayüzleri zor işleri mümkün kılar" denir ve bu bugün de hala geçerlidir.

Linux, Unix işletim sistemi ailesini örnek aldığı için, Unix ile aynı zengin komut satırı araçları mirasını paylaşır. Unix, grafiksel kullanıcı arayüzünün yaygın olarak benimsenmesinden önce, 1980'lerin başında (ilk olarak on yıl önce geliştirilmiş olmasına rağmen) öne çıktı ve bunun sonucunda, bunun yerine kapsamlı bir komut satırı arayüzü geliştirdi. Aslında, Linux'u erken benimseyenlerin onu, örneğin Windows NT'ye tercih etmelerinin en güçlü nedenlerinden biri, "zor görevleri mümkün kılan" güçlü komut satırı arayüzüydü.

Bu Kitabın Konusu

Bu kitap, Linux komut satırında "yaşamaya" dair geniş bir genel bakış sunmaktadır. Shell programı `bash` gibi tek bir programa odaklanan bazı kitapların aksine, bu kitap komut satırı arayüzüyle daha geniş bir anlamda nasıl başa çıkılacağını anlatmaya çalışacaktır. Tüm bunlar nasıl çalışır? Neler yapabilir? En iyi kullanım şekli nedir?

Bu kitap, Linux sistem yönetimi hakkında bir kitap değildir. Komut satırı hakkında yapılan ciddi tartışmalar her zaman sistem yönetimi konularına yönelse de, bu kitap sadece birkaç yönetim konusuna değinmektedir. Bununla birlikte, ciddi sistem yönetimi görevleri için gerekli olan komut satırı kullanımına ilişkin sağlam bir temel sağlayarak okuyucuyu ek çalışmalara hazırlayacaktır.

Bu kitap, Linux odaklıdır. Diğer birçok kitap, genel Unix ve macOS gibi diğer platformları da dahil ederek çekiciliğini artırmaya çalışır. Bunu yaparken, içeriğini "sulandırarak" yalnızca genel konuları ele alırlar. Bu kitap ise yalnızca güncel Linux dağıtımlarını ele alır. İçeriğin yüzde doksan beşi, diğer Unix benzeri sistemlerin kullanıcıları için de yararlıdır, ancak bu kitap, modern Linux komut satırı kullanıcılarına yönelik olarak yazılmıştır.

Bu Kitabı Kimler Okumalı

Bu kitap, diğer platformlardan geçiş yapmış yeni Linux kullanıcıları içindir. Muhtemelen siz, Microsoft Windows'un bir sürümünün "ileri düzey kullanıcısı"sınız. Belki patronunuz size bir Linux sunucusunu yönetmenizi söyledi veya Raspberry Pi gibi tek kartlı bilgisayarların (SBC) heyecan verici yeni dünyasına giriyorsunuz. Belki de tüm güvenlik sorunlarından bıkmış ve Linux'u denemek isteyen bir masaüstü kullanıcısısınız. Sorun değil. Herkes buraya hoş geldiniz.

Bununla birlikte, Linux'u öğrenmenin kısa yolu yoktur. Komut satırını öğrenmek zordur ve gerçek bir çaba gerektirir. Çok zor olduğu için değil, çok *geniş* olduğu için zordur. Ortalama bir Linux sisteminde, komut satırında kullanabileceğiniz binlerce program vardır. Kendinizi uyarılmış sayın; komut satırını öğrenmek sıradan bir çaba değildir.

Öte yandan, Linux komut satırını öğrenmek son derece ödüllendiricidir. Şu anda kendinizi "ileri düzey kullanıcı" olarak görüyorsanız, bekleyin. Henüz gerçek gücün ne olduğunu bilmiyorsunuz. Ve diğer birçok bilgisayar becerisinin aksine, komut satırı bilgisi uzun ömürlüdür. Bugün öğrendiğiniz beceriler 10 yıl sonra da hala yararlı olacaktır. Komut satırı zamanın testinden geçmiştir.

Ayrıca, programlama deneyiminiz olmadığı varsayılmaktadır, ancak endişelenmeyin, bu konuda da size yardımcı olacağız.

Bu Kitapta Neler Var

Bu materyal, yanınızda oturup size rehberlik eden bir öğretmen gibi, özenle seçilmiş bir sırayla sunulmaktadır. Birçok yazar bu materyali "sistematik" bir şekilde ele alır ve her konuyu sırayla kapsamlı bir şekilde ele alır. Bu, yazarın bakış açısından mantıklıdır, ancak yeni kullanıcılar için çok kafa karıştırıcı olabilir.

Diğer bir amaç ise, Windows'un düşünce tarzından farklı olan Unix'in düşünce tarzını size tanıtmaktır. Bu süreçte, bazı şeylerin neden bu şekilde çalıştığını ve nasıl bu hale geldiğini anlamanıza yardımcı olmak için birkaç yan geziye çıkacağız. Linux sadece bir yazılım parçası değildir; aynı zamanda kendi dili ve tarihi olan daha büyük Unix kültürünün küçük bir parçasıdır. Birkaç şikayet de ekleyebilirim.

Bu kitap, komut satırı deneyiminin bazı yönlerini ele alan dört bölüme ayrılmıştır:

- **Bölüm 1 – Kabuğu Öğrenmek,** komutların yapısı, dosya sistemi gezinme, komut satırı düzenleme ve komutlar için yardım ve belgeler bulma gibi komut satırının temel dilini keşfetmeye başlar.
- **Bölüm 2 – Yapılandırma ve Ortam,** komut satırından bilgisayarın çalışmasını kontrol eden yapılandırma dosyalarının düzenlenmesini ele alır.
- **Bölüm 3 – Yaygın Görevler ve Temel Araçlar**, komut satırından yaygın olarak gerçekleştirilen birçok sıradan görevi incelemektedir. Linux gibi Unix benzeri işletim sistemleri, veriler üzerinde güçlü işlemler gerçekleştirmek için kullanılan birçok "klasik" komut satırı programı içerir.
- **Bölüm 4 – Kabuk Betikleri Yazma**, birçok yaygın bilgisayar görevini otomatikleştirmek için kullanılan, temel düzeyde ancak öğrenmesi kolay bir teknik olan kabuk programlamayı tanıtmaktadır. Kabuk programlamayı öğrenerek, diğer birçok programlama diline uygulanabilecek kavramlara aşina olacaksınız.

Bu Kitabı Nasıl Okumalı

Kitabın başından başlayın ve sonuna kadar okuyun. Bu kitap bir referans kitabı olarak yazılmamıştır, daha çok başlangıcı, ortası ve sonu olan bir hikaye gibidir.

Ön Koşullar

Bu kitabı kullanmak için tek ihtiyacınız olan, çalışan bir Linux kurulumudur. Bunu iki yoldan biriyle elde edebilirsiniz:

1. **(Çok yeni olmayan) bir bilgisayara Linux kurun.** Hangi dağıtımı seçeceğiniz önemli değildir, ancak günümüzde çoğu kişi Ubuntu, Fedora veya OpenSUSE ile başlamaktadır. Şüpheniz varsa, önce Ubuntu'yu deneyin. Modern bir Linux dağıtımını kurmak, donanımınıza bağlı olarak çok kolay veya çok zor olabilir. Birkaç yıllık, en az 2 GB RAM ve 6 GB boş sabit disk alanına sahip bir masaüstü bilgisayar öneririm. Mümkünse dizüstü bilgisayarları ve kablosuz ağları kullanmaktan kaçının, çünkü bunlar genellikle çalıştırmak daha zordur.
2. **"Canlı CD" veya USB flash sürücü kullanın.** Birçok Linux dağıtımında yapabileceğiniz harika şeylerden biri, bunları hiç yüklemeden doğrudan CD-ROM veya USB flash sürücüden çalıştırmaktır. BIOS ayarlarına girin ve bilgisayarınızı CD-ROM sürücüsünden veya USB aygıtından önyükleme yapacak şekilde ayarlayın ve yeniden başlatın. Bu yöntemi kullanmak, yüklemeden önce bir bilgisayarın Linux uyumluluğunu test etmek için harika bir yoldur. Dezavantajı, sabit sürücünüze Linux yüklemeye kıyasla çok yavaş olabilmesidir. Ubuntu ve Fedora (diğerleri arasında) canlı sürümlerine sahiptir.

Linux'u nasıl kurarsanız kurun, bu kitaptaki dersleri uygulamak için ara sıra süper kullanıcı (yani yönetici) ayrıcalıklarına ihtiyacınız olacaktır.

Çalışan bir kurulumunuz olduktan sonra, okumaya başlayın ve kendi bilgisayarınızla birlikte ilerleyin. Bu kitaptaki materyallerin çoğu "uygulamalı"dır, bu yüzden oturun ve yazmaya başlayın!

Neden "GNU/Linux" Demiyorum

Bazı çevrelerde, Linux işletim sistemini "GNU/Linux işletim sistemi" olarak adlandırmak politik olarak doğrudur. "Linux" ile ilgili sorun, çok sayıda farklı kişi tarafından geniş ve dağıtık bir geliştirme çabasıyla yazıldığı için, onu adlandırmanın tamamen doğru bir yolu olmamasıdır. Teknik olarak konuşursak, Linux işletim sisteminin çekirdeğinin adıdır, daha fazlası değildir. Çekirdek, işletim sistemini çalıştırdığı için elbette çok önemlidir, ancak tam bir işletim sistemi oluşturmak için yeterli değildir.

Serbest Yazılım hareketini kuran, Serbest Yazılım Vakfı'nı başlatan, GNU Projesi'ni oluşturan,

GNU C Derleyicisinin (gcc) ilk sürümünü yarattı, GNU Genel Kamu Lisansını (GPL) oluşturdu, vb. vb. vb. GNU Projesinin katkılarını doğru bir şekilde yansıtmak için buna "GNU/Linux" denilmesini *ısrarla* talep ediyor. GNU Projesi Linux çekirdeğinden önce ortaya çıkmış ve projenin katkıları son derece takdir edilmeye değer olsa da, bunları isme eklemek, önemli katkılarda bulunan diğer herkese haksızlık olur. Ayrıca, çekirdek ilk olarak önyükleme yaptığı ve diğer her şey onun üzerinde çalıştığı için, "Linux/GNU"nun teknik olarak daha doğru olacağını düşünüyorum.

Popüler kullanımda, "Linux" çekirdeği ve tipik bir Linux dağıtımında bulunan diğer tüm özgür ve açık kaynaklı yazılımları, yani sadece GNU bileşenlerini değil, tüm Linux ekosistemini ifade eder. İşletim sistemi pazarı, DOS, Windows, macOS, Solaris, Irix ve AIX gibi tek kelimelik isimleri tercih ediyor gibi görünüyor. Ben de popüler formatı kullanmayı tercih ettim. Ancak, bunun yerine "GNU/Linux" kullanmayı tercih ediyorsanız, lütfen bu kitabı okurken zihninizde arama ve değiştirme işlemi yapın. Benim için sorun olmaz.

Yedinci İnternet Sürümünde Yenilikler

Kabuk kendisi yaklaşık 10 yılda bir büyük sürüm güncellemeleri alırken, donanım ve araçlar sürekli olarak gelişmektedir. *The Linux Command Line'ın* bu baskısı, komut satırı ortamındaki değişiklikleri yansıtacak şekilde yeniden modernize edilmiştir. Çok sayıda küçük düzenleme ve düzeltme yapılmıştır. Bu baskı, No Starch Press tarafından yayınlanan basılı versiyon olan *The Linux Command Line: A Complete Introduction, Third Edition* ile de uyumludur. Değişikliklerin ayrıntılı listesi için LinuxCommand.org adresinde bulunan sürüm notlarına bakın. Bu baskıda yeni olan bir diğer özellik de kitaptaki örnek komut dosyalarının bir koleksiyonudur. Bu da LinuxCommand.org adresinden indirilebilir.

Teşekkürler

Bu kitabın yayınlanmasına katkıda bulunan aşağıdaki kişilere teşekkür etmek istiyorum:

Jenny Watson, Wiley Publishing'in satın alma editörü, bana bir kabuk betikleme kitabı yazmamı öneren kişi.

John C. Dvorak, tanınmış köşe yazarı ve uzman. Video podcast'i "Cranky Geeks"in bir bölümünde, Dvorak yazma sürecini şöyle tanımladı: "Cehennem. Günde 200 kelime yazın, bir yıl içinde romanınız hazır." Bu tavsiye sayesinde, kitabım tamamlanana kadar her gün bir sayfa yazdım.

Dmitri Popov, Free Software Magazine dergisinde "Writer ile Kitap Şablonu Oluşturma" başlıklı bir makale yazdı ve bu makale, metni yazmak için OpenOffice.org Writer'ı (ve daha sonra LibreOffice Writer'ı) kullanmam için bana ilham verdi. Sonuçta, bu harika bir şekilde işe yaradı.

Mark Polesky, İlk Baskı metnini olağanüstü bir şekilde gözden geçirdi ve test etti.

Jesse Becker, Tomasz Chrzczonowicz, Michael Levin ve Spence Miner de metni test etti ve

ilk baskının bazı bölümlerini inceledi.

Karen M. Shotts, orijinal metni düzenleyerek benim sözde İngilizcemi cilalamak için saatlerce emek harcadı.

Yedinci İnternet Sürümü

Yedinci İnternet Sürümüne dahil edilen değerli geri bildirimlerde bulunan aşağıdaki kişilere özel teşekkürlerimi sunarım: Vitor Centeio, Elmar Deininger, Francesco Di Viesto, Jaroslaw Kolosowski, Kayck Matias ve Wang Zheng.

Önceki İnternet Sürümleri

Önceki sürümlerde yer alan değerli geri bildirimleri sağlayan aşağıdaki kişilere özel teşekkürlerimizi sunarız: Ala'a Ali, Adrian Arpidez, Mikey Barboza, Jesse Becker, Emanuele Bernardi, Andreas Bjørnestad, Hu Bo, Steve Bragg, John Burns, Heriberto Cantú, Enzo Cardinal, Paolo Casati, Tomasz Chrzczonowicz, Richard Cooke, Ethan Dowlatshah, Lixin Duan, Joshua Escamilla, Marc Evans, Ryan Flynn, Bruce Fowler, Devin Harper, Jørgen Heitmann, Janrodion, Jonathan Jones, Sunil Joshi, Ma Jun, Eric Kammerer, Robert Kennington, Seth King, Chris Knight, Jaroslaw Kolosowski, Klaus M. Körmendi, Jim Kovacs, Michael Levin, Bartłomiej Majka, Bashar Maree, Frank McTipps, Vladimir Milovanović, Sea Monkey, Tim Nelson, Mike O'Donnell, Ok-tay-Akin Okutan, Nick Owens, Justin Page, Michael Parrish, Esra Purba, Parviz Ra-soulipour, Amir Razqandi, Patrick, Waldo Ribeiro, Pat Roche, Nick Rose, Satej Kumar Sahu, Avid Seeker, Mikhail Sizov, Ben Slater, Pickles Spill, Gabriel Stutzman, Pooya Taherkhani, Francesco Turco, Wolfram Volpi, Boyang Wang, Carl Westman, John Wiersba, Valter Wierzba ve Christian Wuethrich.

Son olarak, bana çok sayıda nazik e-posta gönderen LinuxCommand.org okuyucularına da çok teşekkür ederim. Onların teşvikleri, gerçekten önemli bir şeyin peşinde olduğumu düşünmemi sağladı!

Görüşlerinizi bekliyoruz!

Bu kitap, birçok açık kaynaklı yazılım projesi gibi devam eden bir projedir. Teknik bir hata bulursanız, lütfen bana şu adrese yazın:

bshotts@users.sourceforge.net

Okuduğunuz kitabın tam baskısını belirtmeyi unutmayın. Değişiklikleriniz ve önerileriniz gelecek sürümlerde yer alabilir.

Daha Fazla Okuma

- Yukarıda bahsedilen ünlü kişilerle ilgili bazı Wikipedia makaleleri:
  https://en.wikipedia.org/wiki/Linus_Torvalds
  https://en.wikipedia.org/wiki/Richard_Stallman

- Özgür Yazılım Vakfı ve GNU Projesi:
  https://en.wikipedia.org/wiki/Free_Software_Foundation
  https://www.fsf.org
  https://www.gnu.org
- Richard Stallman, "GNU/Linux" isimlendirme konusunda kapsamlı yazılar yazmıştır: https://www.gnu.org/gnu/why-gnu-linux.html
  https://www.gnu.org/gnu/gnu-linux-faq.html#tools

Bölüm 1 – Kabuğu Öğrenmek

1 – Kabuk Nedir?

Komut satırından bahsettiğimizde, aslında *kabuğu* kastediyoruz. Kabuk, klavye komutlarını alıp bunları işletim sistemine ileterek yürütmesini sağlayan bir programdır. Hemen hemen tüm Linux dağıtımları, GNU Projesi'nden `bash` adlı bir kabuk programı sağlar. "Bash" adı, "Bourne Again SHell"in kısaltmasıdır ve `bash'ın`, Steve Bourne tarafından yazılan orijinal Unix kabuk programı olan `sh'nin` geliştirilmiş bir versiyonu olduğunu ifade eder.

Terminal Emülatörleri

Grafik kullanıcı arayüzü (GUI) kullanırken, kabukla etkileşim kurmak için *terminal emülatörü* adı verilen başka bir programa ihtiyacımız vardır. Masaüstü menülerimize bakarsak, muhtemelen bir tane bulabiliriz. KDE `konsole` kullanır, GNOME ise `gnome-terminal` kullanır, ancak menümüzde muhtemelen sadece "terminal" olarak adlandırılır. Linux için bir dizi başka terminal emülatörü de mevcuttur, ancak hepsi temelde aynı şeyi yapar: kabuğa erişim sağlar. Muhtemelen, sahip olduğu özelliklerin sayısına göre bir veya başka bir terminal emülatörünü tercih edeceksiniz.

İlk Tuş Vuruşlarınızı Yapın

Öyleyse başlayalım. Terminal emülatörünü başlatın! Açıldığında, şuna benzer bir şey görmelisiniz:

```
[me@linuxbox ~]$
```

Buna *kabuk istemcisi* denir ve kabuk girişi kabul etmeye hazır olduğunda görünür. Dağıtımına bağlı olarak görünümü biraz değişebilir, ancak genellikle *kullanıcı adınız@makineadı*, ardından geçerli çalışma dizini (bununla ilgili birazdan daha fazla bilgi vereceğiz) ve dolar işareti içerir.

Not: Komut isteminin son karakteri dolar işareti yerine pound işareti ("#") ise

---

işareti ise, terminal oturumu *süper kullanıcı* ayrıcalıklarına sahiptir. Bu, ya root kullanıcısı olarak oturum açtığımız ya da süper kullanıcı (yönetici) ayrıcalıkları sağlayan bir terminal emülatörü seçtiğimiz anlamına gelir.

Şimdiye kadar her şey yolunda gidiyorsa, biraz yazmayı deneyelim. Komut istemine şöyle anlamsız bir şeyler yazın:

```
[me@linuxbox ~]$ kaekfjaeifj
```

Bu komut anlamsız olduğu için, kabuk bize bunu bildirir ve başka bir şans verir.

```
bash: kaekfjaeifj: komut bulunamadı
[me@linuxbox ~]$
```

Komut Geçmişi

Yukarı ok tuşuna basarsak, önceki komut `kaekfjaeifj'nin` komut isteminden sonra yeniden göründüğünü göreceğiz. Buna *komut geçmişi* denir. Çoğu Linux dağıtımı varsayılan olarak son 1.000 komutu hatırlar. Aşağı ok tuşuna basarsak önceki komut kaybolur.

İmleç Hareketi

Yukarı ok tuşuna tekrar basarak önceki komutu geri çağırın. Sol ve sağ ok tuşlarını denersek, imleci komut satırının herhangi bir yerine nasıl konumlandırabileceğimizi göreceğiz. Bu, komutları düzenlemeyi kolaylaştırır.

Fare ve Odak Hakkında Birkaç Söz

Kabuk tamamen klavyeyle ilgili olsa da, terminal emülatörünüzle fareyi de kullanabilirsiniz. Pencere sistemine (GUI'yi çalıştıran temel motor) yerleşik bir mekanizma, hızlı kopyala-yapıştır tekniğini destekler. Sol fare düğmesini basılı tutarak fareyi metnin üzerine sürükleyerek (veya bir kelimeye çift tıklayarak) metni vurguladığınızda, metin pencere sistemi tarafından tutulan bir arabelleğe kopyalanır. Orta fare düğmesine basıldığında metin imlecin bulunduğu yere yapıştırılır. Deneyin. Çoğu masaüstü ortamında çalışır.

Not: Terminal penceresi içinde kopyalama ve yapıştırma işlemlerini gerçekleştirmek için `Ctrl-c` ve `Ctrl-v` tuşlarını kullanmaya kalkışmayın. Bu tuşlar çalışmaz. Bu kontrol kodları, kabuk için farklı anlamlara sahiptir ve Microsoft Windows'un piyasaya sürülmesinden yıllar önce atanmıştır. Terminal penceresini kullanırken bunun yerine `Shift-Ctrl-c` ve `Shift-Ctrl-v` tuşlarını kullanın.

Grafik masaüstü ortamınız (büyük olasılıkla KDE veya GNOME), Windows gibi davranmak amacıyla, muhtemelen *odak politikasını* "odaklamak için tıklayın" olarak ayarlamıştır. Bu, bir pencerenin odaklanabilmesi (aktif hale gelmesi) için üzerine tıklamanız gerektiği anlamına gelir. Bu, geleneksel pencere sistemi davranışı olan "odak fareyi takip eder" ile çelişir; bu davranışta, fare pencerenin üzerine getirildiğinde pencere odaklanır. Pencere, üzerine tıklayana kadar ön plana çıkmaz, ancak giriş alabilir. Odak politikasını "odak fareyi takip eder" olarak ayarlamak, kopyala ve yapıştır tekniğini daha da kullanışlı hale getirecektir. Mümkünse deneyin (bazı masaüstü ortamları artık bunu desteklemiyor olsa da). Bence denerseniz, bunu tercih edeceksiniz. Bu ayarı, pencere yöneticinizin yapılandırma programında bulabilirsiniz.

Bazı Basit Komutları Deneyin

Terminal emülatörümüze metin girmeyi öğrendiğimize göre, şimdi birkaç basit komut deneyelim. Güncel saati ve tarihi gösteren `date` komutuyla başlayalım.

```
[me@linuxbox ~]$ date
8 Mart Perşembe 15:09:41 EST 2025
```

Bir başka kullanışlı komut da `uptime` komutudur. Bu komut, sistemin ne kadar süredir çalıştığını ve çeşitli zaman aralıklarında çalışan işlemlerin ortalama sayısını gösterir.

```
[me@linuxbox ~]$ uptime
15:12:22 3 gün, 23:40, 7 kullanıcı, ortalama yük: 0,37, 0,37, 0,64
```

Disk sürücülerimizdeki mevcut boş alan miktarını görmek için `df komutunu` girin.

```
[me@linuxbox ~]$ df
Dosya sistemi                 1K      Kullanılan Kullanılabilir
/dev/sda2                bloklar   5012392 9958716  34% /
                        15115452
```

```
/dev/sda5              59631908   26545424   30008432  %47 /home
/dev/sda1                147764      17370     122765  %13 /boot
tmpfs                    256856          0     256856   0  /dev/shm
```

Aynı şekilde, boş bellek miktarını görüntülemek için `free` komutunu girin.

```
[me@linuxbox ~]$ serb
est
             toplam       kullanılmış       boş    paylaşılan   tamponlar     önbellek
Bellek:      513712       503976         9736           0        5312      122916
-/+ tampon: 375748                      94753
Takas:      1052248       104712
```

Terminal Oturumunu Sonlandırma

Terminal oturumunu, terminal emülatör penceresini kapatarak, kabuk isteminde `exit` komutunu girerek veya `Ctrl-d tuşlarına` basarak sonlandırabiliriz.

```
[me@linuxbox ~]$ exit
```

Perdenin Arkasındaki Konsol

Terminal emülatörü çalıştırmıyor olsak bile, grafik masaüstünün arkasında birkaç terminal oturumu çalışmaya devam eder. Çoğu Linux dağıtımında `Ctrl-Alt-F1` ile `Ctrl-Alt-F6` tuşlarına basarak *sanal terminaller* veya *sanal konsollar* olarak adlandırılan bu oturumlara erişebiliriz. Bir oturuma erişildiğinde, kullanıcı adımızı ve şifremizi girebileceğimiz bir oturum açma istemini gösterir. Bir sanal konsoldan diğerine geçmek için `Alt-F1` ile `Alt-F6 tuşlarına` basın. Çoğu sistemde `Alt-F7 tuşuna` basarak grafik masaüstüne geri dönebiliriz.

Özet

Bu bölüm, kabuğa giriş, komut satırına kısa bir bakış ve terminal oturumunu başlatma ve sonlandırma dersiyle Linux komut satırına yolculuğumuzun başlangıcını işaret etti. Ayrıca, bazı basit komutları nasıl vereceğimizi ve komut satırında küçük düzenlemeler yapmayı da gördük. O kadar da korkutucu değildi, değil mi?

Bir sonraki bölümde, birkaç komut daha öğrenecek ve Linux dosya sistemini keşfedeceğiz.

Daha Fazla Okuma

- Bourne Shell'in babası Steve Bourne hakkında daha fazla bilgi edinmek için şu Wikipedia makalesine bakın: http://en.wikipedia.org/wiki/Steve_Bourne
- Bu Wikipedia makalesi, `bash'in` orijinal yazarı Brian Fox hakkındadır: https://en.wikipedia.org/wiki/Brian_Fox_(computer_programmer)
- İşte bilgisayar biliminde kabuk kavramı hakkında bir makale: http://en.wikipedia.org/wiki/Shell_(computing)

2 – Gezinme

Öğrenmemiz gereken ilk şey (yazmayı öğrenmenin yanı sıra), Linux sistemimizdeki dosya sisteminde nasıl gezinileceğidir. Bu bölümde, aşağıdaki komutları tanıtacağız:

- `pwd` – Geçerli çalışma dizinini yazdırır.
- `cd` – Dizini değiştirir.
- `ls` – Dizin içeriğini listeler.

Dosya Sistemi Ağacını Anlamak

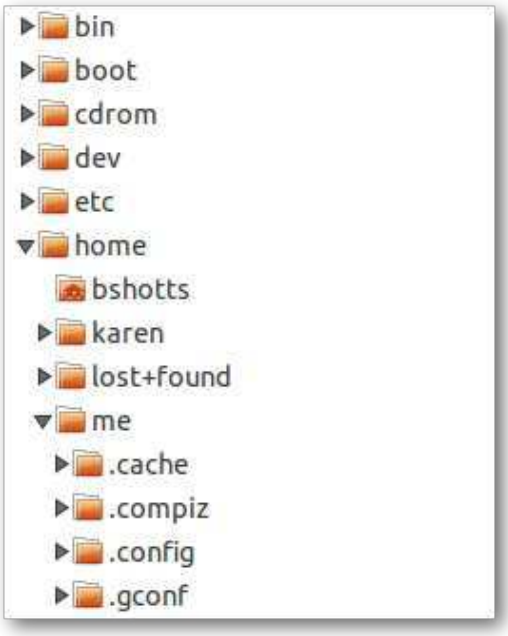
Windows gibi, Linux gibi Unix benzeri işletim sistemleri de dosyalarını *hiyerarşik dizin yapısı* olarak adlandırılan bir şekilde düzenler. Bu, dosyaların ve diğer dizinlerin bulunabileceği ağaç benzeri bir *dizin* yapısı (diğer sistemlerde bazen klasör olarak adlandırılır) içinde düzenlendikleri anlamına gelir. Dosya sistemindeki ilk dizin *kök dizin* olarak adlandırılır. Kök dizin, daha fazla dosya ve alt dizin içeren dosyalar ve alt dizinler içerir.

Her depolama aygıtı için ayrı bir dosya sistemi ağacına sahip olan Windows'un aksine, Linux gibi Unix benzeri sistemlerde, bilgisayara kaç tane sürücü veya depolama aygıtı bağlı olursa olsun, her zaman tek bir dosya sistemi ağacı vardır. Depolama aygıtları, sistemi korumakla sorumlu kişi (veya kişiler) olan *sistem yöneticisinin* isteğine göre ağacın çeşitli noktalarına bağlanır (veya daha doğrusu, *monte edilir*).

Geçerli Çalışma Dizini

Çoğumuz, Şekil 1'deki gibi dosya sistemi ağacını temsil eden grafiksel bir dosya yöneticisine aşinayızdır. Ağacın genellikle ters çevrilmiş olarak gösterildiğine, yani kökün en üstte ve çeşitli dalların aşağıya doğru uzandığına dikkat edin.

Ancak, komut satırında resim yoktur, bu nedenle dosya sistemi ağacında gezinmek için şunu yapmamız



gerekir

Şekil 1: Grafiksel dosya yöneticisi tarafından gösterilen dosya sistemi ağacı

Bunu farklı bir şekilde düşünün. Dosya sisteminin ters çevrilmiş bir ağaç şeklindeki bir labirent olduğunu ve bizim de ortasında durduğumuzu hayal edin. Herhangi bir anda, tek bir dizinin içindeyiz ve dizinde bulunan dosyaları, üstümüzdeki dizine (*üst dizin* olarak adlandırılır) giden yolu ve altımızdaki tüm alt dizinleri görebiliriz. Bulunduğumuz dizin, *geçerli çalışma dizini* olarak adlandırılır. Geçerli çalışma dizinini görüntülemek için `pwd` (çalışma dizinini yazdır) komutunu kullanırız.

```
[me@linuxbox ~]$ pwd
/home/me
```

Sistemimize ilk kez giriş yaptığımızda (veya bir terminal emülatörü oturumu başlattığımızda), geçerli çalışma dizini *ana dizine* ayarlanır. Her kullanıcı hesabına kendi ana dizini verilir ve bu, normal bir kullanıcının dosya yazmasına izin verilen tek yerdir.

Bir Dizinin İçeriğini Listeleme

Geçerli çalışma dizinindeki dosya ve dizinleri listelemek için `ls` komutunu kullanırız.

```
[me@linuxbox ~]$ ls
Masaüstü Belgeler Müzik Resimler Genel Şablonlar Videolar
```

Aslında, `ls` komutunu sadece geçerli çalışma dizininin değil, herhangi bir dizinin içeriğini listelemek için kullanabiliriz ve bu komutun yapabileceği başka birçok eğlenceli şey de vardır. Bir sonraki bölümde `ls komutunu` daha ayrıntılı olarak inceleyeceğiz.

Geçerli Çalışma Dizinini Değiştirme

Çalışma dizinimizi (ağaç şeklindeki labirentte bulunduğumuz yeri) değiştirmek için `cd` komutunu kullanırız. Bunu yapmak için, `cd` yazıp ardından boşluk ve istediğiniz çalışma dizininin *yol adını* yazın. Yol adı, istediğimiz dizine ulaşmak için ağacın dalları boyunca izlediğimiz yoldur. Yol adlarını iki farklı şekilde belirtebiliriz: *mutlak yol adları* veya *göreli yol adları*. Önce mutlak yol adlarına bakalım.

Mutlak Yol Adları

Mutlak yol adı, kök dizinle başlar ve istenen dizin veya dosyaya giden yol tamamlanana kadar ağaç dallarını takip eder. Örneğin, sistemimizde sistemimizin programlarının çoğunun kurulu olduğu bir dizin vardır. Dizinin yol adı `/usr/bin'dir`. Bu, kök dizinden (yol adındaki baştaki eğik çizgi ile gösterilir) `bin` adlı bir dizin içeren `usr` adlı bir dizin olduğu anlamına gelir.

```
[me@linuxbox ~]$ cd /usr/bin
[me@linuxbox bin]$ pwd
/usr/bin
[me@linuxbox bin]$ ls
...Çok sayıda dosyanın listesi ...
```

Şimdi, geçerli çalışma dizinini `/usr/bin` olarak değiştirdiğimizi ve bu dizinin dosyalarla dolu olduğunu görebiliriz. Kabuk isteminin nasıl değiştiğine dikkat ettiniz mi? Kolaylık olması açısından, genellikle çalışma dizininin adı otomatik olarak görüntülenmek üzere ayarlanır.

Göreceli Yol Adları

Mutlak yol adı kök dizinden başlayıp hedefine giden bir yol iken, göreceli yol adı çalışma dizininden başlar. Bunu yapmak için, dosya sistemi ağacındaki göreceli konumları temsil etmek üzere birkaç özel gösterim kullanır. Bu özel gösterimler "." (nokta) ve ".." (nokta nokta) şeklindedir.

`.` notasyonu çalışma dizinini, `..` notasyonu ise çalışma dizininin üst dizinini ifade eder. Örneğin, çalışma dizinini tekrar `/usr/ bin` olarak değiştirelim.

```
[me@linuxbox ~]$ cd /usr/bin
[me@linuxbox bin]$ pwd
/usr/bin
```

Şimdi, çalışma dizinini `/usr/bin'in` üst dizini olan `/usr` olarak değiştirmek istediğimizi varsayalım. Bunu iki farklı şekilde yapabiliriz: mutlak yol adı kullanarak

```
[me@linuxbox bin]$ cd /usr
[me@linuxbox usr]$ pwd
/usr
```

veya göreceli yol adını kullanarak:

```
[me@linuxbox bin]$ cd ..
[me@linuxbox usr]$ pwd
/usr
```

Aynı sonucu veren iki farklı yöntem. Hangisini kullanmalıyız? En az yazım gerektirenini!

Aynı şekilde, çalışma dizinini `/usr'den /usr/bin'e` iki farklı şekilde değiştirebiliriz: mutlak yol adını kullanarak

```
[me@linuxbox usr]$ cd /usr/bin
[me@linuxbox bin]$ pwd
/usr/bin
```

veya göreceli yol adı kullanarak:

```
[me@linuxbox usr]$ cd ./bin
[me@linuxbox bin]$ pwd
/usr/bin
```

Şimdi, burada belirtilmesi gereken önemli bir nokta var. Neredeyse tüm durumlarda, `./` girmeden de yapabiliriz. Bu, ima edilmiştir.

```
[me@linuxbox usr]$ cd bin
```

aynı şeyi yapar. Genel olarak, bir şeyin yol adını belirtmezsek, çalışma dizini varsayılır.

'da Bazı Yararlı Kısayollar

Tablo 2-1'de, geçerli çalışma dizinini hızlı bir şekilde değiştirebileceğiniz bazı yararlı yöntemler gösterilmektedir.

Tablo 2-1: `cd` Kısayolları

Kısayol	Sonuç
`cd`	Çalışma dizinini ana dizininize değiştirir.
`cd -`	Çalışma dizinini önceki çalışma dizinine değiştirir.
`cd ~`*`kullanıcı_adı`*	Çalışma dizinini *kullanıcı_adı kullanıcı* adının ana dizinine değiştirir. Örneğin, `cd ~bob` komutu, dizini *bob* kullanıcısının ana dizinine değiştirir.

Dosya Adları Hakkında Önemli Bilgiler

Linux sistemlerinde dosyalar, Windows gibi diğer sistemlere benzer şekilde adlandırılır, ancak bazı önemli farklılıklar vardır.

1. Nokta karakteriyle başlayan dosya adları gizlidir. Bu, `ls -a komutu` verilmedikçe `ls komutunun` bunları listelemeyeceği anlamına gelir. Hesabınız oluşturulduğunda, hesabınız için yapılandırma yapmak üzere ev dizinize birkaç gizli dosya yerleştirilmiştir. 11. bölümde, ortamınızı nasıl özelleştirebileceğinizi görmek için bu dosyalardan bazılarına daha yakından bakacağız. Ayrıca, bazı uygulamalar yapılandırma ve ayar dosyalarını ev dizinize gizli dosyalar olarak yerleştirir.

2. Linux'ta, Unix'te olduğu gibi, dosya adları ve komutlar büyük/küçük harfe duyarlıdır. "File1" ve "file1" dosya adları farklı dosyalara işaret eder.
3. Linux'ta, diğer bazı işletim sistemlerinde olduğu gibi "dosya uzantısı" kavramı yoktur. Dosyalara istediğiniz adı verebilirsiniz. Bir dosyanın içeriği veya amacı başka yollarla belirlenir. Unix benzeri işletim sistemleri, dosyaların içeriğini veya amacını belirlemek için dosya uzantıları kullanmasa da, birçok uygulama programı bunu yapar.
4. Linux, boşluk ve noktalama işaretleri içerebilen uzun dosya adlarını destekler, ancak oluşturduğunuz dosya adlarında noktalama işaretlerini nokta, tire ve alt çizgi ile sınırlayın. *En önemlisi, dosya adlarına boşluk eklemeyin.* Dosya adında kelimeler arasındaki boşlukları göstermek istiyorsanız, alt çizgi karakterleri kullanın. Daha sonra kendinize teşekkür edeceksiniz.

Özet

Bu bölümde, kabuğun sistemin dizin yapısını nasıl işlediği açıklanmıştır. Mutlak ve göreli yol adları ile bu yapı içinde hareket etmek için kullandığımız temel komutlar hakkında bilgi edindik. Bir sonraki bölümde, bu bilgileri kullanarak modern bir Linux sistemini gezeceğiz.

3 – Sistemi Keşfetmek

Artık dosya sisteminde nasıl hareket edeceğimizi bildiğimize göre, Linux sistemimizi rehberli bir turla gezme zamanı geldi. Ancak başlamadan önce, bu turda işimize yarayacak birkaç komut daha öğreneceğiz.

- `ls` – Dizin içeriğini listeler.
- `file` – Dosya türünü belirler.
- `less` – Dosya içeriğini görüntüler.

`ls` ile Daha Fazla Eğlence

`ls` komutu muhtemelen en çok kullanılan Linux komutudur ve bunun iyi bir nedeni vardır. Bu komutla, dizin içeriğini görebilir ve çeşitli önemli dosya ve dizin özniteliklerini belirleyebiliriz. Gördüğümüz gibi, `ls komutunu` girerek geçerli çalışma dizininde bulunan dosya ve alt dizinlerin listesini alabiliriz.

```
[me@linuxbox ~]$ ls
Masaüstü Belgeler Müzik Resimler Genel Şablonlar Videolar
```

Geçerli çalışma dizininin yanı sıra, listelenecek bir dizin de şu şekilde belirtebiliriz:

```
me@linuxbox ~]$ ls /usr
bin oyunlar dahil lib yerel sbin paylaş src
```

Birden fazla dizin de belirtebiliriz. Aşağıdaki örnekte, kullanıcının ana dizini (sembolü "~" karakteridir) ve `/usr` dizinini listeliyoruz:

```
[me@linuxbox ~]$ ls ~ /usr
/home/me:
Masaüstü Belgeler Müzik Resimler Genel Şablonlar Videolar
```

```
/usr:
bin oyunlar dahil lib yerel sbin paylaş src
```

Daha fazla ayrıntı göstermek için çıktının formatını da değiştirebiliriz.

```
[me@linuxbox  ~]$ ls -l
toplam 56
drwxrwxr-x 2  me me 4096  2025-10-26  17:20 Masaüstü
drwxrwxr-x 2   ben ben   2025-10-26  17:20 Belgeler
                 4096
drwxrwxr-x 2   ben ben   2025-10-26  17:20 Müzik
                 4096
drwxrwxr-x 2   ben ben   2025-10-26  17:20 Resimler
                 4096
drwxrwxr-x 2   ben ben   2025-10-26  17:20 Genel
                 4096
drwxrwxr-x 2   ben ben   26.10.2025  17:20 Şablonlar
                 4096
drwxrwxr-x 2   ben ben   2025-10-26  17:20 Videolar
                 4096
```

Komuta "-l" ekleyerek çıktıyı uzun biçime değiştirdik.

Seçenekler ve Argümanları

Bu, çoğu komutun nasıl çalıştığına dair önemli bir noktaya bizi getiriyor. Komutların ardından genellikle davranışlarını değiştiren bir veya daha fazla *seçenek* ve ayrıca komutun etki ettiği öğeler olan bir veya daha fazla *argüman* gelir. Bu nedenle çoğu komut şu şekilde görünür:

```
komut -seçenekler argümanlar
```

Çoğu komut, öncesinde tire bulunan tek bir karakterden oluşan seçenekler kullanır, örneğin `-l`. Ancak GNU Projesi'nden gelenler de dahil olmak üzere birçok komut, öncesinde iki tire bulunan bir kelimeden oluşan *uzun seçenekleri* de destekler. Ayrıca, birçok komut birden fazla kısa seçeneğin bir araya getirilmesine izin verir. Aşağıdaki örnekte, `ls` komutuna iki seçenek verilmiştir: uzun biçimli çıktı üretmek için `l` seçeneği ve sonucu dosyanın değiştirilme zamanına göre sıralamak için `t` seçeneği.

```
[me@linuxbox ~]$ ls -lt
```

Sıralama sırasını tersine çevirmek için uzun seçenek “--reverse” ekleyeceğiz.

```
[me@linuxbox ~]$ ls -lt --reverse
```

Linux'ta dosya adları gibi komut seçeneklerinin de büyük/küçük harfe duyarlı olduğunu unutmayın.

`ls` komutunun birçok olası seçeneği vardır. Tablo 3-1 en yaygın olanları açıklamaktadır.

Tablo 3- 1: Yaygın `ls` Seçenekleri

Seçenek	Uzun seçenek	Açıklama
`-a`	`--all`	Normalde listelenmeyen (yani gizli olan) nokta ile başlayan adlara sahip dosyalar da dahil olmak üzere tüm dosyaları listeler.
`-`	`--almost-all`	`-a` seçeneği gibidir, ancak `.` (geçerli dizin) ve `..` (üst dizin) listelenmez.
`-d`	`--directory`	Normalde, bir dizin belirtilirse, `ls` dizin kendisini değil, dizinin içeriğini listeler. Dizin içeriği yerine dizinle ilgili ayrıntıları görmek için bu seçeneği `-l` seçeneği ile birlikte kullanın.
`-F`	`--classify`	Bu seçenek, listelenen her adın sonuna bir gösterge karakteri ekler; örneğin, ad bir dizin ise eğik çizgi (/) ekler.
`-h`	`--insan tarafından okunabilir`	Uzun biçimli listelerde, dosya boyutlarını bayt cinsinden değil, insan tarafından okunabilir biçimde görüntüleyin.
`-l`		Sonuçları uzun formatta görüntüleyin.
`-r`	`--reverse`	Sonuçları ters sırayla görüntüleyin. Normalde, `ls` sonuçları alfabetik sırayla görüntüler.
`-S`		Sonuçları dosya boyutuna göre sıralayın.
`-t`		Değişiklik zamanına göre sıralayın.

Uzun Biçim ına Daha Yakından Bakış

Daha önce gördüğümüz gibi, `-l` seçeneği `ls` komutunun sonuçlarını uzun formatta görüntülemesini sağlar. Bu format çok sayıda yararlı bilgi içerir. İşte

-rw-r--r--

```
-rw-r--r--  1 root  root  3576296  2017-04-03  11:05 Ubuntu deneyimini
                                                     yaşayın.ogg
-rw-r--r--  1 root  root  1186219  2017-04-03  11:05 kubuntu-broşür.png
-rw-r--r--  1 root  root    47584  03.04.2017  11:05 logo-Edubuntu.png
-rw-r--r--  1 root  root    44355  2017-04-03  11:05 logo-Kubuntu.png
-rw-r--r--  1 root  root    34391  2017-04-03  11:05 logo-Ubuntu.png
-rw-r--r--  1 root  root    32059  2017-04-03  11:05 oo-cd-kapak.odf
-rw-r--r--  1 root  root   159744  2017-04-03  11:05 oo-türevleri.doc
-rw-r--r--  1 root  root    27837  2017-04-03  11:05 oo-maxwell.odt
-rw-r--r--  1 root  root    98816  2017-04-03  11:05 oo-trig.xls
-rw-r--r--  1 root  root   453764  2017-04-03  11:05 oo-welcome.odt
-rw-r--r--  1 root  root   358374  2017-04-03  11:05 ubuntu Sax.ogg
```

Tablo 3-2, dosyalardan birinin farklı alanlarını ve anlamlarını göstermektedir.

Tablo 3-2: `ls` için Uzun Liste Alanları

Alan	Anlam
`-rw-r--r--`	Dosyaya erişim hakları. İlk karakter dosya türünü belirtir dosya türünü belirtir. Farklı türler arasında, baştaki tire normal bir dosyayı, "d" ise bir dizini belirtir. Sonraki üç karakter dosyanın sahibinin erişim haklarını, sonraki üç karakter dosyanın grubunun üyelerinin erişim haklarını ve son üç karakter ise diğer herkesin erişim haklarını belirtir. Bölüm 9'da bunun tam anlamı daha ayrıntılı olarak ele alınmaktadır.
`1`	Dosyanın sabit bağlantı sayısı. Bu bölümün ilerleyen kısımlarında yer alan "Sembolik Bağlantılar" ve "Sabit Bağlantılar" bölümlerine bakın.
`root`	Dosyanın sahibinin kullanıcı adı.
`root`	Dosyanın sahibi olan grubun adı.
`32059`	Dosyanın bayt cinsinden boyutu.
`2017-04-03`	Dosyanın son değiştirilme tarihi ve saati.
`oo-cd-kapak.odf`	Dosyanın adı.

`file` ile Dosya Türünü Belirleme

Sistemi keşfederken, dosyalarda ne tür veriler bulunduğunu bilmek faydalı olacaktır.

bunu yapmak için, dosyanın türünü belirlemek üzere `file` komutunu kullanacağız. Daha önce de belirttiğimiz gibi, Linux'ta dosya adlarının dosyanın içeriğini yansıtması gerekmez. `picture.jpg` adlı bir dosyanın normalde JPEG sıkıştırılmış bir görüntü içermesi beklenir, ancak Linux'ta bu zorunlu değildir. `file` komutunu şu şekilde çağırırız:

```
file dosya adı
```

Çalıştırıldığında, `file` komutu dosyanın içeriğinin kısa bir açıklamasını yazdırır. Örneğin:

```
[me@linuxbox ~]$ file picture.jpg
picture.jpg: JPEG görüntü verisi, JFIF standardı 1.01
```

Birçok dosya türü vardır. Aslında, Linux gibi Unix benzeri işletim sistemlerinin temel fikirlerinden biri "her şey bir dosyadır"dır. Derslerimize devam ederken, bu ifadenin ne kadar doğru olduğunu göreceğiz.

Sistemimizdeki dosyaların çoğu tanıdıktır (örneğin, MP3 ve JPEG), ancak biraz daha az bilinen birçok tür ve oldukça garip olan birkaç tür de vardır.

`less` ile Dosya İçeriğini Görüntüleme

`less` komutu, metin dosyalarını görüntülemek için kullanılan bir programdır. Linux sistemimizde, birçok dosya insan tarafından okunabilir metinler içerir. `less` programı, bu dosyaları incelemek için kullanışlı bir yol sunar.

"Metin" Nedir?

Bilgisayarda bilgileri temsil etmenin birçok yolu vardır. Tüm yöntemler, bilgiler ile bunları temsil etmek için kullanılacak bazı sayılar arasındaki ilişkiyi tanımlamayı içerir. Sonuçta bilgisayarlar sadece sayıları anlar ve tüm veriler sayısal temsil biçimine dönüştürülür.

Bu temsil sistemlerinden bazıları karmaşıktır (örneğin sıkıştırılmış video dosyaları), bazıları ise basittir. En eski ve en basit olanlardan biri *ASCII metinidir*. ASCII (telaffuz: "as-key"), American Standard Code for Information Interchange (Amerikan Bilgi Değişim Standart Kodu) ifadesinin kısaltmasıdır. Bu, klavye karakterlerini sayılara eşlemek için ilk olarak teletip makinelerde kullanılan basit bir kodlama şemasıdır.

Metin, karakterleri sayılara basit bir bire bir eşlemeyle dönüştürür. Kompakt bir sistemdir: Elli karakterlik metin, 50 baytlık veriye dönüştürülür. Metnin yalnızca karakterleri sayılara basit bir eşlemeyle dönüştürdüğünü anlamak önemlidir. Microsoft Word veya LibreOffice Writer ile oluşturulan kelime işlemci belgeleriyle aynı şey değildir. Bu dosyalar, basit ASCII metinlerinin aksine, yapısını ve biçimlendirmesini tanımlamak için kullanılan birçok metin dışı öğe içerir. Düz ASCII metin dosyaları yalnızca karakterlerin kendisini ve sekme, satır başı ve satır sonu gibi birkaç temel kontrol kodunu içerir.

Linux sisteminde birçok dosya metin biçiminde saklanır ve metin dosyalarıyla çalışan birçok Linux aracı vardır. Windows bile bu biçimin önemini kabul eder. Tanınmış NOTEPAD.EXE programı, düz ASCII metin dosyaları için bir düzenleyicidir.

Metin dosyalarını neden incelemek isteyelim? Çünkü sistem ayarlarını içeren dosyaların çoğu (*yapılandırma dosyaları* olarak adlandırılır) bu biçiminde saklanır ve bunları okuyabilmek bize sistemin nasıl çalıştığı hakkında fikir verir. Ayrıca, sistemin kullandığı bazı gerçek programlar (*komut dosyaları* olarak adlandırılır) da bu biçimde saklanır. Sonraki bölümlerde, sistem ayarlarını değiştirmek ve kendi komut dosyalarımızı yazmak için metin dosyalarını nasıl düzenleyeceğimizi öğreneceğiz, ancak şimdilik sadece içeriklerine bakacağız.

`less` komutu şu şekilde kullanılır:

```
less dosya adı
```

Başlatıldığında, `less` programı bir metin dosyasında ileri ve geri kaydırma yapmamızı sağlar. Örneğin, sistemdeki tüm kullanıcı hesaplarını tanımlayan dosyayı incelemek için aşağıdaki komutu girin:

```
[me@linuxbox ~]$ less /etc/passwd
```

`less` programı başlatıldığında, dosyanın içeriğini görüntüleyebiliriz. Dosya bir sayfadan uzunsa, yukarı ve aşağı kaydırabiliriz. `less programından` çıkmak için `q tuşuna` basın.

Tablo 3-3, `less` tarafından kullanılan en yaygın klavye komutlarını listeler.

Tablo 3-3: `less` Komutları

Komut	Eylem
Page Up veya `b`	Bir sayfa geri kaydır
Page Down veya boşluk	Bir sayfa ileri kaydır
Yukarı ok	Bir satır yukarı kaydır
Aşağı ok	Bir satır aşağı kaydır
`G`	Metin dosyasının sonuna git
`1G` veya `g`	Metin dosyasının başına git
/karakterler	*Karakterlerin* bir sonraki geçtiği yere doğru ileriye doğru arama yap
`n`	Önceki aramanın bir sonraki örneğini ara
`h`	Yardım ekranını görüntüle
`q`	Çıkış `less`

Az Daha Fazladır

`less` programı, daha önceki bir Unix programı olan `more'un` geliştirilmiş bir versiyonu olarak tasarlanmıştır. `less` adı, modernist mimar ve tasarımcıların sloganı olan "az daha çoktur" ifadesinden esinlenerek verilmiştir.

`less`, uzun metin belgelerini sayfa sayfa kolayca görüntülemeyi sağlayan, *pager* adı verilen programlar sınıfına girer. `more` programı sadece ileriye doğru sayfa çevirme özelliğine sahipken, `less` programı hem ileriye hem de geriye doğru sayfa çevirme özelliğine sahiptir ve birçok başka özelliği de vardır.

Rehberli Tur

Linux sistemindeki dosya sistemi düzeni, diğer Unix benzeri sistemlerdekine çok benzer. Tasarım, *Linux Dosya Sistemi Hiyerarşi Standardı* adlı yayınlanmış bir standartta belirtilmiştir. Tüm Linux dağıtımları bu standarda tam olarak uymaz, ancak çoğu oldukça yakındır.

Ardından, Linux sistemimizin nasıl çalıştığını görmek için dosya sisteminde dolaşacağız. Bu, gezinme becerilerimizi pratik etme fırsatı verecektir. Keşfedeceğimiz şeylerden biri, ilginç dosyaların çoğunun insan tarafından okunabilir düz metin biçiminde olduğudur. Turumuza devam ederken

turumuza devam ederken, şunu deneyin:

1. Belirli bir dizine `cd` ile girin
2. `ls -l` ile dizin içeriğini listeleyin
3. İlginç bir dosya görürseniz, `file` komutuyla içeriğini belirleyin
4. Metin gibi görünüyorsa, `daha az` görüntülemeyi deneyin
5. Yanlışlıkla metin olmayan bir dosyayı görüntülemeyi denersek ve terminal penceresi karışırsa, `reset` komutunu girerek düzeltebiliriz.

Kopyala-yapıştır hilesini unutmayın! Fare kullanıyorsanız, dosya adını çift tıklayarak kopyalayabilir ve orta tuşla komutlara yapıştırabilirsiniz.

Gezinirken, bir şeylere bakmaktan çekinmeyin. Normal kullanıcıların çoğu, sistemde değişiklik yapamaz. Bu, sistem yöneticisinin işidir! Bir komut bir şeyden şikayet ederse, başka bir şeye geçin. Biraz zaman ayırıp etrafa bakın. Sistemi keşfetmek bizim hakkımız. Unutmayın, Linux'ta sır yoktur!

Tablo 3-4, keşfedebileceğimiz dizinlerden sadece birkaçını listelemektedir. Linux dağıtımımıza bağlı olarak bazı küçük farklılıklar olabilir. Etrafı incelemekten ve daha fazlasını denemekten çekinmeyin!

Tablo 3-4: Linux Sistemlerindeki Dizinler

Dizin	Yorumlar
`/`	Kök dizin. Her şeyin başladığı yer.
`/bin`	Sistemin önyükleme yapması ve çalışması için gerekli olan ikili dosyalar (programlar) içerir. Modern Linux dağıtımlarında `/bin` yerine `/usr/bin` kullanıldığına dikkat edin.
`/boot`	Linux çekirdeğini, ilk RAM disk görüntüsünü (başlatma sırasında gerekli sürücüler için) ve önyükleyiciyi içerir. İlginç dosyalar şunlardır: • `/boot/grub/grub.cfg` veya `menu.lst`, önyükleyiciyi yapılandırmak için kullanılır. • `/boot/vmlinuz` (veya benzeri), Linux çekirdeği
`/dev`	Bu, *aygıt düğümlerini* içeren özel bir dizindir. "Her şey bir dosyadır" ifadesi aygıtlar için de geçerlidir. Çekirdek, anladığı tüm aygıtların listesini burada tutar.

Dizin	Yorum
/etc	/etc dizini, sistem genelindeki tüm yapılandırma dosyalarını içerir. Bu dizindeki her şey okunabilir metin olmalıdır. /etc içindeki her şey ilginç olsa da, işte en sevilenler: • /etc/crontab, cron programını kullanan sistemlerde otomatik işlerin ne zaman çalışacağını tanımlar. • /etc/fstab, depolama aygıtları ve bunlarla ilişkili bağlama noktalarının bir tablosudur. • /etc/passwd, kullanıcı hesaplarının bir listesidir.
/home	Normal yapılandırmalarda, her kullanıcıya /home içinde bir dizin verilir. Sıradan kullanıcılar yalnızca kendi ana dizinlerine dosya yazabilirler. Bu sınırlama, sistemi hatalı kullanıcı faaliyetlerinden korur.
/lib	Çekirdek sistem programları tarafından kullanılan paylaşılan kütüphane dosyalarını içerir. Bunlar, Windows'taki dinamik bağlantı kütüphanelerine (DLL'ler) benzer. Bu dizin, modern dağıtımlarda /usr/lib lehine kullanımdan kaldırılmıştır.
/lost+found	Ext4 gibi geleneksel Linux dosya sistemini kullanan her biçimlendirilmiş bölüm veya aygıtta bu dizin bulunur. Dosya sistemi bozulması durumunda kısmi kurtarma için kullanılır. Sistemimizde gerçekten kötü bir şey olmadıkça, bu dizin boş kalır.
/media	Modern Linux sistemlerinde, /media dizini, takıldığında otomatik olarak bağlanan USB sürücüler, CD-ROM'lar vb. gibi çıkarılabilir ortamlar için bağlama noktalarını içerir.
/mnt	Eski Linux sistemlerinde, /mnt dizini manuel olarak bağlanan aygıtların bağlanma noktalarını içerir.
/opt	/opt dizini, "isteğe bağlı" yazılımları yüklemek için kullanılır. Bu dizin, esas olarak sisteme yüklenebilecek ticari yazılım ürünlerini barındırmak için kullanılır.

Dizin	Yorum
/proc	/proc dizini özeldir. Sabit diskte depolanan dosyalar anlamında gerçek bir dosya sistemi değildir. Daha çok, Linux çekirdeği tarafından yönetilen sanal bir dosya sistemidir. İçerdiği "dosyalar" çekirdeğe açılan gözetleme delikleridir. Dosyalar okunabilir ve çekirdeğin bilgisayarı nasıl gördüğüne dair bir fikir verir. Bu dizini taramak, bilgisayarın donanımı hakkında birçok ayrıntıyı ortaya çıkarabilir.
/root	Bu, root hesabının ana dizini.
/run	Bu, geleneksel /tmp dizininin modern bir alternatifidir. /tmp'den farklı olarak, /run dizini, içeriğini fiziksel bir diskte değil bellekte depolayan *tempfs* dosya sistemi türü kullanılarak bağlanır.
/sbin	Bu dizin "sistem" ikili dosyalarını içerir. Bunlar, genellikle süper kullanıcıya ayrılmış olan hayati sistem görevlerini gerçekleştiren programlardır. Modern Linux dağıtımlarının /sbin yerine /usr/sbin
/sys	/sys dizini, çekirdek tarafından algılanan aygıtlar hakkında bilgiler içerir. Bu, /dev dizininin içeriğine çok benzer, ancak gerçek donanım adresleri gibi daha ayrıntılı bilgileri içerir.
/tmp	/tmp dizini, çeşitli programlar tarafından oluşturulan geçici, geçici dosyaları depolamak için tasarlanmıştır. Bazı dağıtımlar, sistem her yeniden başlatıldığında bu dizini boşaltır.
/usr	/usr dizin ağacı, Linux sistemlerinde muhtemelen en büyük dizindir. Normal kullanıcılar tarafından kullanılan tüm programları ve destek dosyalarını içerir.
/usr/bin	/usr/bin, Linux dağıtımı tarafından yüklenen yürütülebilir programları içerir. Bu dizinde binlerce program bulunması olağandışı bir durum değildir.
/usr/lib	/usr/bin içindeki programlar için paylaşılan kütüphaneler.

Dizin	Yorum
/usr/local	/usr/local dizini, dağıtımda bulunmayan ancak sistem genelinde kullanılmak üzere tasarlanmış programların kurulduğu yerdir. Kaynak kodundan derlenen programlar normalde /usr/local/bin dizinine kurulur. Yeni kurulmuş bir Linux sisteminde bu dizin mevcuttur, ancak sistem yöneticisi içine bir şey koyana kadar boş kalır.
/usr/sbin	Daha fazla sistem yönetimi programı içerir.
/usr/share	/usr/share, /usr/bin içindeki programlar tarafından kullanılan tüm paylaşılan verileri içerir. Buna varsayılan yapılandırma dosyaları, simgeler, ekran arka planları, ses dosyaları vb. dahildir.
/usr/share/doc	Sisteme yüklenen çoğu paket, bir tür dokümantasyon içerir. /usr/share/doc içinde, paketlere göre düzenlenmiş dokümantasyon dosyaları bulunur.
/var	/tmp ve /home hariç, şimdiye kadar incelediğimiz dizinler nispeten statiktir; yani içerikleri değişmez. /var dizin ağacı, değişmesi muhtemel verilerin depolandığı yerdir. Çeşitli veritabanları, spool dosyaları, kullanıcı postaları vb. burada bulunur.
/var/log	/var/log, çeşitli sistem etkinliklerinin kayıtlarını içeren *günlük dosyalarını* içerir. Bunlar önemlidir ve zaman zaman izlenmelidir. En yararlı olanları /var/log/messages ve/veya /var/log/syslog'dur, ancak bunlar tüm sistemlerde mevcut değildir. Güvenlik nedenleriyle, bazı sistemlerde yalnızca süper kullanıcıların günlük dosyalarını görüntüleyebildiğini unutmayın.
~/.config ve ~/.local	Bu iki dizin, her masaüstü kullanıcısının ana dizininde bulunur. Masaüstü uygulamaları için kullanıcıya özgü yapılandırma ve program durumu verilerini depolamak için kullanılırlar.

Sembolik Bağlantılar

Etrafımıza baktığımızda, muhtemelen /usr/lib gibi bir dizinde aşağıdaki gibi bir giriş içeren bir dizin listesi göreceğiz:

```
lrwxrwxrwx 1 root root    11 2025-08-11 07:34 libc.so.6 -> libc-2.6.so
```

Listenin ilk harfinin "l" olduğunu ve girdinin iki dosya adına sahip gibi göründüğünü fark ettiniz mi? Bu, *sembolik bağlantı* (*yumuşak bağlantı* veya *sym-link* olarak da bilinir) adı verilen özel bir dosya türüdür. Çoğu Unix benzeri sistemde, bir dosyanın birden fazla adla referanslanması mümkündür. Bunun değeri açık olmayabilir, ancak gerçekten kullanışlı bir özelliktir.

Şu senaryoyu düşünün: Bir program, "foo" adlı bir dosyada bulunan bir tür paylaşılan kaynağın kullanımını gerektirir, ancak "foo" sık sık sürüm değişikliklerine uğrar. Yönetici veya diğer ilgili tarafların "foo"nun hangi sürümünün yüklü olduğunu görebilmesi için dosya adına sürüm numarasını eklemek iyi olur. Bu bir sorun yaratır. Paylaşılan kaynağın adını değiştirirsek, onu kullanabilecek her programı takip etmemiz ve kaynağın yeni bir sürümü yüklendiğinde her seferinde yeni kaynak adını aramak için programı değiştirmemiz gerekir. Bu hiç de eğlenceli bir iş gibi görünmüyor.

İşte burada sembolik bağlantılar imdadımıza yetişir. "foo"nun 2.6 sürümünü, dosya adı `foo-2.6` olarak yüklediğimizi ve ardından `foo-2.6'ya` işaret eden, basitçe `foo` adında bir sembolik bağlantı oluşturduğumuzu varsayalım. Bu, bir program `foo` dosyasını açtığında, aslında `foo-2.6` dosyasını açtığı anlamına gelir. Artık herkes mutlu. "foo"ya bağlı programlar onu bulabilir ve biz de hangi sürümün yüklü olduğunu görebiliriz. `foo-2.7 sürümüne` yükseltme zamanı geldiğinde, dosyayı sistemimize ekler, `foo` sembolik bağlantısını siler ve yeni sürüme işaret eden yeni bir sembolik bağlantı oluştururuz. Bu, sürüm yükseltme sorununu çözmekle kalmaz, aynı zamanda her iki sürümü de makinemizde tutmamızı sağlar. `foo-2.7`'de bir hata olduğunu (lanet olsun o geliştiricilere!) ve eski sürüme geri dönmemiz gerektiğini düşünün. Yine, yeni sürüme işaret eden sembolik bağlantıyı siler ve eski sürüme işaret eden yeni bir sembolik bağlantı oluştururuz.

Bu bölümün başında yer alan dizin listesi (Fedora sisteminin /usr/lib dizininden alınmıştır) `libc-2.6.so` adlı paylaşımlı kütüphane dosyasına işaret eden `libc.so.6` adlı bir sembolik bağlantı göstermektedir. Bu, `libc.so.6` dosyasını arayan programların aslında `libc-2.6.so` dosyasını bulacağı anlamına gelir. Sembolik bağlantıların nasıl oluşturulacağını bir sonraki bölümde öğreneceğiz.

Sabit Bağlantılar

Bağlantılar konusuna değinmişken, *sabit bağlantı* adı verilen ikinci bir bağlantı türü olduğunu da belirtmeliyiz. Sabit bağlantılar da dosyaların birden fazla ada sahip olmasını sağlar, ancak bunu farklı bir şekilde yapar. Sembolik ve sabit bağlantılar arasındaki farkları bir sonraki bölümde daha ayrıntılı olarak ele alacağız.

Özet

Turumuzu geride bıraktığımızda, sistemimiz hakkında çok şey öğrendik. Çeşitli dosyaları

ve dizinleri ve bunların içeriklerini gördük. Bundan çıkarılması gereken bir sonuç, sistemin ne kadar açık olduğudur. Linux'ta birçok önemli dosya, insanlar tarafından okunabilir düz metin biçimindedir. Birçok özel sistemden farklı olarak, Linux her şeyi inceleme ve araştırma için kullanılabilir hale getirir.

Daha Fazla Okuma

- *Linux Dosya Sistemi Hiyerarşi Standardının* tam sürümü burada bulunabilir: https://refspecs.linuxfoundation.org/fhs.shtml
- Unix ve Unix benzeri sistemlerin dizin yapısı hakkında bir makale: http://en.wikipedia.org/wiki/Unix_directory_structure
- ASCII metin formatının ayrıntılı açıklaması: http://en.wikipedia.org/wiki/ ASCII

4 – Dosya ve Dizinleri İşleme

Artık gerçek işlere hazırız! Bu bölümde aşağıdaki komutlar tanıtılacaktır:

- `cp` – Dosya ve dizinleri kopyalar.
- `mv` – Dosya ve dizinleri taşıma/yeniden adlandırma.
- `mkdir` – Dizinleri oluşturmak.
- `rm` – Dosyaları ve dizinleri kaldırır.
- `ln` – Sabit ve sembolik bağlantılar oluşturun.

Bu beş komut, en sık kullanılan Linux komutları arasındadır. Hem dosyaları hem de dizinleri işlemek için kullanılırlar.

Açıkçası, bu komutlarla gerçekleştirilen bazı görevler grafiksel bir dosya yöneticisiyle daha kolay yapılabilir. Dosya yöneticisiyle, bir dosyayı bir dizinden diğerine sürükleyip bırakabilir, dosyaları kesip yapıştırabilir, dosyaları silebilir vb. öyleyse neden bu eski komut satırı programlarını kullanmalıyız?

Cevap, güç ve esnekliktir. Grafiksel bir dosya yöneticisiyle basit dosya işlemlerini gerçekleştirmek kolay olsa da, karmaşık görevler komut satırı programlarıyla daha kolay olabilir. Örneğin, bir dizindeki tüm HTML dosyalarını başka bir dizine kopyalayabiliriz, ancak hedef dizinde bulunmayan veya hedef dizindeki sürümlerden daha yeni olan dosyaları kopyalayabiliriz. Bu, bir dosya yöneticisiyle oldukça zordur, ancak komut satırıyla oldukça kolaydır.

```
cp -u *.html hedef
```

Joker karakterler

Komutlarımızı kullanmaya başlamadan önce, bu komutları bu kadar güçlü kılan bir kabuk özelliğinden bahsetmemiz gerekir. Kabuk, dosya adlarını çok sık kullandığından, dosya adı gruplarını hızlı bir şekilde belirtmemize yardımcı olan özel karakterler sağlar. Bu özel karakterler

joker karakterler olarak adlandırılır. Joker karakterleri (*globbing* olarak da bilinir) kullanarak, karakter kalıplarına göre dosya adlarını seçebiliriz. Tablo 4-1, joker karakterleri ve bunların seçtikleri öğeleri listeler.

Tablo 4-1: Joker karakterler

Joker Karakter	Anlam
`*`	Hiçbir karakter dahil olmak üzere tüm karakterlerle eşleşir
`?`	Herhangi bir tek karakterle eşleşir
`[`*`karakterler`*`]`	*Karakterler* kümesinin üyesi olan herhangi bir karakterle eşleşir
`[!`*`karakterler`*`]` veya `[^`*`karakterler`*`]`	Karakter kümesinin üyesi olmayan herhangi bir karakterle eşleşir *karakterler*
`[[:`*`sınıf`*`:]]`	Belirtilen *sınıfa* üye olan herhangi bir karakterle eşleşir

Tablo 4-2, en sık kullanılan karakter sınıflarını listeler.

Tablo 4-2: Sık Kullanılan Karakter Sınıfları

Karakter sınıfı	Anlam
`[:alnum:]`	Herhangi bir alfasayısal karakterle
eşleşir `[:alpha:]`	Herhangi bir alfabetik karakterle eşleşir
`[:digit:]`	Herhangi bir rakamla eşleşir
`[:lower:]`	Herhangi bir küçük harfle eşleşir
`[:upper:]`	Herhangi bir büyük harfle eşleşir

Joker karakterleri kullanarak dosya adları için karmaşık seçim kriterleri oluşturmak mümkündür. Tablo 4-3, bazı desen örnekleri ve bunların eşleştiği öğeleri göstermektedir.

Tablo 4-3: Joker Karakter Örnekleri

Desen	Eşleşmeler
`*`	Tüm dosyalar
`g*`	"g" ile başlayan tüm dosyalar
`b*.txt`	"b" ile başlayan, ardından herhangi bir karakter gelen ve ".txt" ile biten tüm dosyalar

`Veri???`	"Data" ile başlayan ve tam olarak üç karakterle devam eden tüm dosyalar
`[abc]*`	"a", "b" veya "c" ile başlayan herhangi bir dosya
`YEDEKLEME.[0-9][0-9][0-9]`	"BACKUP." ile başlayan ve tam olarak üç rakamla devam eden herhangi bir dosya
`[[:upper:]]*`	Büyük harfle başlayan herhangi bir dosya
`[![:digit:]]*`	Rakamla başlamayan herhangi bir dosya
`*[[:lower:]123]`	Küçük harfle veya "1", "2" veya "3" rakamlarıyla biten herhangi bir dosya

Joker karakterler, dosya adlarını argüman olarak kabul eden tüm komutlarla kullanılabilir, ancak bu konu hakkında daha ayrıntılı bilgiyi 7. bölümde vereceğiz.

Karakter Aralıkları

Başka bir Unix benzeri ortamdan geliyorsanız veya bu konuyla ilgili başka kitaplar okuduysanız, `[A-Z]` ve `[a-z]` karakter aralığı notasyonlarıyla karşılaşmış olabilirsiniz. Bunlar geleneksel Unix notasyonlarıdır ve eski Linux sürümlerinde de çalışırdı. Hala çalışabilirler, ancak doğru şekilde yapılandırılmadıkları takdirde beklenen sonuçları vermeyecekleri için dikkatli olmalısınız. Şimdilik, bunları kullanmaktan kaçınmalı ve bunun yerine karakter sınıflarını kullanmalısınız.

Nokta Dosyaları

-a seçeneğini kullanarak `ls` komutuyla ana dizini incelersek, adları nokta ile başlayan bir dizi dosya ve dizin olduğunu fark ederiz. Daha önce de belirttiğimiz gibi, bu dosyalar gizlidir. Bu, dosyanın özel bir özelliği değildir; sadece `-a` veya `-A` seçenekleri kullanılmadıkça dosyanın `ls` komutunun çıktısında görünmeyeceği anlamına gelir. Bu gizli özellik, joker karakterler için de geçerlidir. Gizli dosyalar, .* gibi bir joker karakter kalıbı kullanmadığımız sürece görünmez. Ancak, bazı durumlarda (kabuğumuzun yapılandırmasına bağlı olarak), sonuçlarda hem `.` (geçerli dizin) hem de `..` (geçerli dizinin üst dizini) görebiliriz. Bunları hariç tutmak için `.[!.]*` veya `.??*` gibi kalıplar kullanabiliriz.

Joker Karakterler GUI'de de Çalışır

Joker karakterler, komut satırında sıkça kullanıldıkları için değil, bazı grafik dosya yöneticileri tarafından da desteklendiği için özellikle değerlidir.

- Nautilus'ta (GNOME için dosya yöneticisi), Ctrl-s tuşlarına basıp joker karakterler içeren bir dosya seçim deseni girerek dosyaları seçebilirsiniz; bu durumda, o anda görüntülenen dizindeki dosyalar seçilecektir.
- Dolphin'de (KDE için dosya yöneticisi) joker karakter seçimi yapmak için kullanabileceğiniz "Filtre..." adlı bir seçenek vardır. Örneğin, /usr/bin dizininde küçük harf "u" ile başlayan tüm dosyaları görmek istiyorsanız, filtre istemine "/usr/bin/u*" yazın ve sonuç görüntülenir.

Komut satırı arayüzünde bulunan birçok fikir, grafik arayüze de yansıtılmıştır. Bu, Linux masaüstünü bu kadar güçlü kılan birçok şeyden biridir.

`mkdir` – Dizin Oluşturma

`mkdir` komutu, dizinler oluşturmak için kullanılır. Şu şekilde çalışır:

```
mkdir dizini...
```

Notasyonla ilgili bir not: Bir komutun açıklamasında (yukarıdaki gibi) bir argümanın ardından üç nokta geliyorsa, bu argümanın tekrarlanabileceği anlamına gelir. Dolayısıyla, aşağıdaki komut `dir1` adlı tek bir dizin oluşturur

```
mkdir dir1
```

Aşağıdaki komut ise `dir1`, `dir2` ve `dir3` adlı üç dizin oluşturur:

```
mkdir dir1 dir2 dir3
```

`cp` – Dosya ve Dizinleri Kopyalama

`cp` komutu dosya veya dizinleri kopyalar. İki farklı şekilde kullanılabilir. Aşağıdaki komut, tek bir dosya veya dizin olan `item1'i` dosya veya dizin `item2'ye` kopyalar

```
cp item1 item2
```

Aşağıdaki komut ise birden fazla öğeyi (dosya veya dizin) bir dizine kopyalar:

```
cp öğe... dizin
```

'da Kullanışlı Seçenekler ve Örnekler

Tablo 4-4, `cp` için yaygın olarak kullanılan bazı seçenekleri listelemektedir.

Tablo 4-4: Yaygın `cp` Seçenekleri

Seçenek	Uzun seçenek	Anlam
`-a`	`--archive`	Dosyaları, dizinleri ve sahiplik ve izinler dahil tüm özniteliklerini kopyalar. Normalde, kopyalar kopyalama işlemini gerçekleştiren kullanıcının varsayılan özniteliklerini alır. Dosya izinlerine 9. bölümde değineceğiz.
`-`	`--interactive`	Mevcut bir dosyanın üzerine yazmadan önce kullanıcıdan onay isteyin. **Bu seçenek belirtilmezse, `cp` dosyaları sessizce (yani uyarı vermeden) üzerine yazar.**
`-r`	`--recursive`	Dizinleri ve içeriklerini yinelemeli olarak kopyalar. Dizinleri kopyalarken bu seçenek (veya `-a` seçeneği) gereklidir.
`-u`	`--update`	Bir dizinden başka bir dizine dosya kopyalarken, hedef dizinde mevcut olmayan veya mevcut dosyalardan daha yeni olan dosyaları kopyalayın. Bu,

		kopya alınması gereken dosyaları atlayarak büyük miktarda dosya kopyalanırken kullanışlıdır.
`-v`	`--verbose`	Kopyalama işlemi gerçekleştirilirken bilgilendirici mesajlar görüntüleyin.

Tablo 4-5, bazı tipik `cp` kullanım örneklerini listeler.

Tablo 4-5: Yaygın `cp` Uygulamaları

Komut	Sonuç
`cp` *`dosya1 dosya2`* **üzerine yazılır.**	*file1 dosyasını file2* dosyasına kopyalar. ***file2 dosyası* varsa,** ***file1* dosyasının içeriği ile.** *file2 dosyası* mevcut değilse, oluşturulur.
`cp -i` *`dosya1 dosya2`*	Önceki komutla aynıdır, ancak *file2 dosyası* varsa dosyası varsa, üzerine yazılmadan önce kullanıcıya sorulur.
`cp` *`file1 file2 dir1`*	*file1* ve *file2* dosyalarını *dir1* dizinine kopyalar. *dir1* dizini önceden mevcut olmalıdır.
`cp dir1/* dir2` içine kopyalar.	Joker karakter kullanarak, *dir1* içindeki tüm dosyaları dir2 *dir2* dizinine kopyalar. *dir2* dizini önceden oluşturulmuş olmalıdır.
`cp -r` *`dir1 dir2`*	*dir1* dizinindeki içeriği dir2 dizinine kopyalayın. *dir2 dizinine kopyalayın*. *dir2* dizini mevcut değilse, oluşturulur ve kopyalama işleminden sonra *dir1* diziniyle aynı içeriği barındırır. *dir2* dizini mevcutsa, *dir1* dizini (ve içeriği) *dir2* dizinine kopyalanır.

`mv` – Dosyaları Taşıma ve Yeniden Adlandırma

`mv` komutu, nasıl kullanıldığına bağlı olarak hem dosya taşıma hem de dosya yeniden adlandırma işlemlerini gerçekleştirir. Her iki durumda da, işlemden sonra orijinal dosya adı artık mevcut olmaz. `mv`, burada gösterildiği gibi, `cp` ile hemen hemen aynı şekilde kullanılır ve dosya veya dizin öğesini taşır veya yeniden adlandırır `item1 item2`

```
mv item1 item2
```

veya bir veya daha fazla öğeyi bir dizinden başka bir dizine taşımak için:

```
mv öğe... dizin
```

Yararlı Seçenekler ve Örnekleri

mv, Tablo 4-6'da açıklanan cp ile birçok seçeneği paylaşır.

Tablo 4-6: mv Seçenekleri

Seçenek	Uzun seçenek	Anlam
-i	--interactive	Mevcut bir dosyayı üzerine yazmadan önce kullanıcıdan onay isteyin. **Bu seçenek belirtilmezse, mv dosyaları sessizce üzerine yazar.**
-u	--update	Dosyaları bir dizinden başka bir dizine taşırken, yalnızca mevcut olmayan veya hedef dizindeki mevcut dosyalardan daha yeni olan dosyaları taşır.
-v	--verbose	Taşıma işlemi gerçekleştirilirken bilgilendirici mesajlar görüntüler.

Tablo 4-7, mv kullanımına ilişkin bazı örnekler sunmaktadır.

Tablo 4-7: mv Örnekleri

Komut	Sonuç
mv *file1 file2*	*file1 dosyasını file2 dosya*sına taşır. ***file2* dosyası varsa silinir ve *file1 dosyası file2 olarak* yeniden adlandırılır.** *file2* dosyası yoksa, *file1 dosyası* sadece *file2 olarak* yeniden adlandırılır. **Her iki durumda da *file1* dosyası ortadan kalkar.**
mv -i *dosya1 dosya2*	Önceki komutla aynıdır, ancak *dosya2* varsa, silinmeden önce kullanıcıya sorulur.
mv *dosya1 dosya2 dir1*	*file1* ve *file2 dosyalarını dir1* dizinine taşır. *dir1* dizini önceden mevcut olmalıdır.
mv *dir1 dir2*	*dir2* dizini mevcut değilse, *dir1'i dir2* olarak yeniden adlandırır. *dir2* dizini mevcutsa, *dir1* dizinini (ve içeriğini) *dir2* dizinine taşır.

`rm` – Dosyaları ve Dizinleri Sil

`rm` komutu, aşağıdaki örnekte gösterildiği gibi dosya ve dizinleri kaldırmak (silmek) için kullanılır:

```
rm öğe...
```

Burada öğe, bir veya daha fazla dosya veya dizindir.

Yararlı Seçenekler ve rm Örnekleri

Tablo 4-8, `rm` için yaygın olarak kullanılan bazı seçenekleri açıklamaktadır.

Tablo 4-8: `rm` Seçenekleri

Seçenek	Uzun Seçenek	Anlam
`-i`	`--interactive`	Mevcut bir dosyayı silmeden önce kullanıcıdan onay isteyin. **Bu seçenek belirtilmezse, `rm` dosyaları sessizce siler.**
`-r`	`--recursive`	Dizinleri yinelemeli olarak sil. Bu, silinen dizinin alt dizinleri varsa, bunları da silmek anlamına gelir. Bir dizini silmek için bu seçenek belirtilmelidir.
`-f`	`--force`	Var olmayan dosyaları yok say ve sorma. Bu, `--interactive` seçeneğini geçersiz kılar.
`-v`	`--verbose`	Silme işlemi gerçekleştirilirken bilgilendirici mesajlar görüntüler.

Tablo 4-9, `rm` komutunun kullanımına ilişkin bazı örnekler sunmaktadır.

Tablo 4-9: `rm` Örnekleri

Komut	Sonuç
`rm file1`	*file1 dosyasını* sessizce siler.
`rm -i file1`	Önceki komutla aynıdır, ancak silme işlemi gerçekleştirilmeden önce kullanıcıdan onay istenir.
`rm -r dosya1 dizin1`	*file1* ve *dir1* ile içeriğini siler.

`rm -rf file1 dir1`	Önceki komutla aynıdır, ancak *file1* veya *dir1* mevcut değilse `rm` sessizce devam eder.

`rm` komutunu kullanırken dikkatli olun!

Linux gibi Unix benzeri işletim sistemlerinde silme işlemini geri alma komutu yoktur. `rm` ile bir şeyi sildiğinizde, o şey tamamen silinir. Linux, sizin akıllı olduğunuzu ve ne yaptığınızı bildiğinizi varsayar.

Joker karakterlere özellikle dikkat edin. Bu klasik örneği ele alalım. Diyelim ki bir dizindeki HTML dosyalarını silmek istiyorsunuz. Bunu yapmak için şunu yazın:

```
rm *.html
```

Bu doğru bir işlemdir, ancak yanlışlıkla `*` ve `.html` arasına yanlışlıkla boşluk koyarsanız:

```
rm * .html
```

`rm` komutu dizindeki tüm dosyaları siler ve ardından `.html` adlı bir dosya olmadığı konusunda hata verir.

İşte yararlı bir ipucu: `rm` ile joker karakterler kullandığınızda (yazdıklarınızı dikkatlice kontrol etmenin yanı sıra!), önce joker karakteri `ls` ile test edin. Bu, silinecek dosyaları görmenizi sağlar. Ardından yukarı ok tuşuna basarak komutu geri çağırın ve `ls` yerine `rm` yazın.

`ln` – Bağlantı Oluşturma

`ln` komutu, sabit veya sembolik bağlantılar oluşturmak için kullanılır. İki şekilde kullanılır. Aşağıdaki komut sabit bir bağlantı oluşturur:

```
ln dosya bağlantı
```

Aşağıdaki komut sembolik bağlantı oluşturur

```
ln -s öğe bağlantısı
```

burada `öğe` bir dosya veya dizindir.

Sabit Bağlantıları

Sabit bağlantılar, daha modern olan sembolik bağlantılara kıyasla, Unix'in orijinal bağlantı oluşturma yöntemidir. Varsayılan olarak, her dosya, dosyaya adını veren tek bir sabit bağlantıya sahiptir. Sabit bir bağlantı oluşturduğumuzda, dosya için ek bir dizin girişi oluştururuz. Sabit bağlantıların iki önemli sınırlaması vardır.

1. Sabit bağlantı, kendi dosya sistemi dışındaki bir dosyaya referans veremez. Bu, bağlantının, bağlantının kendisinin bulunduğu disk bölümünde bulunmayan bir dosyaya referans veremeyeceği anlamına gelir.
2. Sabit bağlantı bir dizine referans veremez.

Sabit bağlantı, dosyanın kendisinden ayırt edilemez. Sembolik bağlantının aksine, sabit bağlantı içeren bir dizini listelediğimizde, bağlantının özel bir göstergesini görmeyiz. Sabit bağlantı silindiğinde, bağlantı kaldırılır, ancak dosyanın içeriği, dosyaya olan tüm bağlantılar silinene kadar varlığını sürdürür (yani, alanı serbest bırakılmaz).

Sabit bağlantıları bilmek önemlidir, çünkü zaman zaman bunlarla karşılaşabilirsiniz, ancak modern uygulamalarda sembolik bağlantılar tercih edilmektedir. Sembolik bağlantıları bir sonraki bölümde ele alacağız.

Sembolik Bağlantılar

Sembolik bağlantılar, sabit bağlantıların sınırlamalarını aşmak için oluşturulmuştur. Sembolik bağlantılar, başvurulan dosya veya dizine bir metin işaretçisi içeren özel bir dosya türü oluşturarak çalışır. Bu bakımdan, Windows kısayoluyla hemen hemen aynı şekilde çalışırlar, ancak elbette Windows özelliğinden yıllar önce ortaya çıkmışlardır.

Sembolik bir bağlantının işaret ettiği dosya ile sembolik bağlantının kendisi birbirinden büyük ölçüde ayırt edilemez. Örneğin, sembolik bağlantıya bir şey yazarsak, başvurulan dosya yazılır. Ancak, sembolik bir bağlantıyı sildiğimizde, yalnızca bağlantı silinir, dosyanın kendisi silinmez. Dosya sembolik bağlantıdan önce silinirse, bağlantı var olmaya devam eder, ancak hiçbir şeyi işaret etmez. Bu durumda, bağlantının *bozuk* olduğu söylenir. Birçok uygulamada, `ls` komutu bozuk bağlantıları kırmızı gibi ayırt edici bir renkle göstererek bunların varlığını ortaya çıkarır.

Bağlantı kavramı kafa karıştırıcı görünebilir, ancak sabırlı olun. Tüm bunları deneyeceğiz ve her şey netleşecek.

Bir Oyun Alanı Oluşturalım

Gerçek dosya işlemleri yapacağımız için, dosya işlem komutlarımızla "oynamak" için güvenli bir yer oluşturalım. Öncelikle, çalışmak için bir dizine ihtiyacımız var. Ana dizinde bir dizin oluşturacağız ve ona `oyun alanı adını` vereceğiz.

su Dizinleri Oluşturma

`mkdir` komutu, bir dizin oluşturmak için kullanılır. Oyun alanı dizinimizi oluşturmak için, önce ana dizinde olduğumuzdan emin olacağız ve ardından yeni dizini oluşturacağız.

```
[me@linuxbox ~]$ cd
[me@linuxbox ~]$ mkdir playground
```

Oyun alanımızı biraz daha ilginç hale getirmek için, içinde `dir1` ve `dir2` adlı iki dizin oluşturalım. Bunu yapmak için, mevcut çalışma dizinimizi `playground` olarak değiştirip başka bir `mkdir komutu` çalıştıracağız.

```
[me@linuxbox ~]$ cd playground [me@linuxbox
playground]$ mkdir dir1 dir2
```

`mkdir` komutunun birden fazla argüman kabul ettiğini ve tek bir komutla her iki dizini de oluşturabileceğimizi unutmayın.

'e Dosya Kopyalama

Şimdi, playground'a bazı veriler ekleyelim. Bunu bir dosyayı kopyalayarak yapacağız. `cp` komutunu kullanarak, `passwd` dosyasını `/etc` dizininden mevcut çalışma dizinine kopyalayacağız.

```
[me@linuxbox playground]$ cp /etc/passwd .
```

Geçerli çalışma dizini için kısaltma kullandığımıza dikkat edin, tek bir nokta. Şimdi `ls` komutunu çalıştırırsak, dosyamızı göreceğiz.

```
[me@linuxbox playground]$ ls -l
toplam 12
drwxrwxr-x 2 me me 4096 2025-01-10 16:40 dir1
drwxrwxr-x 2 me me 4096 2025-01-10 16:40 dir2
-rw-r--r-- 1 me me 1650 2025-01-10 16:07 passwd
```

Şimdi, sadece eğlence için, "-v" seçeneğini (ayrıntılı) kullanarak kopyalamayı tekrarlayalım ve ne yaptığını görelim.

```
[me@linuxbox playground]$ cp -v /etc/passwd .
`/etc/passwd' -> `./passwd'
```

cp komutu kopyalamayı tekrar gerçekleştirdi, ancak bu sefer gerçekleştirdiği işlemi belirten kısa bir mesaj görüntüledi. cp'nin herhangi bir uyarı vermeden ilk kopyayı üzerine yazdığını unutmayın. Yine, bu durumda cp, ne yaptığımızı bildiğimizi varsayıyor. Uyarı almak için "-i" (interactive) seçeneğini ekleyeceğiz.

```
[me@linuxbox playground]$ cp -i /etc/passwd .
cp: `./passwd' dosyasını üzerine yazmak istiyor musunuz?
```

Komut istemine y girerek yanıt verirseniz dosya üzerine yazılır; başka herhangi bir karakter (örneğin n) girerseniz cp dosyayı olduğu gibi bırakır.

Dosyalarını Taşıma ve Yeniden Adlandırma

Şimdi, passwd adı pek eğlenceli görünmüyor ve burası bir oyun alanı, o yüzden bunu başka bir şeyle değiştirelim.

```
[me@linuxbox playground]$ mv passwd fun
```

Yeniden adlandırdığımız dosyayı her bir dizine taşıyıp tekrar geri getirerek biraz eğlenelim. Aşağıdaki komut dosyayı önce dir1 dizinine taşır:

```
[me@linuxbox playground]$ mv fun dir1
```

Aşağıdaki komutla dir1'den dir2'ye taşınır:

```
[me@linuxbox playground]$ mv dir1/fun dir2
```

Son olarak, aşağıdaki komut onu mevcut çalışma dizinine geri getirir:

```
[me@linuxbox playground]$ mv dir2/fun .
```

Şimdi, mv komutunun dizinler üzerindeki etkisini görelim. İlk olarak, veri dosyamızı dir1

şöyle:

```
[me@linuxbox playground]$ mv fun dir1
```

Ardından `dir1'i dir2`'ye taşıyoruz ve `ls` ile onaylıyoruz.

```
[me@linuxbox playground]$ mv dir1 dir2
[me@linuxbox playground]$ ls -l dir2
toplam 4
drwxrwxr-x 2 me me     4096 2025-01-11 06:06 dir1
[me@linuxbox playground]$ ls -l dir2/dir1
toplam 4
-rw-r--r-- 1 ben ben   1650 2025-01-10 16:33 eğlence
```

`dir2` zaten mevcut olduğu için, `mv dir1'i dir2'ye` taşıdı. `dir2` mevcut olmasaydı, `mv dir1'i dir2` olarak yeniden adlandırırdı. Son olarak, her şeyi geri koyalım.

```
[me@linuxbox playground]$ mv dir2/dir1 .
[me@linuxbox playground]$ mv dir1/fun .
```

Sabit Bağlantılar ( ) Oluşturma

Şimdi bazı bağlantıları deneyelim. Öncelikle veri dosyamız için şu şekilde bazı sabit bağlantılar oluşturalım:

```
[me@linuxbox playground]$ ln fun fun-hard
[me@linuxbox playground]$ ln fun dir1/fun-hard
[me@linuxbox playground]$ ln fun dir2/fun-hard
```

Şimdi `fun` dosyasının dört örneği var. Oyun alanı dizinimize bir göz atalım.

```
[me@linuxbox playground]$ ls -l
toplam 16
drwxrwxr-x 2 me    me    4096 2025-01-14 16:17 dir1
drwxrwxr-x 2 ben   me    4096 2025-01-14 16:17 dir2
-rw-r--r-- 4 ben   me    1650 2025-01-10 16:33 eğlence
-rw-r--r-- 4 ben   ben   1650 2025-01-10 16:33 eğlence-sert
```

Dikkatimizi çeken bir şey, `fun` ve `fun-hard` listelerinin ikinci alanlarının her ikisinin de `4` rakamını içermesidir. Bu rakam, dosya için şu anda mevcut olan sabit bağlantıların sayısıdır. Dosyanın adı bir bağlantı tarafından oluşturulduğundan, bir dosyanın her zaman en az bir bağlantısı olacağını unutmayın. Peki, `fun` ve `fun-hard` dosyalarının aslında aynı dosya olduğunu nasıl anlarız? Bu durumda, `ls komutu` pek yardımcı olmuyor. `fun` ve `fun-hard` dosyalarının boyutlarının aynı olduğunu (5. alan) görebiliyoruz, ancak listemizden bunu kesin olarak anlamamız mümkün değil. Bu sorunu çözmek için biraz daha derinlemesine incelememiz gerekecek.

Sabit bağlantıları düşünürken, dosyaların iki bölümden oluştuğunu hayal etmek yardımcı olabilir.

1. Dosyanın içeriğini içeren veri kısmı
2. Dosyanın adını içeren ad kısmı

Sabit bağlantılar oluşturduğumuzda, aslında aynı veri bölümüne atıfta bulunan ek ad bölümleri oluşturuyoruz. Sistem, bir *inode* olarak adlandırılan şeye bir disk bloğu zinciri atar ve bu zincir daha sonra ad bölümüyle ilişkilendirilir. Bu nedenle, her sabit bağlantı, dosyanın içeriğini barındıran belirli bir inode'a atıfta bulunur.

`ls` komutu bu bilgiyi ortaya çıkarmak için bir yol sunar. Bu komut, `-i` seçeneği ile çağrılır.

```
[me@linuxbox playground]$ ls -li
toplam 16
12353539  drwxrwxr-x  2 me   me    4096  2025-01-14  16:17 dir1
12353540  drwxrwxr-x  2 ben  ben   4096  2025-01-14  16:17 dir2
12353538  -rw-r--r--  4 ben  ben   1650  2025-01-10  16:33 eğlence
12353538  -rw-r--r--  4 ben  ben   1650  2025-01-10  16:33 fun-hard
```

Bu listeleme sürümünde, ilk alan inode numarasıdır ve gördüğümüz gibi, hem `fun` hem de `fun-hard` aynı inode numarasını paylaşmaktadır, bu da bunların aynı dosya olduğunu doğrulamaktadır.

Sembolik Bağlantıları Oluşturma

Sembolik bağlantılar, sabit bağlantıların iki dezavantajını ortadan kaldırmak için oluşturulmuştur.

1. Sabit bağlantılar fiziksel cihazları kapsayamaz.
2. Sabit bağlantılar dizinlere değil, yalnızca dosyalara referans verebilir.

Sembolik bağlantılar, hedef dosya veya dizine metin işaretçisi içeren özel bir dosya türüdür.

Sembolik bağlantıların oluşturulması, sabit bağlantıların oluşturulmasına benzer.

```
[me@linuxbox playground]$ ln -s fun fun-sym [me@linuxbox
playground]$ ln -s ../fun dir1/fun-sym [me@linuxbox
playground]$ ln -s ../fun dir2/fun-sym
```

İlk örnek oldukça basit; sert bağlantı yerine sembolik bağlantı oluşturmak için "-s" seçeneğini ekliyoruz. Peki ya sonraki iki örnek? Unutmayın, sembolik bağlantı oluşturduğumuzda, hedef dosyanın sembolik bağlantıya göre nerede olduğunu belirten bir metin açıklaması oluşturuyoruz. Burada gösterilen `ls` çıktısına bakarsak bunu daha kolay anlayabiliriz:

```
[me@linuxbox playground]$ ls -l dir1
toplam 4
-rw-r--r-- 4 me    me    1650 2025-01-10 16:33 fun-hard
lrwxrwxrwx 1 me    me       6 2025-01-15 15:17 eğlence-sym -> ../eğlence
```

`dir1` içindeki `fun-sym` listesi, ilk alanda bulunan `l harfi` ile bunun bir sembolik bağlantı olduğunu ve `../fun'a` işaret ettiğini gösterir, ki bu doğrudur. `fun-sym'in` konumuna göre, `fun` onun üstündeki dizinde bulunur. Sembolik bağlantı dosyasının uzunluğunun 6 olduğunu, yani `../fun` dizesindeki karakter sayısını gösterdiğini, işaret ettiği dosyanın uzunluğunu göstermediğini de unutmayın.

Sembolik bağlantılar oluştururken, burada gösterildiği gibi mutlak yol adlarını kullanabiliriz

```
[me@linuxbox playground]$ ln -s /home/me/playground/fun dir1/fun-sym
```

veya önceki örneğimizde yaptığımız gibi göreceli yol adları. Çoğu durumda, göreceli yol adlarının kullanılması daha tercih edilir, çünkü bu genellikle sembolik bağlantılar ve bunların referans aldığı dosyaları içeren bir dizin ağacının, bağlantıları bozmadan yeniden adlandırılmasına veya taşınmasına olanak tanır.

Normal dosyalara ek olarak, sembolik bağlantılar dizinlere de referans verebilir.

```
[me@linuxbox playground]$ ln -s dir1 dir1-sym [me@linuxbox
playground]$ ls -l
toplam 16
 drwxrwxr-x  2 me   ben   4096  2025-01-15  15:17 dir1
 lrwxrwxrwx  1 ben  ben      4  2025-01-16  14:45 dir1-sym -> dir1
 drwxrwxr-x  2 ben  ben   4096  2025-01-15  15:17 dir2
 -rw-r--r--  4 ben  ben   1650  2025-01-10  16:33 eğlence
 -rw-r--r--  4 ben  ben   1650  2025-01-10  16:33 eğlenceli-zor
 lrwxrwxrwx  1 ben  ben      3  2025-01-15  15:15 fun-sym -> fun
```

'da Dosya ve Dizinleri Silme

Daha önce bahsettiğimiz gibi, `rm` komutu dosya ve dizinleri silmek için kullanılır. Bunu, oyun alanımızı biraz temizlemek için kullanacağız. İlk olarak, sabit bağlantılarımızdan birini silelim.

```
[me@linuxbox playground]$ rm fun-hard
[me@linuxbox playground]$ ls -l
toplam 12
drwxrwxr-x 2 me    me    4096 2025-01-15 15:17 dir1
lrwxrwxrwx 1 ben   ben      4 2025-01-16 14:45 dir1-sym -> dir1
drwxrwxr-x 2 ben   ben   4096 2025-01-15 15:17 dir2
-rw-r--r-- 3 ben   ben   1650 2025-01-10 16:33 eğlence
lrwxrwxrwx 1 ben   ben      3 2025-01-15 15:15 eğlence-sym -> eğlence
```

Beklenen şekilde çalıştı. `fun-hard` dosyası kayboldu ve `fun` için gösterilen bağlantı sayısı, dizin listesinin ikinci alanında belirtildiği gibi dörttan üçe düştü. Şimdi, `fun` dosyasını sileceğiz ve eğlencesine, bunun ne işe yaradığını göstermek için `-i` seçeneğini ekleyeceğiz.

```
[me@linuxbox playground]$ rm -i fun rm:
normal dosya `fun'u silmek ister
misiniz?
```

Komut isteminde `y` yazın ve dosya silinir. Şimdi `ls` komutunun çıktısına bakalım. `fun-sym'e` ne olduğunu fark ettiniz mi? Artık mevcut olmayan bir dosyaya işaret eden sembolik bir bağlantı olduğu için bağlantı *bozulmuştur*.

```
[me@linuxbox playground]$ ls -l
toplam 8
drwxrwxr-x 2 me    me    4096 2025-01-15 15:17 dir1
lrwxrwxrwx 1 me    me       4 2025-01-16 14:45 dir1-sym -> dir1
drwxrwxr-x 2 ben   me    4096 2025-01-15 15:17 dir2
lrwxrwxrwx 1 ben   ben      3 2025-01-15 15:15 fun-sym -> fun
```

Çoğu Linux dağıtımı, `ls komutunu` bozuk bağlantıları gösterecek şekilde yapılandırır. Bozuk bir bağlantının varlığı doğası gereği tehlikeli değildir, ancak oldukça dağınıktır. Bozuk bir bağlantıyı kullanmaya çalışırsak, şunu görürüz:

```
[me@linuxbox playground]$ less fun-sym
fun-sym: Böyle bir dosya veya dizin yok
```

Biraz temizlik yapalım. Buradaki sembolik bağlantıları sileceğiz:

```
[me@linuxbox playground]$ rm fun-sym dir1-sym
[me@linuxbox playground]$ ls -l
toplam 8
drwxrwxr-x 2 me    me    4096 2025-01-15 15:17 dir1
drwxrwxr-x 2 ben   ben   4096 2025-01-15 15:17 dir2
```

Sembolik bağlantılarla ilgili unutulmaması gereken bir nokta, çoğu dosya işleminin bağlantının kendisi üzerinde değil, bağlantının hedefi üzerinde gerçekleştirildiğidir. `rm` bir istisnadır. Bir bağlantıyı sildiğimizde, silinen bağlantıdır, hedef değildir.

Son olarak, playground'ımızı kaldıracağız. Bunu yapmak için, ana dizine geri döneceğiz ve `rm` komutunu özyinelemeli seçenek (`-r`) ile kullanarak `playground'ı` ve alt dizinleri dahil tüm içeriğini sileceğiz.

```
[me@linuxbox playground]$ cd
[me@linuxbox ~]$ rm -r playground
```

GUI ile Sembolik Bağlantılar Oluşturma

GNOME ve KDE'deki dosya yöneticileri, sembolik bağlantılar oluşturmak için kolay ve otomatik bir yöntem sunar. GNOME'da, bir dosyayı sürüklerken Ctrl+Shift tuşlarını basılı tutarsanız, dosya kopyalanmak (veya taşınmak) yerine bir bağlantı oluşturulur. KDE'de, bir dosya bırakıldığında küçük bir menü belirir ve dosyayı kopyalama, taşıma veya bağlantı oluşturma seçenekleri sunar.

Özet

Burada çok fazla konuyu ele aldık ve bunların hepsini tam olarak kavramak biraz zaman alacak. Anlayana kadar oyun alanı alıştırmasını tekrar tekrar yapın. Temel dosya işleme komutlarını ve joker karakterleri iyi anlamak önemlidir. Daha fazla dosya ve dizin ekleyerek, çeşitli işlemler için dosyaları belirtmek üzere joker karakterler kullanarak oyun alanı alıştırmasını genişletmekten çekinmeyin. Bağlantı kavramı ilk başta biraz kafa karıştırıcı olabilir, ancak nasıl çalıştıklarını öğrenmek için zaman ayırın. Gerçekten hayat kurtarıcı olabilirler.

Daha fazla bilgi

- Sembolik bağlantılar hakkında bir tartışma: http://en.wikipedia.org/wiki/Symbolic_link

5 – Komutlarla Çalışma

Şu ana kadar, her biri kendi gizemli seçenekleri ve argümanları olan bir dizi gizemli komut gördük. Bu bölümde, bu gizemi biraz ortadan kaldırmaya ve hatta kendi komutlarımızı oluşturmaya çalışacağız. Bu bölümde tanıtılan komutlar şunlardır:

- `type` – Bir komut adının nasıl yorumlandığını belirtir.
- `which` – Hangi yürütülebilir programın çalıştırılacağını gösterir.
- `help` – Kabuk yerleşik komutları için yardım alır.
- `man` – Bir komutun kılavuz sayfasını görüntüler.
- `apropos` – Uygun komutların listesini görüntüler.
- `info` – Bir komutun bilgi girişini görüntüler.
- `whatis` – Tek satırlık kılavuz sayfası açıklamalarını görüntüler.
- `alias` – Bir komut için takma ad oluşturur.

Komutlar Tam Olarak Nedir?

Bir komut dört farklı şeyden biri olabilir:

- `/usr/bin` dizininde gördüğümüz tüm dosyalar gibi **yürütülebilir bir program**. Bu kategori içinde, programlar C ve C++ ile yazılmış programlar gibi *derlenmiş ikili dosyalar* veya kabuk, Perl, Python, Ruby vb. gibi *betik dillerinde* yazılmış programlar olabilir.
- **Kabuğun içine yerleştirilmiş bir komut**. `bash`**,** *kabuk yerleşik komutları* olarak adlandırılan bir dizi komutu dahili olarak destekler. Örneğin, `cd` komutu bir kabuk yerleşik komutudur.
- **Bir kabuk işlevi.** Kabuk işlevleri, *ortama* dahil edilmiş minyatür kabuk betikleridir. Ortamın yapılandırılması ve kabuk işlevlerinin yazılması konularını sonraki bölümlerde ele alacağız, ancak şimdilik bunların varlığından haberdar olun.
- **Bir takma ad.** Takma adlar, diğer komutlardan oluşturulan ve kendimiz tanımlayabileceğimiz komutlardır.

Komutları Tanımlama

Dört komut türünden hangisinin kullanıldığını tam olarak bilmek genellikle yararlıdır ve Linux bunu öğrenmek için birkaç yol sunar.

`type` – Komutun Türünü Görüntüleme

`type` komutu, belirli bir komut adı verildiğinde kabuğun yürüteceği komut türünü görüntüleyen bir kabuk yerleşik komutudur. Şu şekilde çalışır:

```
type komut
```

Burada, *command* incelemek istediğimiz komutun adıdır. Bazı örneklere göz atalım:

```
[me@linuxbox ~]$ type type
type bir kabuk yerleşik
komutudur [me@linuxbox ~]$
type ls
ls, `ls --color=auto' komutunun takma
adıdır [me@linuxbox ~]$ type cp
cp, /usr/bin/cp'dir
```

Burada üç farklı komutun sonuçlarını görüyoruz. `ls` komutuna dikkat edin (Fedora sisteminden alınmıştır) ve `ls` komutunun aslında `--color=auto` seçeneği eklenmiş ls komutunun bir takma adı olduğunu fark edin. Artık `ls` komutunun çıktısının neden renkli görüntülendiğini biliyoruz!

`which` – Yürütülebilir Dosyanın Konumunu Görüntüleme ( )

Bazen bir sistemde birden fazla sürümde yürütülebilir program yüklü olabilir. Bu durum masaüstü sistemlerinde yaygın olmasa da, büyük sunucularda olağandışı değildir. Belirli bir yürütülebilir dosyanın tam konumunu belirlemek için `which` komutu kullanılır.

```
[me@linuxbox ~]$ which ls
/usr/bin/ls
```

`which` komutu yalnızca yürütülebilir programlar için çalışır, gerçek yürütülebilir programların yerine geçen yerleşik komutlar veya takma adlar için çalışmaz. Örneğin, `cd` gibi bir kabuk yerleşik komutunda `which komutunu` kullanmaya çalıştığımızda, ya hiçbir yanıt almayız ya da bir hata mesajı alırız.

```
[me@linuxbox ~]$ which cd
/usr/bin/which: (/usr/local/bin:/usr/bin:/bin:/usr/local
/games:/usr/games)
```

Bu yanıt, "komut bulunamadı" demek için kullanılan süslü bir ifadedir.

Komutun Belgelerini Alma

Komutun ne olduğu hakkında bu bilgiye sahip olarak, artık her komut türü için mevcut olan belgeleri arayabiliriz.

`help` – Kabuk i Yerleşik Komutları için Yardım Alın

bash, her bir kabuk yerleşik komutu için yerleşik bir yardım özelliğine sahiptir. Bunu kullanmak için `help komutunu yazın` ve ardından kabuk yerleşik komutunun adını yazın. İşte bir örnek:

```
[me@linuxbox ~]$ yardım cd
cd: cd [-L|[-P [-e]] [-@]] [dir]
    Kabuk çalışma dizinini değiştirir.

    Geçerli dizini DIR olarak değiştirir. Varsayılan DIR, HOME kabuk
    değişkeninin değeridir.

    CDPATH değişkeni, DIR'yi içeren dizinin arama yolunu tanımlar.
    CDPATH'deki alternatif dizin adları iki nokta üst üste (:) ile
    ayrılır. Boş dizin adı, geçerli dizinle aynıdır. DIR eğik çizgi (/)
    ile başlıyorsa, CDPATH kullanılmaz.

    Dizin bulunamazsa ve kabuk seçeneği `cdable_vars' ayarlanmışsa,
    kelime bir değişken adı olarak kabul edilir. Bu değişkenin bir değeri
    varsa, değeri DIR için kullanılır.

    Seçenekler:
       -L     sembolik bağlantıların izlenmesini zorla: `..'
              örneklerini işledikten sonra DIR içindeki sembolik
              bağlantıları çöz
       -P     sembolik bağlantıları takip etmeden fiziksel dizin yapısını
              kullan: `..' örneklerini işlemeden önce DIR içindeki
              sembolik bağlantıları çöz
       -e     -P seçeneği belirtilmişse ve geçerli çalışma dizini
              başarıyla belirlenemiyorsa, sıfır olmayan bir durumla çıkın
```

```
       -@     destekleyen sistemlerde, genişletilmiş
              özellikleri olan bir dosyayı, dosya özelliklerini içeren bir
              dizin olarak sunar.
     Varsayılan ayar, `-L' belirtilmiş gibi sembolik bağlantıları takip
    etmektir. `..' hemen önceki yol adı bileşenini bir eğik çizgiye veya
    DIR'nin başına kadar kaldırarak işlenir.

    Çıkış Durumu:
    Dizin değiştirilirse ve -P kullanıldığında $PWD başarıyla ayarlanırsa 0
    döndürür; aksi takdirde sıfırdan farklı bir değer döndürür.
```

Notasyonla ilgili bir not: Bir komutun sözdiziminin açıklamasında köşeli parantezler göründüğünde, bunlar isteğe bağlı öğeleri gösterir. Dikey çubuk karakteri, birbirini dışlayan öğeleri gösterir. Örneğin, önceki `cd` komutunu ele alalım:

```
cd [-L|[-P[-e]]] [dir]
```

Bu notasyon, `cd` komutunun ardından isteğe bağlı olarak `-L` veya `-P` gelebileceğini belirtir; ayrıca, `-P` seçeneği belirtilirse, `-e` seçeneği isteğe bağlı `dir` argümanı ile birlikte eklenebilir.

`cd` komutları için `yardım` çıktısı kısa ve doğrudur, ancak hiçbir şekilde öğretici değildir ve gördüğümüz gibi, henüz bahsetmediğimiz birçok şeyi de içeriyor gibi görünüyor! Endişelenmeyin. O kısma da geleceğiz.

Yararlı ipucu: `-m` seçeneği ile `yardım` komutunu kullanarak, `yardım` komutu çıktısını man sayfasına benzeyen alternatif bir biçimde görüntüler (aşağıya bakın).

`--help` – Kullanım Bilgilerini Görüntüle

Birçok yürütülebilir program, komutun desteklediği sözdizimi ve seçeneklerin açıklamasını görüntüleyen "--help" seçeneğini destekler. Örneğin:

```
[me@linuxbox ~]$ mkdir --help
Kullanım: mkdir [SEÇENEK] DİZİN...
Henüz mevcut değilse, DİZİN(LER) oluşturun.

 Uzun seçenekler için zorunlu olan argümanlar, kısa seçenekler için de
zorunludur.
  -m, --mode=MODE    dosya modunu (chmod'da olduğu gibi) ayarlayın, a=rwx
  - umask değil
  -p, --parents      mevcutsa hata vermeyin, gerektiğinde üst dizinleri
                     oluşturun
  -v, --verbose      oluşturulan her dizin için bir mesaj yazdır
```

```
      --help         bu yardımı göster ve çık
      --version      sürüm bilgilerini görüntüler ve çıkar
   -Z, --context=CONTEXT (SELinux) güvenlik bağlamını CONTEXT olarak
   ayarla
Hataları <bug-coreutils@gnu.org> adresine bildirin.
```

Bazı programlar "--help" seçeneğini desteklemez, ancak yine de deneyin. Genellikle aynı kullanım bilgilerini gösteren bir hata mesajı ile sonuçlanır.

`man` – Bir Programın Kılavuzunu Görüntüleyin Sayfası

Komut satırı kullanımı için tasarlanmış çoğu yürütülebilir program, *kılavuz* veya *man sayfası* adı verilen resmi bir belge sağlar. Bunları görüntülemek için `man` adlı özel bir sayfalama programı kullanılır. *`Program`*, görüntülemek istediğiniz komutun adıdır:

```
man program
```

Man sayfalarının formatları biraz farklılık gösterebilir, ancak genellikle aşağıdakileri içerir:

- Bir başlık (sayfanın adı)
- Komutun sözdiziminin özeti
- Komutun amacının açıklaması
- Komutun her bir seçeneğinin listesi ve açıklaması

Ancak man sayfaları genellikle örnekler içermez ve bir öğretici değil, referans amaçlıdır. Örnek olarak, ls komutunun man sayfasını görüntülemeye çalışalım.

```
[me@linuxbox ~]$ man ls
```

Çoğu Linux sisteminde, `man komutu` man sayfasını görüntülemek için `less komutunu` kullanır, bu nedenle sayfayı görüntülerken tanıdık
`less` komutları sayfayı görüntülerken çalışır.

`man komutunun` görüntülediği "kılavuz" bölümlere ayrılmıştır ve yalnızca kullanıcı komutlarını değil, sistem yönetimi komutlarını, programlama arayüzlerini, dosya biçimlerini ve daha fazlasını da kapsar. Tablo 5-1 kılavuzun düzenini açıklamaktadır.

Tablo 5-1: Man Sayfası Düzeni

Bölüm	İçindekiler
1	Kullanıcı komutları

2	Çekirdek sistem çağrıları için programlama arayüzleri
3	C kütüphanesine programlama arayüzleri
4	Aygıt düğümleri ve sürücüler gibi özel dosyalar
5	Dosya biçimleri
6	Ekran koruyucular gibi oyunlar ve eğlenceler
7	Çeşitli
8	Sistem yönetimi komutları

Bazen aradığımız şeyi bulmak için kılavuzun belirli bir bölümüne başvurmamız gerekir. Bu, özellikle bir komutun adı olan bir dosya biçimi arıyorsak geçerlidir. Bölüm numarası belirtmeden, her zaman eşleşmenin ilk örneğini, muhtemelen bölüm 1'de buluruz. Bölüm numarasını belirtmek için `man komutunu` şu şekilde kullanırız:

```
man bölüm arama_terimi
```

İşte bir örnek:

```
[me@linuxbox ~]$ man 5 passwd
```

Bu, /etc/passwd dosyasının dosya biçimini açıklayan man sayfasını görüntüler.

`apropos` – Uygun Komutlarını Görüntüleme

Arama terimine göre man sayfalarının listesinde olası eşleşmeleri aramak da mümkündür. Bu yöntem kaba olmakla birlikte bazen yararlı olabilir. Aşağıda, *partition* arama terimini kullanarak man sayfalarını arama örneği verilmiştir:

```
[me@linuxbox ~]$ apropos partition
addpart (8)           - "add partition" komutunu saran basit bir
sarmalayıcı... all-swaps (7)    - tüm takas bölümlerinin... cfdisk (8) -
disk bölüm tablosunu görüntüleme veya değiştirme cgdisk (8)     - Curses
tabanlı GUID bölüm tablosu (GPT)... delpart (8)    - "bölüm silme"
işleminin basit bir sarmalayıcısı... fdisk (8)      - disk bölüm
tablosunu düzenleme
fixparts (8)          - MBR bölüm tablosu onarım yardımcı programı
```

```
gdisk (8)              - Etkileşimli GUID bölüm tablosu (GPT)...
mpartition (1)         - MSDOS sabit diskini bölümlemek
partprobe (8)          - bölüm tablosundaki değişiklikleri işletim
partx (8)                sistemine bildirme
resizepart (8)         - Linux çekirdeğine varlığını bildir...
sfdisk (8)             - "bölümü yeniden boyutlandır..." komutunu saran
sgdisk (8)               basit bir sarmalayıcı...
                       - Linux için bölüm tablosu düzenleyici
                       - Komut satırı GUID bölüm tablosu (GPT)...
```

Çıktının her satırındaki ilk alan man sayfasının adıdır ve ikinci alan bölümü gösterir. "-k" seçeneği ile `man` komutunun `apropos` ile aynı işlevi yerine getirdiğini unutmayın.

`whatis` – Tek Satırlık Kılavuz Sayfası Görüntüleme Açıklamaları

`whatsis` programı, belirtilen anahtar kelimeyle eşleşen bir man sayfasının adını ve tek satırlık açıklamasını görüntüler:

```
[me@linuxbox ~]$ whatis ls
ls                     (1) - dizin içeriğini listeler
```

En Zorlu Man Sayfası

Gördüğümüz gibi, Linux ve diğer Unix benzeri sistemlerle birlikte sağlanan man sayfaları, öğretici değil, referans belgesi olarak tasarlanmıştır. Birçok man sayfası okuması zordur, ancak bence zorluk açısından birincilik ödülünü `bash` man sayfası almaktadır. Bu kitap için araştırma yaparken, `bash` man sayfasını dikkatlice inceledim ve konuların çoğunu kapsadığından emin oldum. Yazdırıldığında 80 sayfadan uzun ve son derece yoğun olan bu sayfanın yapısı, yeni bir kullanıcı için hiç mantıklı gelmiyor.

Öte yandan, son derece doğru ve özlü olmasının yanı sıra son derece eksiksizdir. Cesaretiniz varsa bir göz atın ve onu okuyup her şeyi anlayabileceğiniz günü sabırsızlıkla bekleyin.

`info` – Bir Programın Bilgilerini Görüntüleme e Girişi

GNU Projesi, programları için man sayfalarına alternatif olarak *info* adlı bir seçenek sunar. Info kılavuzları, uygun bir şekilde `info` adlı bir okuyucu programla görüntülenir.

Info sayfaları, web sayfalarına benzer şekilde *hiperlinklerle* bağlantılıdır. İşte bir örnek:

```
Dosya: coreutils.info, Düğüm: ls çağrısı, Sonraki: dir çağrısı, Yukarı:
Dizin listesi
10.1 `ls': Dizin içeriğini listele
=================================
`ls' programı, dosyalar (dizinler dahil her türden) hakkındaki bilgileri
listeler. Seçenekler ve dosya argümanları, her zamanki gibi rastgele
karıştırılabilir.
   Dizin olan seçenek dışı komut satırı argümanları için, varsayılan olarak
`ls' dizinlerin içeriğini yinelemeli olarak listeler ve `.' ile başlayan
dosyaları atlar. Diğer seçenek dışı argümanlar için, varsayılan olarak `ls'
sadece dosya adını listeler. Seçenek dışı argüman belirtilmezse, `ls'
geçerli dizinde çalışır ve tek bir `.' argümanıyla çağrılmış gibi davranır.
   Varsayılan olarak, çıktı alfabetik olarak sıralanır,
--zz-Bilgi: (coreutils.info.gz)ls çağrısı, 63 satır --Üst----------
```

`info` programı, her biri tek bir konuyu içeren ayrı *düğümlere* ayrılmış ağaç yapısında *info dosyalarını* okur. Info dosyaları, okuyucuyu bir düğümden diğerine taşıyabilen köprüler içerir. Köprüler, başındaki yıldız işaretiyle tanınabilir ve üzerine imleci getirip `ENTER` tuşuna basarak etkinleştirilebilir.

`Info'yu` çağırmak için `info` yazın ve isteğe bağlı olarak bir programın adını ekleyin. Tablo 5-2, bir bilgi sayfasını görüntülerken okuyucuyu kontrol etmek için kullanılan komutları açıklamaktadır.

Tablo 5-2: `info` Komutları

Komut	Eylem
`?`	Komut yardımı görüntüleme
`PgUp` veya `Backspace`	Önceki sayfayı göster
`PgDn` veya `Boşluk`	Sonraki sayfayı göster
`n`	Sonraki: Sonraki düğümü görüntüle
`p`	Önceki: Önceki düğümü göster
`u`	Yukarı: Şu anda görüntülenen düğümün üst düğümünü görüntüler, genellikle bir menü
`ENTER`	İmlecin bulunduğu konumdaki köprüyü takip et
`q`	Çık

Şimdiye kadar ele aldığımız komut satırı programlarının çoğu GNU Projesi'nin *coreutils* paketinin bir parçasıdır, bu nedenle

```
[me@linuxbox ~]$ info coreutils
```

coreutils paketinde bulunan her programa hiperlinkler içeren bir menü sayfası görüntüler.

README ve Diğer Program Belgeleri Dosyaları

Sistemimizde yüklü olan birçok yazılım paketinin belgeleri `/usr/share/doc` dizininde yer alır. Bunların çoğu düz metin biçiminde (genellikle popüler bir düz metin belge biçimi olan Markdown'da) saklanır ve `less` ile görüntülenebilir. Bazı dosyalar HTML biçimindedir ve bir web tarayıcısıyla görüntülenebilir. Bazı dosyaların uzantısı ".gz" ile biter. Bu, `gzip` sıkıştırma programı ile sıkıştırıldıklarını gösterir. gzip paketi, gzip ile sıkıştırılmış metin dosyalarının içeriğini görüntüleyen `zless` adlı özel bir `less` sürümü içerir.

`alias` ile Kendi Komutlarımızı Oluşturma

Şimdi programlama ile ilk deneyimimize geçelim! `Alias` komutunu kullanarak kendi komutumuzu oluşturacağız. Ancak başlamadan önce, küçük bir komut satırı hilesi açıklamamız gerekiyor. Her komutu noktalı virgülle ayırarak bir satıra birden fazla komut yazmak mümkündür. Şu şekilde çalışır:

```
komut1; komut2; komut3...
```

İşte kullanacağımız örnek:

```
[me@linuxbox ~]$ cd /usr; ls; cd -
bin oyunlar dahil lib yerel sbin paylaş src
/home/me
[me@linuxbox ~]$
```

Gördüğümüz gibi, bir satırda üç komutu birleştirdik. Önce dizini `/usr` olarak değiştirdik, ardından dizini listeledik ve son olarak orijinal dizine geri döndük (`cd komutunu` kullanarak). `-'`) böylece başladığımız yere geri dönüyoruz. Şimdi bu diziyi yeni bir komut haline getirelim us -

ing `alias`. Yapmamız gereken ilk şey, yeni komutumuz için bir isim bulmak. `test'i` deneyelim. Bunu yapmadan önce, `test` isminin zaten kullanılıp kullanılmadığını öğrenmek iyi bir fikir olabilir. Bunu öğrenmek için, `type` komutunu tekrar kullanabiliriz:

```
[me@linuxbox ~]$ type test
test bir kabuk yerleşik
komutudur
```

Oops! `test` adı zaten alınmış. `foo'yu` deneyelim:

```
[me@linuxbox ~]$ type foo
bash: type: foo: bulunamadı
```

Harika! `foo` alınmamış. Öyleyse takma adımızı oluşturalım:

```
[me@linuxbox ~]$ alias foo='cd /usr; ls; cd -'
```

Burada gösterilen bu komutun yapısına dikkat edin:

```
alias name='string'
```

`alias` komutundan sonra, alias'a bir isim veririz, hemen ardından (boşluk bırakmadan) eşittir işareti gelir, ardından da isme atanacak anlamı içeren tırnak içine alınmış bir dize gelir. Alias'ımızı tanımladıktan sonra, kabuğun bir komut beklediği her yerde kullanabiliriz. Deneyelim:

```
[me@linuxbox ~]$ foo
bin games include lib local sbin share src
/home/me
[me@linuxbox ~]$
```

Alias'ımızı görmek için `type` komutunu tekrar kullanabiliriz:

```
[me@linuxbox ~]$ type foo
foo, `cd /usr; ls; cd -'
```

Bir takma adı kaldırmak için `unalias` komutu şu şekilde kullanılır:

```
[me@linuxbox ~]$ unalias foo
[me@linuxbox ~]$ type foo bash:
type: foo: bulunamadı
```

Alias'ımızı mevcut bir komut adıyla adlandırmaktan kasıtlı olarak kaçındık, ancak bunu yapmak alışılmadık bir durum değildir. Bu genellikle, yaygın olarak kullanılan bir komutun her çağrılışında yaygın olarak istenen bir seçeneği uygulamak için yapılır. Örneğin, daha önce `ls` komutunun renk desteği eklemek için sıklıkla alias olarak kullanıldığını görmüştük.

```
[me@linuxbox ~]$ type ls
ls, `ls --color=auto' olarak takma adlandırılmıştır
```

Ortamda tanımlanmış tüm takma adları görmek için, argüman olmadan `alias` komutunu kullanın. Aşağıda, Fedora sisteminde varsayılan olarak tanımlanmış bazı takma adlar verilmiştir. Bunların ne işe yaradığını bulmaya çalışın.

```
[me@linuxbox ~]$ alias
alias l.='ls -d .* --color=auto'
alias ll='ls -l --color=auto' alias
ls='ls --color=auto'
```

Komut satırında takma ad tanımlamada küçük bir sorun vardır. Kabuk oturumumuz sona erdiğinde bunlar kaybolur. 11. bölümde, her oturum açtığımızda ortamı oluşturan dosyalara kendi takma adlarımızı nasıl ekleyeceğimizi göreceğiz, ama şimdilik, kabuk programlama dünyasına küçük de olsa ilk adımımızı attığımızın keyfini çıkarın!

Özet

Komutların belgelerini nasıl bulacağımızı öğrendiğimize göre, şimdiye kadar karşılaştığımız tüm komutların belgelerini arayın. Kullanılabilir ek seçenekleri inceleyin ve deneyin!

Daha Fazla Okuma

Linux ve komut satırı için birçok çevrimiçi dokümantasyon kaynağı bulunmaktadır. İşte en iyilerinden bazıları:

- *Bash Referans Kılavuzu*, `bash` kabuğu için bir referans kılavuzudur. Hala bir referans

http://www.gnu.org/software/bash/manual/bashref.html

- *Bash SSS*, `bash` ile ilgili sık sorulan soruların yanıtlarını içerir. Bu liste orta ve ileri düzey kullanıcılar için hazırlanmıştır, ancak birçok yararlı bilgi içerir. http://mywiki.wooledge.org/BashFAQ
- GNU Projesi, Linux komut satırı deneyiminin temelini oluşturan programları için kapsamlı belgeler sağlar. Tam listeyi burada görebilirsiniz: http://www.gnu.org/manual/manual.html
- Wikipedia'da man sayfaları hakkında ilginç bir makale bulunmaktadır: http://en.wikipedia.org/wiki/Man_page

6 – Yönlendirme

Bu bölümde, komut satırının belki de en havalı özelliğini ortaya çıkaracağız. Buna *I/O yeniden yönlendirme* denir. I/O, *giriş/çıkış* anlamına gelir ve bu özellik sayesinde komutların giriş ve çıkışlarını dosyalara yeniden yönlendirebilir, ayrıca birden fazla komutu birbirine bağlayarak güçlü komut *ardışık düzenleri* oluşturabiliriz. Bu özelliği göstermek için aşağıdaki komutları tanıtacağız:

- `cat` – Dosyaları birleştirir
- `sort` – Metin satırlarını sıralama
- `uniq` – Tekrarlanan satırları raporlama veya atlama
- `grep` – Desenle eşleşen satırları yazdır
- `wc` – Her dosya için satır sonu, kelime ve bayt sayısını yazdır
- `head` – Dosyanın ilk kısmını çıkarır
- `tail` – Dosyanın son kısmını çıkarır
- `tee` – Standart girişten okur ve standart çıkışa ve dosyalara yazar

Standart Giriş, Çıktı ve Hata

Şimdiye kadar kullandığımız programların çoğu bir tür çıktı üretir. Bu çıktı genellikle iki türden oluşur:

- Programın sonuçları, yani programın üretmek üzere tasarlandığı veriler
- Programın nasıl çalıştığını bize bildiren durum ve hata mesajları

`ls` gibi bir komuta bakarsak, sonuçlarını ve hata mesajlarını ekranda görüntülediğini görebiliriz.

Unix'in "her şey bir dosyadır" temasına uygun olarak, `ls` gibi programlar sonuçlarını *standart çıktı* (genellikle *stdout* olarak ifade edilir) adlı özel bir dosyaya ve durum mesajlarını *standart hata* (*stderr*) adlı başka bir dosyaya gönderir. Varsayılan olarak, hem standart çıktı hem de standart hata ekrana bağlanır ve bir disk dosyasına kaydedilmez.

Ayrıca, birçok program varsayılan olarak klavyeye bağlı olan *standart giriş* (*stdin*) adlı bir özellikten girdi alır.

G/Ç yeniden yönlendirme, çıktının gittiği yeri ve girdinin geldiği yeri değiştirmemizi sağlar. Normalde, çıktı ekrana gider ve girdi klavyeden gelir, ancak G/Ç yeniden yönlendirme ile bunu değiştirebiliriz.

Standart Çıktıyı Yeniden Yönlendirme

G/Ç yeniden yönlendirmesi, standart çıktının nereye gideceğini yeniden tanımlamamızı sağlar. Standart çıktıyı ekrana değil başka bir dosyaya yeniden yönlendirmek için, `>` yeniden yönlendirme operatörünü ve ardından dosya adını kullanırız. Bunu neden yapmak isteyebiliriz? Bir komutun çıktısını bir dosyada saklamak genellikle yararlıdır. Örneğin, kabuğa `ls` komutunun çıktısını ekrana değil `ls-output.txt` dosyasına göndermesini söyleyebiliriz:

```
[me@linuxbox ~]$ ls -l /usr/bin > ls-output.txt
```

Burada, `/usr/bin` dizinini uzun bir liste halinde oluşturduk ve sonuçları `ls-output.txt` dosyasına gönderdik. Komutun yeniden yönlendirilen çıktısını inceleyelim:

```
[me@linuxbox ~]$ ls -l ls-output.txt
-rw-rw-r-- 1 me    me    167878 2025-02-01 15:07 ls-output.txt
```

Güzel — hoş, büyük bir metin dosyası. Dosyayı `less` ile incelersek, `ls-output.txt dosyasının` gerçekten `ls` komutumuzun sonuçlarını içerdiğini göreceğiz.

```
[me@linuxbox ~]$ less ls-output.txt
```

Şimdi, yönlendirme testimizi tekrarlayalım, ancak bu sefer bir değişiklik yapalım. Dizinin adını, var olmayan bir adla değiştireceğiz:

```
[me@linuxbox ~]$ ls -l /bin/usr > ls-output.txt
ls: /bin/usr'ye erişilemiyor: Böyle bir dosya veya dizin yok
```

Bir hata mesajı aldık. Bu mantıklı, çünkü var olmayan `/bin/usr` dizinini belirttik, ama hata mesajı neden `ls-output.txt` dosyasına yönlendirilmedi de ekranda görüntülendi? Cevap, `ls` programının hata mesajlarını standart çıktıya göndermediği içindir. Bunun yerine, çoğu iyi yazılmış Unix programı gibi,

hata mesajlarını *standart hataya* (*stderr*) gönderir. Yalnızca standart çıktıyı yönlendirdiğimiz ve standart hatayı yönlendirmediğimiz için, hata mesajı yine de ekrana gönderildi. Standart hatayı nasıl yönlendireceğimizi birazdan göreceğiz, ama önce çıktı dosyamızda ne olduğunu inceleyelim:

```
[me@linuxbox ~]$ ls -l ls-output.txt
-rw-rw-r-- 1 me    me    0 2025-02-01 15:08 ls-output.txt
```

Dosyanın uzunluğu artık sıfır! Bunun nedeni, “>” yönlendirme operatörüyle çıktıyı yönlendirdiğimizde, hedef dosyanın her zaman baştan yazılmasıdır. `ls` komutumuz hiçbir sonuç üretmediğinden ve yalnızca bir hata mesajı verdiğinden, yeniden yönlendirme işlemi dosyayı yeniden yazmaya başladı ve ardından hata nedeniyle durdu, bu da dosyanın kesilmesine neden oldu. Aslında, bir dosyayı gerçekten kesmemiz (veya yeni, boş bir dosya oluşturmamız) gerekirse, şu gibi bir hile kullanabiliriz:

```
[me@linuxbox ~]$ > ls-output.txt
```

Öncesinde herhangi bir komut olmadan yönlendirme operatörünü kullanmak, mevcut bir dosyayı keser veya yeni, boş bir dosya oluşturur.

Peki, dosyayı baştan yazmak yerine yönlendirilen çıktıyı dosyaya nasıl ekleyebiliriz? Bunun için >> yönlendirme operatörünü şu şekilde kullanırız:

```
[me@linuxbox ~]$ ls -l /usr/bin >> ls-output.txt
```

>> operatörünü kullanmak, çıktının dosyaya eklenmesine neden olur. Dosya henüz mevcut değilse, > operatörü kullanılmış gibi oluşturulur. Bir komutu tekrarlayıp çıktısını bir dosyaya ekleyerek bunu test edelim:

```
[me@linuxbox ~]$ ls -l /usr/bin >> ls-output.txt
[me@linuxbox ~]$ ls -l /usr/bin >> ls-output.txt
[me@linuxbox ~]$ ls -l /usr/bin >> ls-output.txt
[me@linuxbox ~]$ ls -l ls-output.txt
-rw-rw-r-- 1 me    me    503634 2025-02-01 15:45 ls-output.txt
```

`ls` komutunu üç kez tekrarladık ve sonuç olarak üç kat daha büyük bir çıktı dosyası elde ettik.

Grup Komutları

Bir dizi komutu çalıştırmak ve sonuçları bir günlük dosyasına göndermek istediğimiz bir durum düşünelim. Bildiklerimizle bunu şu şekilde yapabiliriz:

```
[me@linuxbox ~]$ command1 > logfile.txt
[me@linuxbox ~]$ command2 >> logfile.txt
[me@linuxbox ~]$ command3 >> logfile.txt
```

Bu dizideki ilk komut `logfile.txt` adlı bir dosya oluşturur/kısaltır ve sonraki her komut çıktısını bu dosyaya ekler. Bu teknik işe yarar, ancak çok fazla gereksiz yazım vardır. Daha iyi bir yol olmalı.

Önceki bölümde gördüğümüz gibi, birden fazla komutu tek bir satıra bu şekilde yazabiliriz:

```
[me@linuxbox ~]$ command1; command2; command3
```

Böylece tüm komutlarımızı ve yönlendirmelerimizi tek bir satıra yerleştirebiliriz:

```
[me@linuxbox ~]$ command1 > logfile.txt; command2 >> logfile.txt; command3
>> logfile.txt
```

Peki, bu diziyi tek bir çıktı akışına sahip tek bir varlık olarak ele alabilseydik ne olurdu? Bunu, bir *grup komutu* oluşturarak yapabiliriz. Bunun için, dizimizi köşeli parantez karakterleriyle çevreleriz:

```
[me@linuxbox ~]$ { command1; command2; command3; } > logfile.txt
```

Dizimizi köşeli parantezlerle çevrelediğimizde, kabuk bunu yeniden yönlendirme açısından tek bir komut olarak kabul eder. Kabuğun köşeli parantezlerin etrafında boşluk gerektirdiğini ve dizideki son komutun noktalı virgül veya satır sonu ile sonlandırılması gerektiğini unutmayın.

Standart Hatayı Yeniden Yönlendirme

Standart hatayı yeniden yönlendirmek, özel bir yeniden yönlendirme operatörünün kolaylığına sahip değildir. Standart hatayı yeniden yönlendirmek için, *dosya tanımlayıcısına* başvurmamız gerekir. Bir program, numaralandırılmış birkaç dosya akışından herhangi birinde çıktı üretebilir. Bu dosya akışlarının ilk üçüne standart giriş, çıkış ve hata olarak atıfta bulunmuş olsak da, kabuk bunları dahili olarak sırasıyla 0, 1 ve 2 dosya tanımlayıcıları olarak referans alır.

Kabuk, dosya tanımlayıcı numarasını kullanarak dosyaları yeniden yönlendirmek için bir notasyon sağlar. Standart hata, dosya tanımlayıcı numarası 2 ile aynı olduğundan, bu notasyonla standart hatayı yeniden yönlendirebiliriz:

```
[me@linuxbox ~]$ ls -l /bin/usr 2> ls-error.txt
```

Dosya tanımlayıcı "2", standart hatayı `ls-error.txt` dosyasına yönlendirmek için yönlendirme operatörünün hemen önüne yerleştirilir.

Standart Çıktı ve Standart Hatayı Tek Bir Dosyasına Yönlendirme

Bir komutun tüm çıktısını tek bir dosyaya kaydetmek isteyebileceğimiz durumlar vardır. Bunu yapmak için, hem standart çıktıyı hem de standart hatayı aynı anda yeniden yönlendirmeliyiz. Bunu yapmanın iki yolu vardır. Burada, eski kabuk sürümlerinde çalışan geleneksel yol gösterilmektedir:

```
[me@linuxbox ~]$ ls -l /bin/usr > ls-output.txt 2>&1
```

Bu yöntemi kullanarak iki yönlendirme gerçekleştiriyoruz. Önce standart çıktıyı `ls-output.txt` dosyasına yönlendiriyoruz, ardından `2>&1` notasyonunu kullanarak dosya tanımlayıcı 2'yi (standart hata) dosya tanımlayıcı 1'e (standart çıktı) yönlendiriyoruz.

Yönlendirmelerin sırasının önemli olduğunu unutmayın. Standart hatanın yönlendirilmesi, her zaman standart çıktının yönlendirilmesinden *sonra* gerçekleşmelidir, aksi takdirde çalışmaz. Aşağıdaki örnek, standart hatayı `ls-output.txt` dosyasına yönlendirir:

```
>ls-output.txt 2>&1
```

Ancak, sıra aşağıdaki gibi değiştirilirse, standart hata ekrana yönlendirilir.

```
2>&1 >ls-output.txt
```

`Bash'ın` son sürümleri, burada gösterilen bu birleşik yönlendirmeyi gerçekleştirmek için ikinci, daha basitleştirilmiş bir yöntem sunar:

```
[me@linuxbox ~]$ ls -l /bin/usr &> ls-output.txt
```

Bu örnekte, hem standart çıktıyı hem de standart hatayı `ls-output.txt` dosyasına yönlendirmek için tek bir `&>` notasyonu kullanıyoruz. Standart çıktı ve standart hata akışlarını tek bir dosyaya şu şekilde ekleyebiliriz:

```
[me@linuxbox ~]$ ls -l /bin/usr &>> ls-output.txt
```

'ın İstenmeyen Çıktılarını Ortadan Kaldırma

Bazen "sessizlik altındır" ve bir komuttan çıktı istemiyoruz, sadece onu atmak istiyoruz. Bu özellikle hata ve durum mesajları için geçerlidir. Sistem, çıktıyı "/dev/null" adlı özel bir dosyaya yönlendirerek bunu yapmanın bir yolunu sunar. Bu dosya, genellikle *bit kovası* olarak adlandırılan ve girişi kabul eden ancak bununla hiçbir şey yapmayan bir sistem aygıtıdır. Bir komuttan gelen hata mesajlarını bastırmak için şunu yaparız:

```
[me@linuxbox ~]$ ls -l /bin/usr 2> /dev/null
```

`/dev/null` Unix Kültüründe

Bit kovası eski bir Unix kavramıdır ve evrenselliği nedeniyle Unix kültürünün birçok alanında karşımıza çıkar. Birisi yorumlarınızı `/dev/null'a` göndereceğini söylediğinde, artık bunun ne anlama geldiğini biliyorsunuz. Daha fazla örnek için, Wikipedia'daki `/dev/null` maddesine bakabilirsiniz.

Standart Girişi Yeniden Yönlendirme

Şimdiye kadar, standart girişi kullanan herhangi bir komutla karşılaşmadık (aslında karşılaştık, ama bu sürprizi biraz sonra açıklayacağız), bu yüzden bir tane tanıtmamız gerekiyor.

`cat` – Dosyaları Birleştirir

`cat` komutu bir veya daha fazla dosyayı okur ve bunları standart çıktıya şu şekilde kopyalar:

```
cat [dosya...]
```

Çoğu durumda, `cat` komutunu DOS'taki `TYPE` komutuna benzer olarak düşünebiliriz.

Bu komutu, sayfalandırma yapmadan dosyaları görüntülemek için kullanabiliriz. Örneğin, aşağıdaki komut `ls-output.txt` dosyasının içeriğini görüntüler:

```
[me@linuxbox ~]$ cat ls-output.txt
```

`cat` genellikle kısa metin dosyalarını görüntülemek için kullanılır. `cat` birden fazla dosyayı argüman olarak kabul edebildiğinden, dosyaları birleştirmek için de kullanılabilir. Diyelim ki, birden fazla parçaya bölünmüş büyük bir dosya indirdik (multimedya dosyaları Usenet'te genellikle bu şekilde bölünür) ve bunları tekrar birleştirmek istiyoruz. Dosyaların isimleri şöyle olsun:

`movie.mpeg.001 movie.mpeg.002 ... movie.mpeg.099`

bu komutla bunları şu şekilde birleştirebiliriz:

```
cat movie.mpeg.0* > movie.mpeg
```

Joker karakterler her zaman sıralı olarak genişlediğinden, argümanlar doğru sırayla düzenlenecektir.

Bu çok güzel, ama bunun standart girişle ne ilgisi var? Henüz hiçbir ilgisi yok, ama başka bir şey deneyelim. `Cat komutunu` argüman olmadan girersek ne olur?

```
[me@linuxbox ~]$ cat
```

Hiçbir şey olmaz, sanki takılmış gibi orada kalır. Öyle görünebilir, ama aslında tam olarak yapması gerekeni yapmaktadır.

`cat` komutuna herhangi bir argüman verilmezse, standart girdiden okur ve standart girdi varsayılan olarak klavyeye bağlı olduğundan, bizim bir şey yazmamızı bekler! Aşağıdaki metni ekleyip `ENTER tuşuna` basmayı deneyin:

```
[me@linuxbox ~]$ cat
Hızlı kahverengi tilki tembel köpeğin üzerinden atlar.
```

Ardından, `Ctrl-d` tuşlarına basın (yani, Ctrl tuşunu basılı tutun ve "d" tuşuna basın) ve `cat` komutuna standart girdide *dosya sonuna* (EOF) ulaştığını bildirin:

```
[me@linuxbox ~]$ cat
Hızlı kahverengi tilki tembel köpeğin üzerinden atlar.
```

```
Çabuk kahverengi tilki tembel köpeğin üzerinden atlar.
```

Dosya adı argümanları olmadığında, `cat` standart girişi standart çıktıya kopyalar, böylece metin satırımızın tekrarlandığını görürüz. Bu davranışı kısa metin dosyaları oluşturmak için kullanabiliriz. Örneğimizdeki metni içeren `lazy_dog.txt` adlı bir dosya oluşturmak istediğimizi varsayalım. Bunu şu şekilde yapabiliriz:

```
[me@linuxbox ~]$ cat > lazy_dog.txt
Hızlı kahverengi tilki tembel köpeğin üzerinden atlar.
```

Komutu ve ardından dosyaya yerleştirmek istediğimiz metni yazın. Sonunda `Ctrl-d` tuşlarına basmayı unutmayın. Komut satırını kullanarak dünyanın en aptal kelime işlemcisini oluşturduk! Sonuçları görmek için `cat komutunu` kullanarak dosyayı tekrar stdout'a kopyalayabiliriz.

```
[me@linuxbox ~]$ cat lazy_dog.txt
Hızlı kahverengi tilki tembel köpeğin üzerinden atlar.
```

Artık `cat'in` dosya adı argümanlarına ek olarak standart girişi nasıl kabul ettiğini bildiğimize göre, standart girişi yeniden yönlendirmeyi deneyelim.

```
[me@linuxbox ~]$ cat < lazy_dog.txt
Hızlı kahverengi tilki tembel köpeğin üzerinden atlar.
```

`<` yeniden yönlendirme operatörünü kullanarak, standart girişin kaynağını klavyeden `lazy_dog.txt` dosyasına değiştiriyoruz. Sonucun, tek bir dosya adı argümanı geçirmekle aynı olduğunu görüyoruz. Bu, dosya adı argümanı geçirmekle karşılaştırıldığında özellikle yararlı değildir, ancak bir dosyayı standart giriş kaynağı olarak kullanmayı göstermek için yararlıdır. Diğer komutlar standart girişi daha iyi kullanır, bunu birazdan göreceğiz.

Devam etmeden önce, `cat` komutunun man sayfasını inceleyin, çünkü birkaç ilginç seçeneği vardır.

Boru hatları

Komutların standart girdiden veri okuma ve standart çıktıya gönderme yeteneği, *boru hatları* adı verilen bir kabuk özelliği tarafından kullanılır. Boru operatörü `|` (dikey çubuk) kullanılarak, bir komutun standart çıktısı başka bir komutun standart girdisine *aktarılabilir*.

```
komut1 | komut2
```

Bunu tam olarak göstermek için bazı komutlara ihtiyacımız olacak. Standart girişi kabul eden bir komut olduğunu söylediğimizi hatırlıyor musunuz? Bu `komut less'tir`. `Less komutunu` kullanarak, sonuçlarını standart çıktıya gönderen herhangi bir komutun çıktısını sayfa sayfa görüntüleyebiliriz:

```
[me@linuxbox ~]$ ls -l /usr/bin | less
```

Bu çok kullanışlıdır! Bu tekniği kullanarak, standart çıktı üreten herhangi bir komutun çıktısını rahatlıkla inceleyebiliriz.

> ve | Arasındaki Fark

İlk bakışta, boru hattı operatörü | ile yönlendirme operatörü > tarafından gerçekleştirilen yönlendirmeyi anlamak zor olabilir. Basitçe söylemek gerekirse, yönlendirme operatörü bir komutu bir dosyaya bağlarken, boru hattı operatörü bir komutun çıktısını ikinci bir komutun girdisine bağlar.

komut1 > dosya1 komut1 |

komut2

Birçok kişi, boru hatları hakkında bilgi edinirken "ne olacağını görmek için" aşağıdakileri dener:

komut1 > komut2

Cevap: bazen gerçekten kötü şeyler olur.

İşte Linux tabanlı bir sunucu cihazını yöneten bir okuyucunun gönderdiği gerçek bir örnek. Süper kullanıcı olarak şunu yaptı:

```
# cd /usr/bin
# ls > less
```

İlk komut onu çoğu programın depolandığı dizine götürdü ve ikinci komut kabuğa `less` dosyasını ls komutunun çıktısıyla üzerine yazmasını söyledi. `/usr/bin` dizini zaten `less` adlı bir dosya (`less` programı) içerdiğinden, ikinci komut `less` program dosyasını `ls'nin` metniyle üzerine yazdı ve böylece sistemindeki `less` programını yok etti.

Buradan çıkarılacak ders, > yeniden yönlendirme operatörünün dosyaları sessizce oluşturduğu veya üzerine yazdığıdır, bu nedenle ona çok dikkatli davranmanız gerekir.

Filtreler

Boru hatları genellikle veriler üzerinde karmaşık işlemler gerçekleştirmek için kullanılır. Birkaç komutu bir boru hattında bir araya getirmek mümkündür. Bu şekilde kullanılan komutlar genellikle *filtreler* olarak adlandırılır. Filtreler girişi alır, bir şekilde değiştirir ve ardından çıktısını verir. Deneyeceğimiz ilk filtre `sort'tur`. `/bin` ve `/usr/bin` dizinlerindeki tüm çalıştırılabilir programların birleşik bir listesini oluşturmak, bunları sıralı bir düzende yerleştirmek ve sonuç listesini görüntülemek istediğimizi düşünün:

```
[me@linuxbox ~]$ ls /bin /usr/bin | sort | less
```

İki dizin (`/bin` ve `/usr/bin`) belirlediğimiz için, `ls` komutunun çıktısı her dizin için birer tane olmak üzere iki sıralı listeden oluşacaktı. Pipelinessize `sort` komutunu ekleyerek, verileri tek bir sıralı liste oluşturacak şekilde değiştirdik.

`sort`, birçok özellik ve seçeneğe sahip güçlü bir komuttur. Bunları 20. bölümde ayrıntılı olarak ele alacağız.

`uniq` - Tekrarlanan Satırları Raporla veya Atla ( )

`uniq` komutu genellikle `sort` ile birlikte kullanılır. `uniq`, standart girişten veya tek bir dosya adı argümanından sıralanmış bir veri listesini kabul eder (ayrıntılar için `uniq` man sayfasına bakın) ve varsayılan olarak listeden tüm yinelenenleri kaldırır. Bu nedenle, listemizde yinelenenler olmadığından emin olmak için (yani, hem `/bin` hem de `/usr/bin` dizinlerinde görünen aynı adlı programlar), `uniq komutunu` boru hattımıza ekleyeceğiz.

```
[me@linuxbox ~]$ ls /bin /usr/bin | sort | uniq | less
```

Bu örnekte, sort komutunun çıktısından tüm yinelenenleri kaldırmak için `uniq` kullanıyoruz. Bunun yerine yinelenenlerin listesini görmek istersek, `uniq'e` şu şekilde "-d" seçeneğini ekleriz:

```
[me@linuxbox ~]$ ls /bin /usr/bin | sort | uniq -d | less
```

wc – Satır, Kelime ve Bayt ı Sayımlarını Yazdır

wc (kelime sayısı) komutu, dosyalarda bulunan satır, kelime ve bayt sayısını görüntülemek için kullanılır. İşte bir örnek:

```
[me@linuxbox ~]$ wc ls-output.txt
 7902 64566 503634 ls-output.txt
```

Bu durumda, ls-out-put.txt dosyasında bulunan satır, kelime ve bayt sayılarını üç rakam olarak yazdırır. Önceki komutlarımızda olduğu gibi, komut satırı argümanları olmadan çalıştırıldığında, wc standart girişi kabul eder. "-l" seçeneği, çıktısını yalnızca satırları raporlamakla sınırlar. Bunu bir boru hattına eklemek, sayım yapmak için kullanışlı bir yoldur. Sıralanmış listemizde bulunan öğe sayısını görmek için şunu yapabiliriz:

```
[me@linuxbox ~]$ ls /bin /usr/bin | sort | uniq | wc -l 2728
```

grep – Deseniyle Eşleşen Satırları Yazdır

grep, dosyalardaki metin kalıplarını bulmak için kullanılan güçlü bir programdır. Şu şekilde kullanılır:

```
grep desen [dosya...]
```

grep, dosyada bir "desen" bulduğunda, onu içeren satırları yazdırır. grep'in eşleştirebileceği desenler çok karmaşık olabilir, ancak şimdilik basit metin eşleştirmelerine odaklanacağız. İleri düzey desenleri, yani *düzenli ifadeleri* 19. bölümde ele alacağız.

Diyelim ki, program listemizdeki adında zip kelimesi geçen tüm dosyaları bulmak istiyoruz. Böyle bir arama, sistemimizdeki dosya sıkıştırma ile ilgili bazı programlar hakkında bize fikir verebilir. Bunu şu şekilde yapabiliriz:

```
[me@linuxbox ~]$ ls /bin /usr/bin | sort | uniq | grep zip
bunzip2
bzip2
gunzip
gzip
unzip
```

```
zip
zipcloak
zipgrep
zipinfo
zipnote
zipsplit
```

İşte `grep` için birkaç kullanışlı seçenek:

- `-i`, `grep'in` arama yaparken büyük/küçük harfleri yok saymasını sağlar (normalde aramalar büyük/küçük harfe duyarlıdır)
- `-l`, `grep'in` yalnızca desene uyan metin içeren dosyaların adlarını görüntülemesini sağlar.
- `-v`, `grep'in` yalnızca desene uymayan satırları yazdırmasını sağlar.
- `-w`, `grep'in` yalnızca tam kelimeleri eşleştirmesini sağlar.

`head` / `tail` – Dosyaların İlk / Son Kısmını Yazdır

Bazen bir komutun tüm çıktısını istemiyoruz. Sadece ilk birkaç satırı veya son birkaç satırı isteyebiliriz. `head` komutu bir dosyanın ilk 10 satırını, `tail` komutu ise son 10 satırını yazdırır. Her iki komut da varsayılan olarak 10 satırlık metin yazdırır, ancak bu `-n` seçeneği ile ayarlanabilir.

```
[me@linuxbox ~]$ head -n 5 ls-output.txt
toplam 343496
-rwxr-xr-x 1 root root         31316 2025-12-05 08:58 [
-rwxr-xr-x 1 root root          8240 2025-12-09 13:39 411toppm
-rwxr-xr-x 1 root root        111276 2025-11-26 14:27 a2p
-rwxr-xr-x 1 root root         25368 2024-10-06 20:16 a52dec
[me@linuxbox ~]$ tail -n 5 ls-output.txt
-rwxr-xr-x 1 kök kök            5234 2025-06-27 10:56 znew
-rwxr-xr-x 1 root root           691 2023-09-10 04:21 zonetab2pot.py
-rw-r--r-- 1 root root           930 2025-11-01 12:23 zonetab2pot.pyc
-rw-r--r-- 1 root root           930 2025-11-01 12:23 zonetab2pot.pyo
lrwxrwxrwx 1 root root             6 2024-01-31 05:22 zsoelim -> soelim
```

Bu komutlar boru hatlarında da kullanılabilir:

```
[me@linuxbox ~]$ ls /usr/bin | tail -n 5
```

```
znew
zonetab2pot.py
zonetab2pot.pyc
zonetab2pot.pyo
zsoelim
```

`Head` ve `tail` ile birlikte `-n` seçeneğini kullanarak, bir dosyanın ortasından bir alıntı kesebiliriz. Beş satırlık bir üstbilgi ve beş satırlık bir altbilgi içeren bir metin dosyamız olduğunu ve bu üstbilgi ve altbilgiyi kaldırarak ortadaki verileri içeren "iyi" kısmı bırakmak istediğimizi varsayalım. Bunun için şu yöntemi kullanabiliriz:

```
[me@linuxbox ~]$ head -n -5 text_header_footer.txt | tail -n +6 > text.txt
```

`Head` ile birlikte kullanılan `-n` seçeneği, son *n* satır hariç tüm satırların çıktısını almamızı sağlayan negatif bir değer kabul eder. Benzer şekilde, `tail` ile birlikte kullanılan `-n` seçeneği, n'inci satırdan başlayan tüm satırların çıktısını almamızı sağlayan artı işareti kabul eder.

`tail` ayrıca bir dosyanın içeriğini gerçek zamanlı olarak takip etmemizi sağlayan bir seçeneğe de sahiptir. Bu, günlük dosyalarının yazılma sürecini izlemek için kullanışlıdır. Aşağıdaki örnekte, `/var/log` içindeki `messages` dosyasına (veya `messages` dosyası yoksa `/var/log/sys-log` dosyasına) bakacağız. Günlük dosyaları güvenlik bilgileri içerebileceğinden, bazı Linux dağıtımlarında bunu yapmak için süper kullanıcı ayrıcalıkları gerekebilir:

```
[me@linuxbox ~]$ tail -f /var/log/messages
8 Şubat 13:40:05 twin4 dhclient: 192.168.1.1'den DHCPACK
8 Şubat 13:40:05 twin4 dhclient: 192.168.1.4'e bağlı -- 1652 saniye
içinde yenilenecek.
8 Şubat 13:55:32 twin4 mountd[3953]: /var/NFSv4/musicbox hem
192.168.1.0/24 hem de twin7.localdomain'e
192.168.1.0/24,twin7.localdomain adreslerinde dışa aktarıldı
8 Şubat 14:07:37 twin4 dhclient: eth0 üzerinde 192.168.1.1 bağlantı
noktası 67'ye DHCPREQUEST
8 Şubat 14:07:37 twin4 dhclient: 192.168.1.1'den DHCPACK
8 Şubat 14:07:37 twin4 dhclient: 192.168.1.4'e bağlı -- 1771 saniye
içinde yenilenecek.
8 Şubat 14:09:56 twin4 smartd[3468]: Aygıt: /dev/hda, SMART Ön Arıza
Özniteliği: 8 Seek_Time_Performance 237'den 236'ya değiştirildi 8 Şubat
14:10:37 twin4 mountd[3953]: /var/NFSv4/musicbox hem 192.168.1.0/24 hem
de twin7.localdomain'e dışa aktarıldı
```

```
192.168.1.0/24,twin7.localdomain
8 Şubat 14:25:07 twin4 sshd(pam_unix)[29234]: (uid=0) tarafından me
kullanıcısı için oturum açıldı
8 Şubat 14:25:36 twin4 su(pam_unix)[29279]: root kullanıcısı için oturum
açıldı (uid=500)
```

`-f` seçeneği kullanıldığında, `tail` dosyayı izlemeye devam eder ve yeni satırlar eklendiğinde bunlar hemen ekranda görünür. Bu, `Ctrl-c` tuşuna basana kadar devam eder.

`tee` – Stdin'den okur ve Stdout ve dosyalarına yazdırır

Tesisat metaforumuza uygun olarak, Linux, borumuzda bir "tee" bağlantısı oluşturan `tee` adlı bir komut sağlar. `Tee` programı standart girişi okur ve hem standart çıktıya (verilerin boru hattında ilerlemesine izin verir) hem de bir veya daha fazla dosyaya kopyalar. Bu, işlemenin ara aşamasında boru hattının içeriğini yakalamak için kullanışlıdır. Burada, önceki örneklerimizden birini tekrarlıyoruz, ancak bu sefer `grep` boru hattının içeriğini filtrelemeden önce tüm dizin listesini `ls.txt` dosyasına yakalamak için `tee komutunu da` ekliyoruz:

```
[me@linuxbox ~]$ ls /usr/bin | tee ls.txt | grep zip bunzip2
bzip2
gunzip
gzip
unzip
zip
zipcloak
zipgrep
zipinfo
zipnote
zipsplit
```

Özet

Her zaman olduğu gibi, bu bölümde ele aldığımız her bir komutun belgelerini inceleyin. Biz sadece en temel kullanımlarını gördük. Hepsinin bir dizi ilginç seçeneği vardır. Linux deneyimimiz arttıkça, komut satırının yönlendirme özelliğinin özel sorunları çözmek için son derece yararlı olduğunu göreceğiz. Standart giriş ve çıkışı kullanan birçok komut vardır ve neredeyse tüm komut satırı programları bilgilendirici mesajlarını görüntülemek için standart hatayı kullanır.

Linux, Hayal Gücüyle İlgilidir ( )

Windows ve Linux arasındaki farkı açıklamam istendiğinde, genellikle bir oyuncak benzetmesi kullanırım.

Windows, Game Boy gibidir. Mağazaya gidip kutusu içinde yepyeni bir tane satın alırsınız. Eve götürür, açar ve oynarsınız. Güzel grafikler, sevimli sesler. Ancak bir süre sonra, oyunla birlikte gelen oyundan sıkılırsınız, bu yüzden mağazaya geri dönüp başka bir tane satın alırsınız. Bu döngü tekrar tekrar devam eder. Sonunda mağazaya geri dönüp tezgâhtaki kişiye "Bunu yapan bir oyun istiyorum!" dersiniz, ancak böyle bir oyun olmadığı, çünkü "pazar talebi" olmadığı söylenir. Sonra "Ama sadece bu tek şeyi değiştirmem gerekiyor!" dersiniz. Tezgâhtaki kişi bunu değiştiremeyeceğinizi söyler. Oyunlar kartuşlarında mühürlenmiştir. Oyuncaklarınızın, başkalarının sizin için gerekli olduğuna karar verdiği oyunlarla sınırlı olduğunu fark edersiniz.

Linux ise dünyanın en büyük Erector Set'i gibidir. Açtığınızda, içinde sadece devasa bir parça koleksiyonu vardır. Çok sayıda çelik destek, vida, somun, dişli, kasnak, motor ve ne yapabileceğinize dair birkaç öneri vardır. Siz de onunla oynamaya başlarsınız. Önerilerden birini yaparsınız, sonra bir başkasını. Bir süre sonra, ne yapacağınız konusunda kendi fikirleriniz olduğunu keşfedersiniz. İhtiyacınız olan her şeye zaten sahip olduğunuz için mağazaya geri dönmenize gerek kalmaz. Erector Set, hayal gücünüzün şeklini alır. İstediğinizi yapar.

Oyuncak seçimi elbette kişisel bir tercihtir, peki hangi oyuncak sizi daha çok memnun eder?

7 – Shell'in Gördüğü Şekilde Dünyayı Görmek

Bu bölümde, ENTER tuşuna bastığımızda komut satırında meydana gelen bazı "sihirli" olaylara bakacağız. Kabuğun birkaç ilginç ve karmaşık özelliğini inceleyeceğiz, ancak bunu tek bir yeni komutla yapacağız.

- echo – Bir satır metin görüntüleyin

Genişletme

Bir komut yazıp ENTER tuşuna her bastığımızda, bash komutumuzu yerine getirmeden önce metin üzerinde birkaç değiştirme işlemi gerçekleştirir. Basit bir karakter dizisinin, örneğin *, kabuk için ne kadar önemli bir anlam taşıyabileceğini birkaç örnekte gördük. Bunu gerçekleştiren sürece *genişletme* denir. Genişletme ile, bir şey girdiğimizde, kabuk bu komutu uygulamadan önce onu başka bir şeye genişletir. Bunun ne anlama geldiğini göstermek için echo komutuna bir göz atalım. echo, çok basit bir görevi yerine getiren bir kabuk yerleşik komutudur. Metin argümanlarını standart çıktıya yazdırır.

```
[me@linuxbox ~]$ echo bu bir test bu
bir test
```

Bu oldukça basit. Echo'ya aktarılan tüm argümanlar görüntülenir. Başka bir örnek deneyelim.

```
[me@linuxbox ~]$ echo *
Masaüstü Belgeler ls-output.txt Müzik Resimler Genel Şablonlar Videolar
```

Peki ne oldu? Echo neden * yazdırmadı? Joker karakterlerle yaptığımız çalışmalardan hatırlayacağımız gibi, * karakteri dosya adındaki herhangi bir karakterle eşleşir, ancak ilk tartışmamızda görmediğimiz şey, kabuğun bunu nasıl yaptığıydı. Basit cevap, kabuğun echo komutu çalıştırılmadan önce * karakterini başka bir şeye (bu örnekte, geçerli çalışma dizinindeki dosyaların adlarına) genişletmesidir. ENTER tuşuna

basıldığında, kabuk komut yürütülmeden önce komut satırındaki tüm uygun karakterleri otomatik olarak genişletir, bu nedenle `echo` komutu `*` karakterini hiç görmez, sadece genişletilmiş sonucunu görür. Bunu bilerek, `echo'nun` beklendiği gibi davrandığını görebiliriz.

Yol adı 'ın Genişletilmesi

Joker karakterlerin çalıştığı mekanizma, *yol adı genişletme* olarak adlandırılır. Önceki bölümlerde kullandığımız bazı teknikleri denersek, bunların gerçekten genişletmeler olduğunu göreceğiz. Şu şekilde görünen bir ana dizin varsayalım:

```
[me@linuxbox ~]$ ls
Masaüstü    ls-output.txt Resimler Şablonlar
Belgeler Müzik              Genel      Videolar
```

Aşağıdaki genişletmeleri gerçekleştirebiliriz:

```
[me@linuxbox ~]$ echo D*
Masaüstü Belgeler
```

ve şunu:

```
[me@linuxbox ~]$ echo *s
Belgeler Resimler Şablonlar Videolar
```

veya hatta şunu:

```
[me@linuxbox ~]$ echo [[:upper:]]*
Masaüstü Belgeler Müzik Resimler Genel Şablonlar Videolar
```

ve ana dizinin ötesine bakarak şunu yapabiliriz:

```
[me@linuxbox ~]$ echo /usr/*/share
/usr/kerberos/share /usr/local/share
```

Gizli Dosyalarının Yol Adı Genişletmesi

Bildiğimiz gibi, nokta karakteriyle başlayan dosya adları gizlidir. Yol adı genişletme de bu davranışı dikkate alır. Aşağıdaki gibi bir genişletme, gizli dosyaları göstermez.

```
echo *
```

İlk bakışta, kalıbı nokta karakteriyle başlatarak gizli dosyaları genişletmeye dahil edebileceğimiz gibi görünebilir, örneğin:

```
echo .*
```

Bu, *globskipdots* kabuk seçeneği ayarlanmışsa (bash 5.2 ve sonraki sürümlerinde varsayılan ayardır) sorunsuz çalışır. Ancak, ayarlanmamışsa, `.` ve `..` sonuçlarda da görünecektir. Bu isimler geçerli çalışma dizinini ve onun üst dizinini ifade ettiğinden, bu kalıbı kullanmak muhtemelen yanlış bir sonuç verecektir. Aşağıdaki komutu denediğimizde bunu görebiliriz:

```
ls -d .* | less
```

Bu durumda yol adı genişletmeyi daha iyi gerçekleştirmek için daha spesifik bir desen kullanmalıyız.

```
echo .[!.]*
```

Bu desen, yalnızca bir nokta ile başlayan ve ardından herhangi bir karakter gelen tüm dosya adlarını genişletir. Bu, çoğu gizli dosyada doğru şekilde çalışacaktır (ancak yine de başında birden fazla nokta bulunan dosya adlarını içermeyecektir). `-A` seçeneği ("hemen hemen tüm") ile `ls` komutu, gizli dosyaların doğru bir listesini sağlayacaktır.

```
ls -A
```

Tilde i Genişletme

cd komutuna giriş bölümünden hatırlayabileceğimiz gibi, tilde karakteri (~) özel bir anlama sahiptir. Bir kelimenin başında kullanıldığında, adı belirtilen kullanıcının ana dizininin adını genişletir veya kullanıcı adı belirtilmezse, geçerli kullanıcının ana dizini genişletir.

```
[me@linuxbox ~]$ echo ~
/home/me
```

"bob" kullanıcısının bir hesabı varsa, şu şekilde genişler:

```
[me@linuxbox ~]$ echo ~bob
/home/bob
```

Aritmetik i Genişletme

Kabuk, genişletme yoluyla aritmetik işlemlerin yapılmasına izin verir. Bu, kabuk istemini hesap makinesi olarak kullanmamızı sağlar.

```
[me@linuxbox ~]$ echo $((2 + 2)) 4
```

Aritmetik genişletme aşağıdaki formu kullanır:

`$((`*`ifade`*`))`

burada *ifade,* değerler ve aritmetik operatörlerden oluşan bir aritmetik ifadedir.

Aritmetik genişletme yalnızca tamsayıları (tam sayılar, ondalık sayılar hariç) destekler, ancak oldukça fazla sayıda farklı işlem gerçekleştirebilir. Tablo 7-1, desteklenen işlemcilerden birkaçını açıklamaktadır.

Tablo 7-1: Aritmetik İşlemciler

İşleç	Açıklama
+	Toplama
-	Çıkarma
*	Çarpma
/	Bölme (ancak, genişletme yalnızca tamsayı aritmetiğini desteklediğinden, sonuçların tamsayılar olduğunu unutmayın)
%	Modulo, basitçe "kalan" anlamına gelir
**	Üs alma

Aritmetik ifadelerde boşluklar önemli değildir ve ifadeler iç içe geçebilir. Örneğin, 5'in karesini 3 ile çarpmak için şunu kullanabiliriz:

komut satırında. Bu komutun birçok özelliği, sistemin küçük veri parçalarını depolama ve her parçaya bir ad verme yeteneği ile ilgilidir. Daha doğru bir ifadeyle *değişkenler* olarak adlandırılan bu tür birçok parça, incelememiz için kullanılabilir. Örneğin, `USER` adlı değişken kullanıcı adımızı içerir. Parametre genişletmeyi çağırmak ve `USER'ın` içeriğini görüntülemek için şunu yapabiliriz:

```
[me@linuxbox ~]$ echo $USER me
```

Kullanılabilir değişkenlerin listesini görmek için şunu deneyin:

```
[me@linuxbox ~]$ printenv | less
```

Diğer genişletme türlerinde, bir kalıbı yanlış yazarsak genişletme gerçekleşmez ve `echo` komutu sadece yanlış yazılan kalıbı görüntüler. Parametre genişletmede, bir değişkenin adını yanlış yazarsak genişletme yine de gerçekleşir, ancak sonuç boş bir dize olur:

```
[me@linuxbox ~]$ echo $SUER

[me@linuxbox ~]$
```

Komut in Değiştirilmesi

Komut ikamesi, bir komutun çıktısını genişletme olarak kullanmamızı sağlar.

```
[me@linuxbox ~]$ echo $(ls)
Masaüstü Belgeler ls-output.txt Müzik Resimler Genel Şablonlar Videolar
```

En sevdiklerimden biri şöyledir:

```
[me@linuxbox ~]$ ls -l $(which cp)
-rwxr-xr-x 1 root root 71516 2025-12-05 08:58 /bin/cp
```

Burada, `hangi cp`'nin sonuçlarını `ls` komutuna argüman olarak geçirdik, böylece

cp programının tam yol adını bilmemize gerek kalmadan programın listesini elde ettik. Sadece basit komutlarla sınırlı değiliz. Tüm ardışık düzenler kullanılabilir (burada sadece kısmi çıktı gösterilmiştir):

```
[me@linuxbox ~]$ file $(ls -d /usr/bin/* | grep zip)
/usr/bin/bunzip2:        `bzip2'ye sembolik bağlantı
/usr/bin/bzip2:          ELF 32-bit LSB yürütülebilir, Intel
80386, sürüm 1 (SYSV), dinamik olarak bağlanmış (paylaşımlı
kütüphaneler kullanır), GNU/Linux 2.6.9 için, soyulmuş
/usr/bin/bzip2recover: ELF 32-bit LSB yürütülebilir, Intel 80386,
sürüm 1 (SYSV), dinamik olarak bağlanmış (paylaşımlı kütüphaneler
kullanır), GNU/Linux 2.6.9 için, soyulmuş
/usr/bin/funzip:          ELF 32-bit LSB yürütülebilir, Intel
80386, sürüm 1 (SYSV), dinamik olarak bağlanmış (paylaşımlı
kütüphaneler kullanır), GNU/Linux 2.6.9 için, soyulmuş
/usr/bin/gpg-zip:         Bourne kabuk betiği metin yürütülebilir dosyası
/usr/bin/gunzip:          `../../bin/gunzip'e sembolik bağlantı
/usr/bin/gzip:            `../../bin/gzip'e sembolik bağlantı
/usr/bin/mzip:            `mtools'a sembolik bağlantı
```

Bu örnekte, ardışık işlemlerin sonuçları `dosya` komutunun argüman listesi haline geldi.

Eski kabuk programları tarafından kullanılan ve `bash` tarafından da desteklenen alternatif bir komut ikamesi sözdizimi vardır. Bu sözdizimi, dolar işareti ve parantez yerine *ters eğik çizgi* kullanır.

```
[me@linuxbox ~]$ ls -l `which cp`
-rwxr-xr-x 1 root root 71516 2025-12-05 08:58 /bin/cp
```

Alıntı

Shell'in genişletmeleri kaç farklı şekilde gerçekleştirebileceğini gördük, şimdi bunu nasıl kontrol edebileceğimizi öğrenme zamanı. Örneğin şunu ele alalım:

```
[me@linuxbox ~]$ echo bu bir          test
bu bir test
```

veya bu:

```
[me@linuxbox ~]$ echo Toplam 100,00 $ Toplam
00,00
```

İlk örnekte, kabuk tarafından yapılan *kelime bölme işlemi*, `echo` komutunun argüman listesinden fazladan boşlukları kaldırdı. İkinci örnekte, parametre genişletme, `$1` değişkeninin değeri tanımlanmamış olduğu için boş bir dize ile değiştirdi. Kabuk, istenmeyen genişletmeleri seçici olarak bastırmak için *alıntı yapma* adlı bir mekanizma sağlar.

Çift tırnak

İlk olarak ele alacağımız alıntı türü *çift tırnak* işaretleridir. Metni çift tırnak işaretleri içine alırsak, kabuk tarafından kullanılan tüm özel karakterler özel anlamlarını yitirir ve sıradan karakterler olarak değerlendirilir. Bunun istisnaları `$`, `\` (ters eğik çizgi) ve `` ` `` (ters tırnak) karakterleridir. Bu, kelime bölme, yol adı genişletme, tilde genişletme ve parantez genişletme işlemlerinin bastırıldığı, ancak parametre genişletme, aritmetik genişletme ve komut ikamesi işlemlerinin yine de gerçekleştirildiği anlamına gelir. Çift tırnak kullanarak, gömülü boşluklar içeren dosya adlarıyla başa çıkabiliriz. Diyelim ki, `two word-s.txt` adlı bir dosyanın talihsiz kurbanı olduk. Bunu komut satırında kullanmaya çalışırsak, kelime bölme işlemi nedeniyle bu dosya istenen tek bir argüman yerine iki ayrı argüman olarak değerlendirilir.

```
[me@linuxbox ~]$ ls -l two words.txt
ls: two dosyasına erişilemiyor: Böyle bir dosya veya dizin yok
ls: words.txt dosyasına erişilemiyor: Böyle bir dosya veya dizin yok
```

Çift tırnak kullanarak kelime bölünmesini durdurur ve istenen sonucu elde ederiz; dahası, hasarı bile onarabiliriz.

```
[me@linuxbox ~]$ ls -l "two words.txt"
-rw-rw-r-- 1 me    me    18 2016-02-20 13:03 two words.txt
[me@linuxbox ~]$ mv "two words.txt" two_words.txt
```

İşte! Artık o sinir bozucu çift tırnak işaretlerini yazmaya devam etmek zorunda değiliz.

Unutmayın, parametre genişletme, aritmetik genişletme ve komut ikamesi hala çift tırnak içinde gerçekleşir.

```
[me@linuxbox ~]$ echo "$USER $((2+2)) $(df -h)"
me 4 Dosya            Boyut Kullanılan Kullanılabilir
sistemi          1,6G 2,0M  1,6G Bağlanma Noktası
tmpfs
```

```
/dev/sda2          94G    19G    71G 21% /
tmpfs             7,8G      0  7,8G    0  /dev/shm
tmpfs             5,0M   4,0K  5,0M    1  /run/lock
/dev/sda1         975M   6,1M  969M    %  /boot/efi
/dev/sdb1         907G   574G  287G   %67 /home
tmpfs             1,6G   1,8M  1,6G    %  /run/user/1000
```

Komut ikamesi üzerinde çift tırnak işaretlerinin etkisini incelemek için biraz zaman ayırmalıyız. Öncelikle kelime bölmenin nasıl çalıştığını biraz daha ayrıntılı olarak inceleyelim. Önceki örneğimizde, kelime bölmenin metnimizdeki fazladan boşlukları nasıl kaldırdığını gördük.

```
[me@linuxbox ~]$ echo bu bir          test
bu bir test
```

Varsayılan olarak, kelime bölme işlemi boşluk, sekme ve satır sonu (satır besleme karakterleri) varlığını arar ve bunları kelimeler arasındaki *sınırlayıcılar* olarak değerlendirir. Bu, tırnak içine alınmamış boşluk, sekme ve satır sonlarının metnin bir parçası olarak değerlendirilmediği anlamına gelir. Bunlar yalnızca ayırıcı olarak işlev görür. Kelimeleri farklı argümanlara ayırdıkları için, örnek komut satırımız bir komut ve ardından dört farklı argüman içerir. Çift tırnak eklediğimizde:

```
[me@linuxbox ~]$ echo "bu bir          test"
bu bir         test
```

kelime bölme bastırılır ve gömülü boşluklar sınırlayıcı olarak değerlendirilmez; aksine, argümanın bir parçası haline gelirler. Çift tırnak eklendiğinde, komut satırımız bir komut ve ardından tek bir argüman içerir.

Kelime bölme mekanizması tarafından satır sonlarının sınırlayıcı olarak kabul edilmesi, komut ikamesi üzerinde ilginç, ancak ince bir etkiye neden olur. Aşağıdakileri ele alalım:

```
[me@linuxbox ~]$ echo $(df -h)
Dosya sistemi Boyut Kullanılan Kullanılabilir Kullanım% Bağlantı noktası
tmpfs 1,6G 2,0M 1,6G 1%
/run /dev/sda2 94G 19G 71G 21% / tmpfs 7.8G 0 7.8G 0% /dev/shm tmpfs
5.0M 4.0K 5.0M 1% /run/lock /dev/sda1 975M 6.1M 969M 1% /boot/efi
/dev/sdb1 907G 574G 287G 67% /home tmpfs 1.6G 1.8M 1.6G 1%
/run/user/1000
[me@linuxbox ~]$ echo "$(df -h)"
Dosya          Boyut Kullanılan Kullanılabilir
sistemi          Kullanım% Bağlantı Noktası/run
tmpfs
```

```
/dev/sda2          94G   19G   71G 21% /
tmpfs             7,8G     0  7,8G    0  /dev/shm
tmpfs             5,0M  4,0K  5,0M    1  /run/lock
/dev/sda1         975M  6,1M  969M    %  /boot/efi
/dev/sdb1         907G  574G  287G  %67  /home
tmpfs             1,6G  1,8M  1,6G    1  /run/user/1000
```

İlk durumda, tırnak işaretleri kullanılmayan komut ikamesi, 49 argüman içeren bir komut satırı ile sonuçlandı. İkinci durumda ise, gömülü boşluklar ve satır sonları içeren tek bir argüman içeren bir komut satırı ile sonuçlandı.

Tek Tırnak

Tüm genişletmeleri bastırmamız gerekiyorsa, *tek tırnak işaretleri* kullanırız. Aşağıda, tırnak işareti kullanılmayan, çift tırnak işaretleri ve tek tırnak işaretleri arasındaki karşılaştırma yer almaktadır:

```
[me@linuxbox ~]$ echo text ~/*.txt {a,b} $(echo foo) $((2+2)) $USER
text /home/me/ls-output.txt a b foo 4 me
[me@linuxbox ~]$ echo "text ~/*.txt {a,b} $(echo foo) $((2+2)) $USER" text
~/*.txt {a,b} foo 4 me
[me@linuxbox ~]$ echo 'text ~/*.txt {a,b} $(echo foo) $((2+2)) $USER' text
~/*.txt {a,b} $(echo foo) $((2+2)) $USER
```

Gördüğümüz gibi, her bir alıntı düzeyinde, giderek daha fazla genişletme bastırılmaktadır.

i Karakterlerinden Kaçınma

Bazen tek bir karakteri alıntılamak isteriz. Bunu yapmak için, karakterin önüne ters eğik çizgi koyabiliriz; bu bağlamda buna *kaçış karakteri* denir. Genellikle bu, genişlemeyi seçici olarak önlemek için çift tırnak içinde yapılır.

```
[me@linuxbox ~]$ echo "Kullanıcı $USER'ın bakiyesi: \$5.00" Kullanıcı
me'nin bakiyesi: $5.00
```

Dosya adındaki bir karakterin özel anlamını ortadan kaldırmak için kaçış karakteri kullanmak da yaygın bir uygulamadır. Örneğin, normalde kabuk için özel bir anlamı olan karakterleri dosya adlarında kullanmak mümkündür. Bunlar arasında `$`, `!`, `&`, boşluklar ve diğerleri bulunur. Dosya adına özel bir karakter eklemek için şunu yapabiliriz:

```
[me@linuxbox ~]$ mv bad\&filename good_filename
```

Ters eğik çizgi karakterinin görünmesini sağlamak için `\\` yazarak kaçış karakteri kullanın. Tek tırnak içinde ters eğik çizgi özel anlamını yitirir ve sıradan bir karakter olarak değerlendirilir.

Ters eğik çizgi kaçış karakterinin bir başka kullanımı da takma adları bastırmaktır. Örneğin, `ls` komutunun `ls='ls --color=auto'` olarak takma adlandırıldığını varsayarsak, birçok Linux dağıtımında varsayılan ayar budur, komutun önüne ters eğik çizgi ekleyebiliriz ve takma ad yok sayılacak ve `ls` komutu renk seçeneği olmadan yürütülecektir.

Geriye eğik çizgi kaçış

Escape karakteri olarak rolünün yanı sıra, ters eğik çizgi, *kontrol kodları* adı verilen belirli özel karakterleri temsil etmek için bir notasyonun parçası olarak da kullanılır. ASCII kodlama şemasında ilk 32 karakter, terminalimiz gibi tele-tip cihazlara komutları iletmek için kullanılır. Bu kodların bazıları tanıdıktır (sekme, geri al, satır besleme ve satır başı), bazıları ise değildir (boşluk, iletim sonu ve onay).

Kaçış Dizisi	Anlam
`\a`	Zil (bilgisayarın bip sesi çıkarmasına neden olan bir uyarı)
`\b`	Geri
`\n`	Yeni satır. Unix benzeri sistemlerde, bu bir satır beslemesi oluşturur.
`\r`	Satır başı
`\t`	Sekme

Yukarıdaki tablo, yaygın olarak kullanılan bazı ters eğik çizgi kaçış dizilerini listelemektedir. Ters eğik çizgi kullanılarak yapılan bu gösterimin ardındaki fikir, C programlama dilinde ortaya çıkmış ve kabuk dahil olmak üzere birçok başka dil tarafından da benimsenmiştir.

`Echo` komutuna `-e` seçeneği eklenerek kaçış dizilerinin yorumlanması sağlanır. Bunları `$' '` içine de yerleştirebilirsiniz. Burada, belirtilen saniye sayısını bekleyen ve ardından çıkan basit bir program olan `sleep` komutunu kullanarak, basit bir geri sayım sayacı oluşturabiliriz:

```
sleep 10; echo -e "Süre doldu\a"
```

Şunu da yapabiliriz:

```
sleep 10; echo "Süre doldu" $'\a'
```

Özet

Kabuğu kullanmaya devam ettikçe, genişletmelerin ve alıntıların giderek daha sık kullanılacağını göreceğiz, bu nedenle bunların nasıl çalıştığını iyi anlamak mantıklıdır. Aslında, bunların kabuk hakkında öğrenilmesi gereken en önemli konular olduğu söylenebilir. Genişletmeyi doğru bir şekilde anlamadan, kabuk her zaman gizem ve kafa karışıklığı kaynağı olmaya devam edecek ve potansiyel gücünün çoğu boşa harcanacaktır.

Daha Fazla Okuma

- `Bash` man sayfasında, bu konuları daha resmi bir şekilde ele alan genişletme ve alıntı yapma ile ilgili önemli bölümler bulunmaktadır.
- *Bash Referans Kılavuzu* da genişletme ve alıntı yapma ile ilgili bölümler içerir: http://www.gnu.org/software/bash/manual/bashref.html

8 – Gelişmiş Klavye Püf Noktaları

Unix'i sık sık şaka yollu olarak "yazmayı sevenler için işletim sistemi" olarak tanımlarım. Elbette, komut satırına sahip olması bile bunun kanıtıdır. Ancak komut satırı kullanıcıları *o* kadar da yazmayı sevmezler. Yoksa neden bu kadar çok komutun `cp`, `ls`, `mv` ve `rm` gibi kısa isimleri olsun ki? Aslında, komut satırının en değerli hedeflerinden biri tembelliktir: en az tuş vuruşuyla en fazla işi yapmak. Diğer bir hedef ise parmaklarımızı klavyeden kaldırıp fareye uzanmak zorunda kalmamaktır. Bu bölümde, klavye kullanımını daha hızlı ve verimli hale getiren `bash` özelliklerine bakacağız.

Aşağıdaki komutlar ele alınacaktır:

- `clear` – Ekranı temizler
- `history` – Geçmiş listesinin içeriğini görüntüler

Komut Satırı Düzenleme

`bash`, komut satırı düzenlemeyi uygulamak için *Readline* adlı bir kütüphane (farklı programların kullanabileceği paylaşılan bir rutinler koleksiyonu) kullanır. Bunun bir kısmını daha önce gördük. Örneğin, ok tuşlarının imleci hareket ettirdiğini biliyoruz, ancak daha birçok özellik var. Bunları işimizde kullanabileceğimiz ek araçlar olarak düşünün. Hepsini öğrenmek önemli değil, ancak çoğu oldukça kullanışlıdır. İstediğinizi seçin.

Not**:** Aşağıdaki tuş dizileri (özellikle `Alt` tuşunu kullananlar) diğer işlevler için GUI tarafından engellenebilir. Sanal konsol kullanıldığında tüm tuş dizileri düzgün çalışmalıdır.

İmleç Hareketi

Aşağıdaki tabloda imleci hareket ettirmek için kullanılan tuşlar listelenmiştir:

Tablo 8-1: İmleç Hareket Komutları

Tuş	Eylem

`Ctrl-a`	İmleci satırın başına taşır.
`Ctrl-e`	İmleci satırın sonuna taşır.
`Ctrl-f`	İmleci bir karakter ileri taşır; sağ ok tuşuyla aynıdır.
`Ctrl-b`	İmleci bir karakter geriye taşır; sol ok tuşuyla aynıdır.
`Alt-f`	İmleci bir kelime ileri hareket ettirir.
`Alt-b`	İmleci bir kelime geriye hareket ettirir.
`Ctrl-l`	Ekranı temizleyin ve imleci sol üst köşeye taşıyın. `clear` komutu da aynı işlevi görür.

Metni Değiştirme

Komut oluştururken hata yapabileceğimizden, bunları verimli bir şekilde düzeltmenin bir yoluna ihtiyacımız vardır. Tablo 8-2, komut satırındaki karakterleri düzenlemek için kullanılan klavye komutlarını açıklamaktadır.

Tablo 8-2: Metin Düzenleme Komutları

Tuş	Eylem
`Ctrl-d`	İmlecin bulunduğu konumdaki karakteri siler.
`Ctrl-t`	İmleç konumundaki karakteri, önündeki karakterle değiştirir (değiştirir).
`Alt-t`	İmleç konumundaki kelimeyi, ondan önceki kelimeyle değiştirir.
`Alt-l`	İmleç konumundaki karakterleri kelimenin sonuna kadar küçük harfe dönüştürür.
`Alt-u`	İmleç konumundan kelimenin sonuna kadar olan karakterleri büyük harfe dönüştürün.

Metin Kesme ve Yapıştırma (Killing ve Yanking)

Readline belgeleri, genellikle kesme ve yapıştırma olarak adlandırdığımız işlemi ifade etmek için *killing* ve *yanking* terimlerini kullanır. Kesilen öğeler, *kill-ring* adı verilen bir tampon (bellekteki geçici bir depolama alanı) içinde saklanır.

Tablo 8-3: Kesme ve Yapıştırma Komutları

Tuş	Eylem
`Ctrl-k`	İmlecin bulunduğu yerden satır sonuna kadar olan metni siler.
`Ctrl-u`	İmleç konumundan satır başına kadar olan metni siler.
`Alt-d`	İmlecin bulunduğu yerden mevcut kelimenin sonuna kadar olan metni sil.
`Alt-Backspace`	İmlecin bulunduğu yerden geçerli kelimenin başına kadar olan metni siler. İmleç kelimenin başında ise, önceki kelimeyi siler.
`Ctrl-y`	Silme halkasından metni al ve imlecin bulunduğu yere ekle.

Meta Tuşu

`Bash` man sayfasının "READLINE" bölümünde bulunan Readline belgelerine göz atarsanız, *meta tuşu* terimiyle karşılaşırsınız. Modern klavyelerde bu tuş `Alt` tuşuna karşılık gelir, ancak her zaman böyle olmamıştır.

Eski zamanlarda (PC'lerden önce, Unix'ten sonra), herkesin kendi bilgisayarı yoktu. Sahip oldukları şey, *terminal* adı verilen bir cihazdı. Terminal, metin görüntüleme ekranı ve klavyeye sahip, metin karakterlerini görüntülemek ve imleci hareket ettirmek için yeterli elektronik donanıma sahip bir iletişim cihazıydı. Genellikle seri kablo ile daha büyük bir bilgisayara veya daha büyük bir bilgisayarın iletişim ağına bağlanıyordu. Birçok farklı marka terminal vardı ve hepsinin farklı klavyeleri ve ekran özellikleri vardı. Hepsi en azından ASCII'yi anlama eğiliminde oldukları için, taşınabilir uygulamalar isteyen yazılım geliştiricileri en düşük ortak paydaya göre yazılım yazdılar. Unix sistemleri, terminaller ve bunların farklı ekran özellikleriyle başa çıkmak için ayrıntılı bir yönteme sahiptir. Readline'ın geliştiricileri, özel bir ekstra kontrol tuşunun varlığından emin olamadıkları için, bir tane icat ettiler ve ona *meta adını* verdiler. `Alt tuşu` modern klavyelerde meta tuşu olarak işlev görürken, terminal kullanıyorsanız (Linux'ta hala yapabilirsiniz!), `Esc tuşuna` basıp bırakarak `Alt tuşunu` basılı tutmakla aynı etkiyi elde edebilirsiniz.

Tamamlama

Kabuğun bize yardımcı olabileceği bir başka yol da, bir komut yazarken `Tab tuşuna` bastığımızda gerçekleşen *tamamlama* işlemidir. Bunun nasıl çalıştığını görelim. Diyelim ki ana dizini şöyle görünüyor:

```
[me@linuxbox ~]$ ls
Masaüstü    ls-output.txt Resimler Şablonlar          Videolar
Belgeler Müzik             Genel
```

Aşağıdakileri yazmayı deneyin, ancak **ENTER tuşuna basmayın**:

```
[me@linuxbox ~]$ ls l
```

Şimdi Tab tuşuna basın.

```
[me@linuxbox ~]$ ls ls-output.txt
```

Kabuk bizim için satırı nasıl tamamladı, gördünüz mü? Bir tane daha deneyelim. Yine EN-TER tuşuna basmayın.

```
[me@linuxbox ~]$ ls D
```

Tab tuşuna basın.

```
[me@linuxbox ~]$ ls D
```

Tamamlama yok, hiçbir şey yok. Bu, D'nin dizindeki birden fazla girdiyle eşleşmesi nedeniyle oldu. Tamamlamanın başarılı olması için, verdiğimiz "ipucu" belirsiz olmamalıdır. Daha ileri gidersek

```
[me@linuxbox ~]$ ls Do
```

ve ardından Tab tuşuna basın:

```
[me@linuxbox ~]$ ls Belgeler
```

tamamlama başarılıdır.

Bu örnek, en yaygın kullanımı olan yol adlarının tamamlanmasını gösterirken, tamamlama değişkenlerde (kelimenin başı $ ise), kullanıcı adlarında

(kelimenin başı ~ ise), komutlar (kelime satırın ilk kelimesi ise) ve ana bilgisayar adları (kelimenin başı @ ise) üzerinde de çalışır. Ana bilgisayar adı tamamlama, yalnızca /etc/hosts dosyasında listelenen ana bilgisayar adları için çalışır.

Tamamlama ile ilişkili bir dizi kontrol ve meta tuş dizisi vardır, bunlar Tablo 8-4'te listelenmiştir.

Tablo 8-4: Tamamlama Komutları

Anahtar	Eylem
Alt-?	Olası tamamlamaların bir listesini görüntüler. *Çoğu sistemde, bunu* Tab tuşuna *ikinci kez basarak da yapabilirsiniz, bu çok daha kolaydır.*
Alt-*	Olası tüm tamamlamaları ekler. Bu, birden fazla olası eşleşmeyi kullanmak istediğinizde kullanışlıdır.

Oldukça az bilinen birkaç tane daha vardır. Bunların listesi bash man sayfasında "READLINE" altında bulunur.

Programlanabilir Tamamlama

Bash'ın son sürümlerinde *programlanabilir tamamlama* adlı bir özellik vardır. Programlanabilir tamamlama, size (veya daha çok dağıtım sağlayıcınıza) tamamlama kuralları ekleme olanağı sağlar. Genellikle bu, belirli uygulamalar için destek eklemek amacıyla yapılır. Örneğin, bir komutun seçenek listesi için tamamlamalar eklemek veya bir uygulamanın desteklediği belirli dosya türlerini eşleştirmek mümkündür. Ubuntu'da varsayılan olarak tanımlanmış oldukça geniş bir set vardır. Programlanabilir tamamlama, daha sonraki bölümlerde ele alacağımız bir tür mini kabuk betiği olan kabuk işlevleri tarafından uygulanır. Merak ediyorsanız,

```
set | less
```

ve bunları bulup bulamadığınızı kontrol edin. Tüm dağıtımlar bunları varsayılan olarak içermez.

Geçmişi Kullanma

Bölüm 1'de gördüğümüz gibi, bash girilen komutların geçmişini saklar. Bu komut listesi, ana dizinde
.bash_history adlı bir dosyada tutulur. Geçmiş özelliği, özellikle komut satırı düzenleme ile birleştirildiğinde, yazmamız gereken miktarı azaltmak için yararlı bir kaynaktır.

Arama Geçmişi

İstediğimiz zaman, aşağıdakileri yaparak geçmiş listesinin içeriğini görüntüleyebiliriz:

```
[me@linuxbox ~]$ history | less
```

Varsayılan olarak, çoğu modern Linux dağıtımı bash'ı girdiğimiz son 1000 komutu saklayacak şekilde yapılandırır. Bu değeri nasıl ayarlayacağımızı 11. Bölümde göreceğiz. Diyelim ki /usr/bin'i listelemek için kullandığımız komutları bulmak istiyoruz. Bunu yapmanın bir yolu şöyledir:

```
[me@linuxbox ~]$ history | grep /usr/bin
```

Ve sonuçlarımız arasında şöyle ilginç bir komut içeren bir satır olduğunu varsayalım:

```
 88 ls -l /usr/bin > ls-output.txt
```

88, geçmiş listesindeki komutun satır numarasıdır. Bunu, *geçmiş genişletme* adı verilen başka bir genişletme türü kullanarak hemen kullanabiliriz. Bulduğumuz satırı kullanmak için şunu yapabiliriz:

```
[me@linuxbox ~]$ !88
```

bash, !88'i geçmiş listesinin 88. satırının içeriği olarak genişletecektir. Bir sonraki bölümde ele alacağımız başka geçmiş genişletme biçimleri de vardır.

bash ayrıca geçmiş listesini aşamalı olarak arama olanağı da sunar. Bu, bash'a karakterleri girerken geçmiş listesini aramasını söyleyebileceğimiz ve her ek karakterle aramamızı daha da daraltabileceğimiz anlamına gelir. Aşamalı aramayı başlatmak için Ctrl-r tuşuna basıp ardından aradığımız metni yazın. Bulduğumuzda, ENTER tuşuna basarak komutu çalıştırabilir veya Ctrl-j tuşuna basarak geçmiş listesinden satırı geçerli komut satırına kopyalayabiliriz. Metnin bir sonraki görünümünü bulmak için (geçmiş listesinde "yukarı" hareket etmek için) Ctrl-r tuşuna tekrar basın. Aramayı sonlandırmak için Ctrl-g veya Ctrl-c tuşuna basın. Burada bunun nasıl çalıştığını görelim:

```
[me@linuxbox ~]$
```

Önce Ctrl-r tuşuna basın.

```
(ters i-arama)`':
```

Komut istemi, ters artımlı arama yaptığımızı göstermek için değişir. "Ters" çünkü "şimdi"den geçmişteki bir zamana doğru arama yapıyoruz. Ardından, arama metnini yazmaya başlarız. Bu örnekte, bu `metin /usr/bin'dir`:

```
(reverse-i-search)`/usr/bin': ls -l /usr/bin > ls-output.txt
```

Arama hemen sonucumuzu döndürür. Sonuçla birlikte, `ENTER tuşuna` basarak komutu çalıştırabiliriz veya `Ctrl-j tuşuna` basarak komutu mevcut komut satırımıza kopyalayabilir ve daha fazla düzenleme yapabiliriz. Kopyalayalım. `Ctrl-j tuşuna` basın.

```
[me@linuxbox ~]$ ls -l /usr/bin > ls-output.txt
```

Kabuk istemimiz geri döner ve komut satırımız yüklenir ve çalışmaya hazır hale gelir!

Tablo 8-5, geçmiş listesini işlemek için kullanılan bazı tuş vuruşlarını listeler.

Tablo 8-5: Geçmiş Komutları

Tuş	Eylem
`Ctrl-p`	Önceki geçmiş girdisine geç. Bu, yukarı ok tuşuyla aynı eylemdir.
`Ctrl-n`	Bir sonraki geçmiş girdisine geçin. Bu, aşağı ok tuşuyla aynı işlemdir.
`Alt-<`	Geçmiş listesinin başına (üstüne) geçin.
`Alt->`	Geçmiş listesinin sonuna (altına), yani geçerli komut satırına geçin.
`Ctrl-r`	Artımlı aramayı tersine çevirir. Bu, geçerli komut satırından geçmiş listesinin başına doğru artımlı olarak arama yapar.
`Alt-p`	Ters arama, artımlı değil. Bu tuşla, arama dizesini yazın ve arama gerçekleştirilmeden önce `ENTER tuşuna` basın.
`Alt-n`	İleri arama, artımlı değil.
`Ctrl-o`	Geçmiş listesindeki mevcut öğeyi yürütü ve bir sonrakine geç. Bu, geçmiş listesindeki bir dizi komutu yeniden yürütmeye çalışırken kullanışlıdır.

Geçmiş Genişletme

Kabuk, `!` karakterini kullanarak geçmiş listesindeki öğeler için özel bir genişletme türü sunar. Geçmiş listesinden bir girişi eklemek için ünlem işaretinin ardından bir sayı gelebileceğini daha önce görmüştük. Tablo 8-6'da açıklanan bir dizi başka genişletme özelliği de vardır.

Tablo 8-6: Geçmiş Genişletme Komutları

Sıra	Eylem
	Son komutu tekrarlar. Yukarı ok tuşuna basıp `ENTER` tuşuna basmak daha kolay olabilir.
`!`*sayı*	Geçmiş listesi öğe *numarasını* tekrarla.
`!`*dize*	Son geçmiş listesi öğesini *string* ile başlayan kısmı tekrarlayın.
`!?`*dize*	Son geçmiş listesi öğesini içeren *dizeyi* tekrarla.

Geçmiş listesi öğelerinin içeriğinden kesinlikle emin değilseniz, `!string` ve `!?string` biçimlerini kullanırken dikkatli olun. Genişletmemize ":p" ekleyerek bu sorunu bir ölçüde azaltabiliriz. Bu, kabuğa genişletmenin sonucunu yazdırmasını ve komut geçmişine yerleştirmesini söyler. İşte bir örnek:

```
[me@linuxbox ~]$ !ls:p
ls -l /usr/bin > ls-output.txt
```

Komut geri çağrıldı ve geçmiş listesinin en son öğesi olarak yerleştirildiğine göre, `Yukarı Ok` ve `ENTER` veya `!!` ve `ENTER` tuşlarıyla komutu çalıştırabiliriz.

Bu arada, `!!` gibi geçmiş genişletmeleri geçmiş listesine kaydedilmez, ancak sonuçları kaydedilir.

Geçmiş genişletme mekanizmasında daha birçok özellik mevcuttur, ancak bu konu zaten çok karmaşıktır ve devam edersek kafalarımız patlayabilir. `Bash` man sayfasının HISTORY EXPAN-SION bölümü tüm ayrıntıları ele almaktadır. Keşfetmekten çekinmeyin!

`script`

`Bash'teki` komut geçmişi özelliğine ek olarak, çoğu Linux dağıtımı, tüm kabuk oturumunu kaydetmek ve bir dosyada saklamak için kullanılabilen `script` adlı bir program içerir. Komutun temel sözdizimi aşağıdaki gibidir:

`script [`*`dosya`*`]`

burada *dosya*, kaydı saklamak için kullanılan dosyanın adıdır. Dosya belirtilmezse, `typescript` dosyası kullanılır. Programın seçeneklerinin ve özelliklerinin tam listesi için `script` man sayfasına bakın.

Özet

Bu bölümde, yoğun yazanlar için iş yükünü azaltmaya yardımcı olan kabuğun sağladığı bazı klavye püf noktalarını ele aldık. Zaman geçtikçe ve komut satırını daha fazla kullanmaya başladıkça, bu püf noktalarını öğrenmek için bu bölüme başvurabiliriz. Şimdilik, bunları isteğe bağlı ve potansiyel olarak yararlı olarak değerlendirin.

Daha Fazla Okuma

- Wikipedia'da bilgisayar terminalleri hakkında iyi bir makale var: http://en.wikipedia.org/wiki/Computer_terminal

9 – İzinler

Unix geleneğindeki işletim sistemleri, MS-DOS geleneğindeki işletim sistemlerinden, sadece *çoklu görev* sistemleri değil, aynı zamanda *çoklu kullanıcı* sistemleri olmalarıyla ayrılır.

Bu tam olarak ne anlama geliyor? Bu, birden fazla kişinin aynı anda bilgisayarı kullanabileceği anlamına gelir. Tipik bir bilgisayarın genellikle tek bir klavyesi ve monitörü olsa da, yine de birden fazla kullanıcı tarafından kullanılabilir. Örneğin, bir bilgisayar bir ağa veya İnternet'e bağlıysa, uzak kullanıcılar `ssh` (güvenli kabuk) aracılığıyla oturum açabilir ve bilgisayarı kullanabilir. Aslında, uzak kullanıcılar grafik uygulamaları çalıştırabilir ve grafik çıktısını uzak bir ekranda görüntüleyebilir.

Linux'un çoklu kullanıcı özelliği yeni bir "yenilik" değil, işletim sisteminin tasarımına derinlemesine yerleşmiş bir özelliktir. Unix'in yaratıldığı ortamı göz önüne alındığında, bu çok mantıklıdır. Yıllar önce, bilgisayarlar "kişisel" hale gelmeden önce, büyük, pahalı ve merkeziydi. Örneğin, tipik bir üniversite bilgisayar sistemi, bir binada bulunan büyük bir merkezi bilgisayardan ve kampüsün her yerine yerleştirilmiş, her biri büyük merkezi bilgisayara bağlı terminallerden oluşuyordu. Bilgisayar, aynı anda birçok kullanıcıyı destekliyordu.

Bunu pratik hale getirmek için, kullanıcıları birbirlerinden korumak için bir yöntem geliştirilmesi gerekiyordu. Sonuçta, bir kullanıcının eylemlerinin bilgisayarı çökertmesine izin verilemezdi, ne de bir kullanıcı başka bir kullanıcıya ait dosyalara müdahale edemezdi.

Modern kişisel bilgisayar kullanıcıları için, bu çoklu kullanıcı özelliği, tek kullanıcılı sistemlerde bile önemlidir, çünkü sistem düzeyinde değişiklikler yalnızca ayrı bir yönetici hesabı tarafından yapılabildiğinden, kullanıcı bilgisayarını daha güvenli bir şekilde kullanabilir.

Bu bölümde, sistem güvenliğinin bu önemli kısmını inceleyecek ve aşağıdaki komutları tanıtacağız:

- `id` – Kullanıcı kimliğini görüntüler.
- `chmod` – Bir dosyanın modunu değiştirir.
- `umask` – Varsayılan dosya izinlerini ayarlar.
- `su` – Başka bir kullanıcı olarak kabuk çalıştırır.

- `sudo` – Bir komutu başka bir kullanıcı olarak çalıştırın.
- `chown` – Bir dosyanın sahibini değiştirir.
- `chgrp` – Bir dosyanın grup sahipliğini değiştirir.
- `addgroup` – Sisteme bir kullanıcı veya grup eklemek.
- `usermod` – Bir kullanıcı hesabını değiştirir.
- `passwd` – Bir kullanıcının şifresini değiştirir.

Kullanıcılar, Grup Üyeleri ve Diğerleri

Bölüm 3'te sistemi incelerken, `/etc/shadow` gibi bir dosyayı incelemeye çalışırken bir sorunla karşılaşmış olabiliriz:

```
[me@linuxbox ~]$ file /etc/shadow
/etc/shadow: normal dosya, okuma izni yok
[me@linuxbox ~]$ less /etc/shadow
/etc/shadow: İzin reddedildi
```

Bu hata mesajının nedeni, normal kullanıcılar olarak bu dosyayı okuma iznimiz olmamasıdır.

Unix güvenlik modelinde, bir kullanıcı dosya ve dizinlere *sahip* olabilir. Bir kullanıcı bir dosya veya dizine sahip olduğunda, bu dosya veya dizine erişim üzerinde kontrol sahibi olur. Kullanıcılar ise, sahipleri tarafından dosya ve dizinlere erişim izni verilen bir veya daha fazla kullanıcıdan oluşan bir *gruba* ait olabilirler. Bir gruba erişim izni vermenin yanı sıra, sahip, herkese bazı erişim hakları da verebilir; bu haklara *"diğerleri*" (bazen "*dünya*" olarak da adlandırılır) denir. Kimliğimizle ilgili bilgileri bulmak için `id` komutunu kullanırız.

```
[me@linuxbox ~]$ id
uid=500(ben) gid=500(ben) groups=500(ben)
```

Çıktıya bakalım. Kullanıcı hesapları oluşturulduğunda, kullanıcılara *kullanıcı kimliği* (*uid*) adı verilen bir numara atanır ve bu numara, insanlar için bir kullanıcı adına eşlenir. Kullanıcıya bir *birincil grup kimliği* (*gid*) atanır ve kullanıcı ek gruplara da ait olabilir. Önceki örnek Fedora sisteminden alınmıştır. Ubuntu gibi diğer sistemlerde çıktı biraz farklı görünebilir:

```
[me@linuxbox ~]$ id
```

```
uid=1000(ben) gid=1000(ben)
gruplar=4(adm),20(dialout),24(cdrom),25(floppy),29(audio),30(dip),44(v
ideo),46(plugdev),108(lpadmin),114(admin),1000(ben)
```

Gördüğümüz gibi, uid ve gid numaraları farklıdır. Bunun nedeni, Fedora'nın normal kullanıcı hesaplarının numaralandırmasını 500'den başlatması, Ubuntu'nun ise 1000'den başlatmasıdır. Ayrıca Ubuntu kullanıcısının çok daha fazla gruba ait olduğunu da görebiliriz. Bu, Ubuntu'nun sistem aygıtları ve hizmetleri için ayrıcalıkları yönetme şekliyle ilgilidir.

Peki bu bilgiler nereden geliyor? Linux'taki pek çok şey gibi, birkaç metin dosyasından geliyor. Kullanıcı hesapları /etc/passwd dosyasında, gruplar ise `/etc/`group dosyasında tanımlanmıştır. Kullanıcı hesapları ve gruplar oluşturulduğunda, bu dosyalar, kullanıcının şifresi hakkındaki bilgileri içeren `/etc/shadow` dosyası ile birlikte değiştirilir. Her kullanıcı hesabı için, `/etc/passwd` dosyası kullanıcı (giriş) adını, uid'yi, gid'yi, kullanıcının gerçek adını, ana dizini ve giriş kabuğunu tanımlar.
`/etc/passwd` ve `/etc/group` dosyalarının içeriğini incelersek, normal kullanıcı hesaplarının yanı sıra süper kullanıcı (her zaman uid 0) ve çeşitli diğer sistem kullanıcıları için hesaplar olduğunu görürüz.

Bir sonraki bölümde, işlemleri ele aldığımızda, bu diğer "kullanıcıların" bazılarının aslında oldukça meşgul olduğunu göreceğiz.

Birçok Unix benzeri sistem normal kullanıcıları "kullanıcılar" gibi ortak bir gruba atarken, modern Linux uygulaması, kullanıcıyla aynı ada sahip benzersiz, tek üyeli bir grup oluşturmaktır. Bu, belirli izin atama türlerini kolaylaştırır.

Okuma, Yazma ve Yürütme

Dosyalar ve dizinlere erişim hakları okuma erişimi, yazma erişimi ve yürütme erişimi olarak tanımlanır. ls komutunun çıktısına bakarsak, bunun nasıl uygulandığına dair bazı ipuçları elde edebiliriz:

```
[me@linuxbox ~]$ > foo.txt
[me@linuxbox ~]$ ls -l foo.txt
-rw-rw-r-- 1 me     me    0 2016-03-06 14:52 foo.txt
```

Listedeki ilk 10 karakter *dosya öznitelikleridir*. Bu karakterlerin ilki *dosya türüdür*. Tablo 9-1, en sık karşılaştığımız dosya türlerini açıklamaktadır (daha az yaygın olan başka türler de vardır):

Tablo 9-1: Dosya Türleri

Özellik	Dosya Türü

`-`	Normal bir dosya.
`d`	Bir dizin.
`l`	Sembolik bir bağlantı. Sembolik bağlantılarda, kalan dosya özniteliklerinin her zaman "rwxrwxrwx" olduğunu ve sahte değerler olduğunu unutmayın. Gerçek dosya öznitelikleri, sembolik bağlantının işaret ettiği dosyanın öznitelikleridir.
`c`	*Özel karakter dosyası*. Bu dosya türü, verileri bir bayt akışı olarak işleyen bir aygıtı ifade eder, örneğin bir terminal veya `/dev/null` gibi bir aygıtı ifade eder.
`b`	*Özel blok dosyası*. Bu dosya türü, sabit disk, SSD veya DVD sürücüsü gibi verileri bloklar halinde işleyen bir aygıtı ifade eder.

Dosya özniteliğinin kalan dokuz karakteri, *dosya modu olarak* adlandırılır *ve* dosyanın sahibi, dosyanın grup sahibi ve diğer herkes için okuma, yazma ve yürütme izinlerini temsil eder.

Kullanıcı	Grup	Diğer
`rwx`	`rwx`	`rwx`

Tablo 9-2, `r`, `w` ve `x` mod özniteliklerinin dosyalar ve dizinler üzerindeki etkisini açıklamaktadır:

Tablo 9-2: İzin Öznitelikleri

Özellik	Dosyalar	Dizinler
`r`	Bir dosyanın açılıp okunmasına izin verir.	Bir dizinin içeriğinin listelenmesine izin verir, ancak yürütme özniteliği de ayarlanmadıkça dosya bilgileri kullanılamaz.
`w`	Bir dosyanın yazılmasına veya kesilmesine izin verir, ancak bu öznitelik dosyaların yeniden adlandırılmasına veya silinmesine izin vermez. Dosyaları silme veya yeniden adlandırma yeteneği, dizin öznitelikleri tarafından belirlenir.	Yürütme özniteliği de ayarlanmışsa, bir dizindeki dosyaların oluşturulmasına, silinmesine ve yeniden adlandırılmasına izin verir.
`x`	Bir dosyanın	Bir dizinin programa girilmesine

	program olarak değerlendirilmesine ve yürütülmesine izin verir. Komut dosyası dillerinde yazılmış program dosyaları, yürütülebilmesi için okunabilir olarak da ayarlanmalıdır.	(yani `cd` *`dizini`*) ve dizin meta verileri (yani `ls -l` *`dizini`*) erişilmek üzere. `cp`, `rm` ve `mv` gibi dosya işlemleri, dizine bu erişimi gerektirir.

Tablo 9-3, dosya öznitelik ayarlarının bazı örneklerini vermektedir:

Tablo 9-3: İzin Özniteliği Örnekleri

Dosya Öznitelikleri	Anlam
`-rwx------`	Dosya sahibi tarafından okunabilir, yazılabilir ve çalıştırılabilir normal bir dosya. Başka hiç kimse erişemez.
`-rw-------`	Dosya sahibi tarafından okunabilir ve yazılabilir normal bir dosya. Başka hiç kimse erişemez.
`-rw-r--r--`	Dosya sahibi tarafından okunabilir ve yazılabilir normal bir dosya. Dosya sahibi grubunun üyeleri dosyayı okuyabilir. Dosya diğerleri tarafından okunabilir.
`-rwxr-xr-x`	Dosyanın sahibi tarafından okunabilir, yazılabilir ve çalıştırılabilir olan normal bir dosya. Dosya, diğer herkes tarafından okunabilir ve çalıştırılabilir.
`-rw-rw----`	Dosyanın sahibi ve dosyanın grup sahibi üyeleri tarafından okunabilir ve yazılabilir olan normal bir dosya.
`lrwxrwxrwx`	Sembolik bir bağlantı. Tüm sembolik bağlantılar "sahte" izinlere sahiptir. Gerçek izinler, sembolik bağlantının işaret ettiği gerçek dosyada tutulur.
`drwxrwx---`	Bir dizin. Sahibi ve sahibi grubunun üyeleri dizine girebilir ve dizin içinde dosya oluşturabilir, yeniden adlandırabilir ve silebilir.
`drwxr-x---`	Bir dizin. Sahibi, dizine girebilir ve dizin içindeki dosyaları oluşturabilir, yeniden adlandırabilir ve silebilir. Sahip grubunun üyeleri dizine girebilir, ancak dosya oluşturamaz, silemez veya yeniden adlandıramaz.

`chmod` – Dosya Modunu Değiştirme

Bir dosya veya dizinin modunu (izinlerini) değiştirmek için `chmod` komutu kullanılır.

Yalnızca dosyanın sahibi veya süper kullanıcı bir dosya veya dizinin modunu değiştirebilir. `chmod`, mod değişikliklerini belirtmenin iki farklı yolunu destekler: sekizlik sayı gösterimi ve sembolik gösterim. Önce sekizlik sayı gösterimini ele alacağız.

Sekizlik Nedir?

Sekizlik (8 tabanlı) ve onun kuzeni *onaltılık* (16 tabanlı) sayı sistemleri, bilgisayarlarda sayıları ifade etmek için sıklıkla kullanılan sayı sistemleridir. Biz insanlar, (en azından çoğumuz) 10 parmakla doğduğumuz için, 10 tabanlı sayı sistemini kullanarak sayarız. Öte yandan, bilgisayarlar sadece bir parmakla doğdukları için tüm sayma işlemlerini *ikili* (2 tabanlı) olarak yaparlar. Sayı sistemlerinde sadece 0 ve 1 olmak üzere iki rakam vardır. Dolayısıyla, ikili sistemde sayma işlemi şöyle görünür:

0, 1, 10, 11, 100, 101, 110, 111, 1000, 1001, 1010, 1011...

Sekizlik sistemde sayma, sıfırdan yediye kadar olan rakamlarla yapılır: 0, 1, 2,

3, 4, 5, 6, 7, 10, 11, 12, 13, 14, 15, 16, 17, 20, 21...

Onaltılık sayma, sıfırdan dokuza kadar olan rakamları ve "A"dan "F"ye kadar olan harfleri kullanır:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10, 11, 12, 13...

İkili sayı sisteminin mantığını anlayabiliyoruz (çünkü bilgisayarların sadece bir parmağı var), peki sekizlik ve onaltılık sayı sistemlerinin ne yararı var? Cevap, insanlara sağladığı kolaylıkla ilgili. Çoğu zaman, küçük veri parçaları bilgisayarlarda *bit desenleri* olarak temsil edilir. Örneğin bir RGB rengini ele alalım. Çoğu bilgisayar ekranında, her piksel üç renk bileşeninden oluşur: sekiz bit kırmızı, sekiz bit yeşil ve sekiz bit mavi. Hoş bir orta mavi renk, 24 basamaklı bir sayı olur:

010000110110111111001101

Bütün gün bu tür sayıları okumak ve yazmak ister miydiniz? Sanmıyorum. İşte burada başka bir sayı sistemi yardımcı olabilir. Onaltılık sayıdaki her basamak, ikili sistemde dört basamağı temsil eder. Sekizlik sistemde ise her basamak üç ikili basamağı temsil eder. Dolayısıyla 24 basamaklı orta mavimiz altı basamaklı onaltılık bir sayıya sıkıştırılabilir:

436FCD

Onaltılık sayıdaki rakamlar ikili sayıdaki bitlerle "sıralandığı" için, rengimizin kırmızı bileşeninin 43, yeşil bileşeninin 6F ve mavi bileşeninin CD olduğunu görebiliriz.

> Günümüzde onaltılık gösterim (genellikle "hex" olarak adlandırılır) sekizlikten daha yaygındır, ancak birazdan göreceğimiz gibi, sekizlik sisteminin üç ikili biti ifade etme yeteneği çok yararlı olacaktır...

Sekizlik gösterimde, istenen izinlerin modelini ayarlamak için sekizlik sayıları kullanırız. Sekizlik sayıdaki her basamak üç ikili basamağı temsil ettiğinden, bu, dosya modunu depolamak için kullanılan şemaya çok iyi uyar. Tablo 9-4 ne demek istediğimizi göstermektedir.

Tablo 9-4: İkili ve Sekizlik Dosya Modları

Sekizlik	İkili	Dosya Modu
0	000	---
1	001	--x
2	010	-w-
3	011	-wx
4	100	r--
5	101	r-x
6	110	rw-
7	111	rwx

Üç sekizlik rakam kullanarak, dosya modunu sahibi, grup sahibi ve dünya için ayarlayabiliriz.

```
[me@linuxbox ~]$ > foo.txt
[me@linuxbox ~]$ ls -l foo.txt
-rw-rw-r-- 1 me     me    0 2016-03-06 14:52 foo.txt
[me@linuxbox ~]$ chmod 600 foo.txt
[me@linuxbox ~]$ ls -l foo.txt
-rw------- 1 ben    ben   0 2016-03-06 14:52 foo.txt
```

`600` argümanını geçirerek, grup sahibi ve diğerlerinden tüm izinleri kaldırırken, sahibinin okuma ve yazma izinlerini ayarlayabildik. Sekizlikten ikiliye eşlemeyi hatırlamak zahmetli görünebilir, ancak genellikle sadece birkaç yaygın olanı kullanmamız gerekir: 7 (`rwx`), 6 (`rw-`), 5 (`r-x`), 4 (`r--`) ve 0 (`---`).

`chmod`, dosya modlarını belirtmek için sembolik notasyonu da destekler. Sembolik notasyon üç bölüme ayrılır.

- Değişikliğin kimi etkileyeceği
- Hangi işlemin gerçekleştirileceği
- Hangi izin ayarlanacak.

Kimin etkileneceğini belirtmek için, Tablo 9-5'te gösterildiği gibi "u", "g", "o" ve "a" karakterlerinin bir kombinasyonu kullanılır.

Tablo 9-5: `chmod` Sembolik Notasyonu

Sembol	Anlam
`u`	"Kullanıcı"nın kısaltması, yani dosya veya dizinin sahibi.
`g`	Grup sahibi.
`o`	Diğerleri için kısaltma.
`a`	"Tüm"ün kısaltmasıdır. "u", "g" ve "o" harflerinin birleşimidir.

Hiçbir karakter belirtilmezse, "tümü" varsayılır. İşlem, bir izin ekleneceğini belirten "+", bir izin kaldırılacağını belirten "-" veya yalnızca belirtilen izinlerin uygulanacağını ve diğerlerinin kaldırılacağını belirten "=" olabilir.

İzinler "r", "w" ve "x" karakterleriyle belirtilir. Tablo 9-6 sembolik gösterim örnekleri vermektedir:

Tablo 9-6: `chmod` Sembolik Notasyon Örnekleri

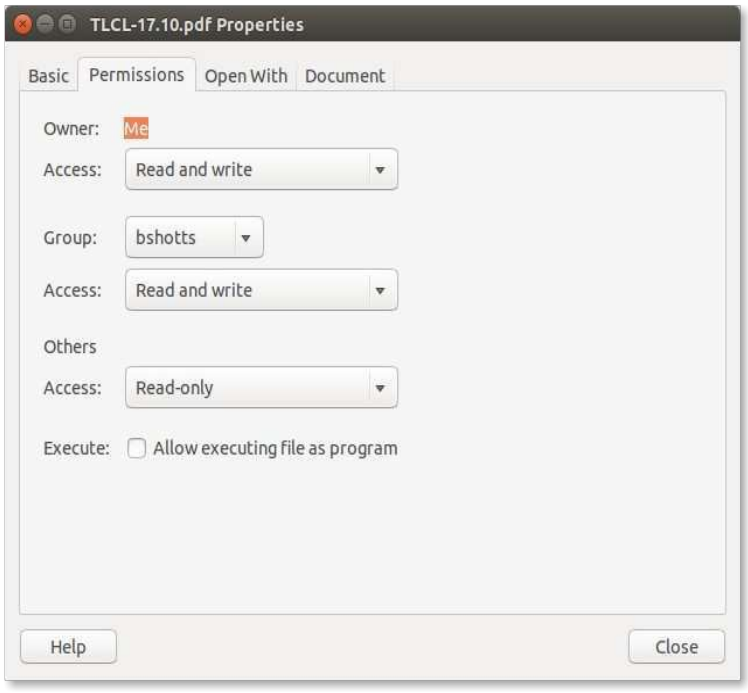
Notasyon	Anlam
`u+x`	Sahibine yürütme izni ekleyin.
`u-x`	Sahibin yürütme iznini kaldırın.
`+x`	Kullanıcı, grup ve diğerleri için yürütme izni ekleyin. Bu, `a+x` ile eşdeğerdir.
`o-rw`	Kullanıcı ve grup sahibi dışındaki herkesten okuma ve yazma izinlerini kaldırır.
`go=rw`	Grup sahibi ve kullanıcı dışındaki herkese okuma ve yazma izni verin. Grup sahibi veya diğerlerinin daha önce yürütme izni varsa, bu izin kaldırılır.
`u+x,go=rx`	Kullanıcı için yürütme izni ekleyin ve grup ve diğerleri için okuma ve yürütme izinlerini ayarlayın. Birden fazla belirtim virgülle ayrılabilir.

Bazı insanlar sekizlik gösterimi kullanmayı tercih ederken, bazıları sembolik gösterimi çok sever. Sembolik gösterim, diğerlerini etkilemeden tek bir özniteliği ayarlamamıza olanak tanıyan bir avantaj sunar.

Daha fazla ayrıntı ve seçeneklerin listesi için `chmod` man sayfasına bakın. "--recursive" seçeneği ile ilgili bir uyarı: bu seçenek hem dosyalara hem de dizinlere etki eder, bu nedenle dosyalar ve dizinlerin aynı izinlere sahip olmasını nadiren istediğimiz için umduğumuz kadar kullanışlı değildir.

GUI ile Dosya Modunu Ayarlama

Dosya ve dizinlerin izinlerinin nasıl ayarlandığını gördük, şimdi GUI'deki izin diyaloglarını daha iyi anlayabiliriz. Hem Files (GNOME) hem de Dolphin (KDE) uygulamalarında, bir dosya veya dizin simgesine sağ tıklamak bir özellikler diyalog penceresi açar. İşte GNOME'dan bir örnek:



Şekil 2: GNOME dosya izinleri iletişim kutusu

Burada sahip, grup ve diğerleri için ayarları görebiliriz.

`umask` – Varsayılan İzinleri Ayarlama

`umask` komutu, bir dosya oluşturulduğunda dosyaya verilen varsayılan izinleri kontrol eder. Dosyanın mod özniteliklerinden kaldırılacak bit maskesini ifade etmek için sekizlik gösterim kullanır. Bir göz atalım.

```
[me@linuxbox ~]$ rm -f foo.txt
[me@linuxbox ~]$ umask
0002
[me@linuxbox ~]$ > foo.txt [me@linuxbox
~]$ ls -l foo.txt
-rw-rw-r-- 1 ben    ben   0 2025-03-06 14:53 foo.txt
```

İlk olarak, yeni bir başlangıç yapabilmek için `foo.txt` dosyasının eski kopyalarını sildik. Ardından, mevcut değeri görmek için `umask` komutunu argüman olmadan çalıştırdık. Komut, maskemizin ondalık gösterimi olan `0002` değerini verdi (`0022` değeri de yaygın olarak kullanılan başka bir varsayılan değerdir). Ardından, `foo.txt` dosyasının yeni bir örneğini oluşturduk ve izinlerini inceledik.

Hem kullanıcının hem de grubun okuma ve yazma iznine sahip olduğunu, diğer herkesin ise yalnızca okuma iznine sahip olduğunu görebiliriz. World'ün yazma iznine sahip olmamasının nedeni maskenin değeridir. Örneğimizi tekrarlayalım, bu sefer maskeyi kendimiz ayarlayalım.

```
[me@linuxbox ~]$ rm foo.txt
[me@linuxbox ~]$ umask 0000
[me@linuxbox ~]$ > foo.txt [me@linuxbox
~]$ ls -l foo.txt
-rw-rw-rw- 1 me     me    0 2025-03-06 14:58 foo.txt
```

Maskeyi `0000` olarak ayarladığımızda (etkin olarak kapattığımızda), dosyanın artık herkes tarafından yazılabilir olduğunu görürüz. Bunun nasıl çalıştığını anlamak için sekizlik sayılara tekrar bakmamız gerekir. Maskeyi `0002` olarak değiştirir, ikili sayıya dönüştürür ve özniteliklerle karşılaştırırsak ne olduğunu görebiliriz.

Orijinal dosya modu	`---` `rw-` `rw-` `rw-`
Maske	`000` `000` `000` `010`
Sonuç	`---` `rw-` `rw-` `r--`

Şu anda baştaki sıfırları (buna birazdan geleceğiz) göz ardı edin ve

maskemizde 1'in göründüğü yerlerde bir öznitelik kaldırılmış — bu durumda, genel yazma izni. Maskenin yaptığı budur. Maskenin ikili değerinde 1'in göründüğü her yerde bir öznitelik kaldırılır. `0022` maske değerine bakarsak, ne yaptığını görebiliriz.

Orijinal dosya modu	`--- rw- rw- rw-`
Maske	`000 000 010 010`
Sonuç	`--- rw- r-- r--`

Yine, ikili değerde 1 görünen yerlerde, karşılık gelen öznitelik ayarlanmamıştır. Bunun nasıl çalıştığını anlamak için bazı değerlerle (yedi rakamını deneyin) oynayın. İşiniz bittiğinde, temizlemeyi unutmayın.

```
[me@linuxbox ~]$ rm foo.txt; umask 0002
```

Çoğu zaman maskeyi değiştirmemiz gerekmez; dağıtım tarafından sağlanan varsayılan değer yeterlidir. Ancak, bazı yüksek güvenlikli durumlarda, maskeyi kontrol etmek isteyebiliriz.

Bazı Özel İzinler

Genellikle üç basamaklı bir sayı olarak ifade edilen sekizlik izin maskesini görsek de, teknik olarak daha doğru olanı dört basamaklı olarak ifade etmektir. Neden? Çünkü okuma, yazma ve yürütme izinlerine ek olarak, daha az kullanılan başka izin ayarları da vardır.

Bunlardan ilki *setuid bitidir* (sekizlik 4000). Yürütülebilir bir dosyaya uygulandığında, *etkin kullanıcı kimliğini* gerçek kullanıcının (programı fiilen çalıştıran kullanıcı) kimliğinden programın sahibinin kimliğine ayarlar. Çoğu zaman bu, süper kullanıcıya ait birkaç programa verilir. Sıradan bir kullanıcı *"setuid* root" olan bir programı çalıştırdığında, program süper kullanıcının etkin ayrıcalıklarıyla çalışır. Bu, programın sıradan bir kullanıcının normalde erişiminin yasak olduğu dosya ve dizinlere erişmesine olanak tanır. Açıkçası, bu durum güvenlik endişelerini artırdığı için, setuid programlarının sayısı kesinlikle minimumda tutulmalıdır.

En az kullanılan ikinci ayar, *setgid bitidir* (sekizlik 2000). Bu bit, setuid biti gibi, *etkin grup kimliğini* gerçek kullanıcının *gerçek grup kimliğinden* dosya sahibinin grup kimliğine değiştirir. Bir dizinde setgid biti ayarlanmışsa, dizinde yeni oluşturulan dosyalara dosyanın yaratıcısının grup sahipliği yerine dizinin grup sahipliği verilir. Bu, bir paylaşılan dizinde, bir

ortak grubun üyeleri, dosya sahibinin birincil grubu ne olursa olsun, dizindeki tüm dosyalara erişmesi gerektiğinde yararlıdır.

Üçüncüsü, *yapışkan bit* (sekizlik 1000) olarak adlandırılır. Bu, eski Unix'ten kalma bir özelliktir ve burada yürütülebilir bir dosyayı "takas edilemez" olarak işaretlemek mümkündü. Linux, dosyalarda yapışkan biti yok sayar, ancak bir dizine uygulandığında, kullanıcı dizinin sahibi, dosyanın sahibi veya süper kullanıcı olmadığı sürece kullanıcıların dosyaları silmesini veya yeniden adlandırmasını engeller. Bu, genellikle `/tmp` gibi paylaşılan bir dizine erişimi kontrol etmek için kullanılır.

İşte bu özel izinleri ayarlamak için sembolik notasyonla `chmod'u` kullanmanın bazı örnekleri. İşte bir programa setuid atamanın bir örneği:

`chmod u+s` *`program`*

Aşağıda, bir dizine setgid atamanın bir örneği verilmiştir:

`chmod g+s` *`dir`*

Son olarak, bir dizine yapışkan bit atamanın bir örneği:

`chmod +t` *`dir`*

`ls` komutunun çıktısını görüntülerken, özel izinleri belirleyebilirsiniz. İşte bazı örnekler. İlk olarak, setuid atanan bir program örneği:

`-rwsr-xr-x`

İşte setgid özniteliğine sahip bir dizinin örneği:

`drwxrwsr-x`

İşte sticky bit atanmış bir dizinin örneği:

`drwxrwxrwt`

Kimlik Değiştirme

Bazen başka bir kullanıcının kimliğini almamız gerekebilir. Genellikle bazı yönetim görevlerini gerçekleştirmek için süper kullanıcı ayrıcalıkları elde etmek isteriz, ancak bir hesabı test etmek gibi durumlarda başka bir normal kullanıcı "olmak" da mümkündür. Alternatif bir kimlik almak için üç yol vardır.

1. Oturumu kapatın ve alternatif kullanıcı olarak tekrar oturum açın.
2. `su` komutunu kullanın.
3. `sudo` komutunu kullanın.

İlk tekniği atlayacağız çünkü nasıl yapılacağını biliyoruz ve diğer ikisi kadar kullanışlı değil. Kendi kabuk oturumumuzdan, `su` komutu başka bir kullanıcının kimliğini üstlenmemizi ve o kullanıcının kimliğiyle yeni bir kabuk oturumu başlatmamızı veya o kullanıcı olarak tek bir komut vermemizi sağlar. `sudo` komutu, bir yöneticinin `/etc/sudoers` adlı bir yapılandırma dosyası oluşturmasına ve belirli kullanıcıların üstlendiği kimlikle çalıştırmasına izin verilen belirli komutları tanımlamasına olanak tanır. Hangi komutun kullanılacağı, büyük ölçüde kullandığınız Linux dağıtımına bağlıdır. Dağıtımınız muhtemelen her iki komutu da içerir, ancak yapılandırması birini veya diğerini tercih eder. `su` ile başlayacağız. Ancak, `su` kullanımının modern Linux dağıtımlarında gözden düştüğünü unutmayın.

`su` – İkame Kullanıcı ve Grup Kimlikleri ile Kabuk Çalıştırma

`su` komutu, başka bir kullanıcı olarak bir kabuk başlatmak için kullanılır. Komut sözdizimi şöyledir:

```
su [-[l]] [kullanıcı]
```

"-l" seçeneği eklenirse, sonuçta ortaya çıkan kabuk oturumu, belirtilen kullanıcı için bir *oturum açma kabuğu* olur. Bu, kullanıcının ortamının yüklendiği ve çalışma dizininin kullanıcının ana dizinine değiştirildiği anlamına gelir. Genellikle istediğimiz budur. Kullanıcı belirtilmezse, süper kullanıcı varsayılır. (Garip bir şekilde) `-l` seçeneğinin `-` olarak kısaltılabileceğini ve en sık bu şekilde kullanıldığını unutmayın. Kök hesabının bir şifresi olduğunu varsayarsak (modern dağıtımlarda bu geleneksel değildir), süper kullanıcı için bir kabuğu şu şekilde başlatabiliriz:

```
[me@linuxbox ~]$ su -
Şifre: [root@linuxbox
~]#
```

Komutu girdikten sonra, süper kullanıcının şifresi istenir. Şifre başarıyla girilirse, bu kabuğun süper kullanıcı ayrıcalıklarına sahip olduğunu belirten yeni bir kabuk istemci görünür (sonunda `$` yerine `#` işareti bulunur) ve geçerli çalışma dizini artık süper kullanıcının ana dizini olur (normalde `/root`). Yeni kabuğa girdikten sonra, süper kullanıcı olarak komutları yürütebiliriz. İşlem bittiğinde, önceki kabuğa dönmek için `exit komutunu` girin.

```
[root@linuxbox ~]# exit
[me@linuxbox ~]$
```

Bu şekilde `su komutunu` kullanarak yeni bir etkileşimli komut başlatmak yerine tek bir komut çalıştırmak da mümkündür.

```
su -c 'komut'
```

Bu formu kullanarak, tek bir komut satırı yeni kabuğa yürütülmek üzere aktarılır. Komutu tırnak içine almak önemlidir, çünkü genişlemenin bizim kabuğumuzda değil, yeni kabukta gerçekleşmesini istiyoruz.

```
[me@linuxbox ~]$ su -c 'ls -l /root/*' Şifre:
-rw------- 1 root root      754 2025-08-11 03:19 /root/anaconda-ks.cfg

/root/Mail:
toplam 0
[me@linuxbox ~]$
```

`sudo` – Başka Bir Kullanıcı Olarak Komut Yürütme

`sudo` komutu birçok yönden `su komutuna` benzer, ancak bazı önemli ek özelliklere sahiptir. Yönetici, `sudo komutunu,` sıradan bir kullanıcının kontrollü bir şekilde farklı bir kullanıcı (genellikle süper kullanıcı) olarak komutları çalıştırmasına izin verecek şekilde yapılandırabilir. Özellikle, bir kullanıcı bir veya daha fazla belirli komutla sınırlandırılabilir ve diğer komutları kullanamaz. Bir diğer önemli fark ise, `sudo'nun` kullanımı için süper kullanıcının şifresine erişim gerekmemesidir. `Sudo` kullanarak kimlik doğrulama yapmak için kullanıcının kendi şifresi gerekir. Örneğin, `sudo'nun` süper kullanıcı ayrıcalıkları gerektiren "backup_script" adlı hayali bir yedekleme programını çalıştırmamıza izin verecek şekilde yapılandırıldığını varsayalım. `Sudo` ile bu işlem şu şekilde yapılır:

```
[me@linuxbox ~]$ sudo backup_script
Şifre:
Sistem Yedekleme Başlıyor...
```

Komutu girdikten sonra, şifremiz (süper kullanıcının şifresi değil) istenir ve kimlik doğrulama tamamlandığında, belirtilen komut yürütülür. `su` ve `sudo` arasındaki önemli bir fark, `sudo'nun` yeni bir kabuk başlatmaması ve başka bir kullanıcının ortamını yüklememesidir. Bu, komutların `sudo` kullanılmadan olduğundan farklı bir şekilde tırnak içine alınmasına gerek olmadığı anlamına gelir. Bu davranışın geçersiz kılınabileceğini unutmayın.

çeşitli seçenekler belirterek. Ayrıca, `sudo'nun -i` seçeneği kullanılarak etkileşimli bir süper kullanıcı oturumu başlatmak için de kullanılabileceğini unutmayın (`su - komutuna` çok benzer). Ayrıntılar için `sudo` man sayfasına bakın.

`sudo` tarafından verilen ayrıcalıkları görmek için `-l` seçeneğini kullanarak bunları listeleyin:

```
[me@linuxbox ~]$ sudo -l
Kullanıcı me bu ana bilgisayarda aşağıdaki komutları
    çalıştırabilir: (ALL) ALL
```

Modern Linux Dağıtımları ve `sudo`

Düzenli kullanıcıların sıkça karşılaştığı sorunlardan biri, süper kullanıcı ayrıcalıkları gerektiren görevleri yerine getirmektir. Bu görevler arasında yazılım yükleme ve güncelleme, sistem yapılandırma dosyalarını düzenleme ve aygıtlara erişim yer alır. Windows dünyasında, bu genellikle kullanıcılara yönetici ayrıcalıkları vererek yapılır. Bu, kullanıcıların bu görevleri yerine getirmelerini sağlar. Ancak, aynı zamanda kullanıcı tarafından yürütülen programların da aynı yeteneklere sahip olmasını sağlar. Bu, çoğu durumda istenen bir durumdur, ancak virüsler gibi kötü amaçlı yazılımların (*malware*) bilgisayarı serbestçe kullanmasına da izin verir.

Unix dünyasında, Unix'in çok kullanıcılı mirası nedeniyle, normal kullanıcılar ve yöneticiler arasında her zaman daha büyük bir ayrım olmuştur. Unix'te benimsenen yaklaşım, süper kullanıcı ayrıcalıklarını yalnızca gerektiğinde vermektir. Bunu yapmak için, genellikle `su` ve `sudo` komutları kullanılır.

Yıllar önce, çoğu Linux dağıtımı bu amaçla `su komut`una güveniyordu. `su`, `sudo'nun` gerektirdiği yapılandırmayı gerektirmiyordu ve Unix'te kök hesabı olması geleneksel bir uygulamaydı. Ancak bu, bir sorun ortaya çıkardı. Kullanıcılar gereksiz yere root olarak çalışmaya yöneliyorlardı. Hatta bazı kullanıcılar, can sıkıcı "izin reddedildi" mesajlarından kurtulmak için sistemlerini yalnızca root kullanıcısı olarak çalıştırıyorlardı. Bu şekilde Linux sisteminin güvenliği Windows sisteminin güvenliği seviyesine indirgeniyordu. Hiç de iyi bir fikir değildi.

Ubuntu piyasaya sürüldüğünde, yaratıcıları farklı bir yol izlediler. Ubuntu, varsayılan olarak root hesabına girişleri devre dışı bırakır (hesap için şifre belirlemeyerek) ve bunun yerine süper kullanıcı ayrıcalıkları vermek için `sudo` kullanır. İlk kullanıcı hesabına `sudo` aracılığıyla süper kullanıcı ayrıcalıklarına tam erişim izni verilir ve sonraki kullanıcı hesaplarına da benzer yetkiler verilebilir. Bu ayrıcalık verme yöntemi, günümüzde çoğu modern dağıtımda kabul gören standarttır.

`chown` – Dosya Sahibini ve Grubunu Değiştirme

`chown` komutu, bir dosya veya dizinin sahibini ve grup sahibini değiştirmek için kullanılır. Bu komutu kullanmak için süper kullanıcı ayrıcalıkları gereklidir. `chown` komutunun sözdizimi şöyledir:

```
chown [sahip][:[grup]] dosya...
```

`chown`, komutun ilk argümanına bağlı olarak dosya sahibini ve/veya dosya grup sahibini değiştirebilir. Tablo 9-7'de bazı örnekler verilmiştir.

Tablo 9-7: `chown` Argüman Örnekleri

Argüman	Sonuç
`bob`	Dosyanın sahipliğini mevcut sahibinden kullanıcıya değiştirir `bob`
`bob:users`	Dosyanın sahipliğini mevcut sahibinden kullanıcıya değiştirir. `bob` ve dosya grubunun sahibini grup `kullanıcıları` olarak değiştirir.
`:admins`	Grup sahibini grup `yöneticileri` olarak değiştirir. Dosya sahibi değişmez.
`bob:`	Dosya sahibini mevcut sahibinden kullanıcı `bob` olarak değiştirir ve grup sahibini kullanıcı `bob'un` oturum açma grubu olarak değiştirir.

İki kullanıcımız olduğunu varsayalım: süper kullanıcı ayrıcalıklarına sahip `janet` ve sahip olmayan `tony`. Kullanıcı `janet`, kendi ana dizininden kullanıcı `tony'nin` ana dizinine bir dosya kopyalamak istiyor. Kullanıcı `janet`, `tony'nin` dosyayı düzenleyebilmesini istediği için, kopyalanan dosyanın sahipliğini `janet'ten tony'ye` değiştirir.

```
[janet@linuxbox ~]$ sudo cp myfile.txt ~tony
Şifre:
[janet@linuxbox ~]$ sudo ls -l ~tony/myfile.txt
 -rw-r--r-- 1 root root root 2025-03-20 14:30 /home/tony/myfile.txt
[janet@linuxbox ~]$ sudo chown tony: ~tony/myfile.txt
[janet@linuxbox ~]$ sudo ls -l ~tony/myfile.txt
 -rw-r--r-- 1 tony tony tony 2025-03-20 14:30 /home/tony/myfile.txt
```

Burada, kullanıcı `janet'in` dosyayı kendi dizininden kullanıcının ana dizinine kopyaladığını görüyoruz.

tony. Ardından, janet dosyanın sahipliğini root'tan (sudo kullanmanın bir sonucu olarak) tony'ye değiştirir. İlk argümanda sondaki iki noktayı kullanarak, janet dosyanın grup sahipliğini de tony'nin oturum açma grubuna, yani tony grubuna değiştirir.

İlk sudo kullanımından sonra janet'e şifresi sorulmadığını unutmayın. Bunun nedeni, çoğu yapılandırmada sudo'nun zamanlayıcısı bitene kadar birkaç dakika boyunca bize "güvenmesidir".

chgrp – Grup Sahipliğini Değiştirme

Eski Unix sürümlerinde, chown komutu yalnızca dosya sahipliğini değiştiriyordu, grup sahipliğini değiştirmiyordu. Bu amaçla, ayrı bir komut olan chgrp kullanılıyordu. Chown ile hemen hemen aynı şekilde çalışır, ancak daha sınırlıdır.

Ayrıcalıklarımızı Kullanma

Artık bu izinlerin nasıl çalıştığını öğrendiğimize göre, bunu gösterme zamanı geldi. Yaygın bir sorunun çözümünü göstereceğiz: paylaşımlı bir dizin oluşturmak. Arkadaşlarımız janet ve tony'yi tekrar ziyaret edelim. İkisi de müzik koleksiyonuna sahip ve her ikisinin de Ogg Vorbis veya MP3 formatında müzik dosyalarını depolayacağı paylaşımlı bir dizin oluşturmak istiyorlar. Daha önce olduğu gibi, janet kullanıcısı sudo aracılığıyla süper kullanıcı ayrıcalıklarına erişebilir.

Janet ve Tony'nin üye olacağı bir grup oluşturulması gerekiyor. Bu işlem iki adımda gerçekleştirilir. İlk olarak, groupadd komutunu kullanarak grubu oluştururuz, ardından usermod komutunu kullanarak gruba kullanıcılar ekleriz:

```
[janet@linuxbox ~]$ sudo groupadd music
[janet@linuxbox ~]$ sudo usermod -a -G music janet
[janet@linuxbox ~]$ sudo usermod -a -G music tony
```

usermod komutuyla kullanılan seçenekler --append ve --group kısaltmalarıdır ve belirtilen kullanıcıyı /etc/group dosyasındaki ilgili gruba eklerler.

Ardından, janet müzik dosyaları için bir dizin oluşturur.

```
[janet@linuxbox ~]$ sudo mkdir /usr/local/share/Music
Şifre:
```

Janet, kendi ana dizini dışındaki dosyaları işlediği için süper kullanıcı ayrıcalıkları gereklidir. Dizin oluşturulduktan sonra, aşağıdaki sahiplik ve izinlere sahiptir:

```
[janet@linuxbox ~]$ ls -ld /usr/local/share/Music
drwxr-xr-x 2 root root 4096 2025-03-21 18:05 /usr/local/share/Music
```

Gördüğümüz gibi, dizin root'a aittir ve izin modu 755'tir. Bu dizini paylaşılabilir hale getirmek için, janet yazma izni vermek üzere grup sahipliğini ve grup izinlerini değiştirmelidir.

```
[janet@linuxbox ~]$ sudo chown :music /usr/local/share/Music
[janet@linuxbox ~]$ sudo chmod 2775 /usr/local/share/Music [janet@linuxbox
~]$ ls -ld /usr/local/share/Music
drwxrwsr-x 2 root music 4096 2025-03-21 18:05 /usr/local/share/Music
```

chown komutunu kullanarak, janet dizinin grup sahibini music olarak ayarlar ve ardından chmod komutunu kullanarak dizin izinlerini 2775 olarak ayarlar. Bu, setguid'i, dizindeki tüm dosyaların dizinle aynı grup sahipliğini miras almasını sağlayacak şekilde ayarlar. Bunu chmod 2775 komutunu çalıştırarak yaptık, ancak chmod g+s komutuyla sembolik yöntemi kullanarak da aynı şeyi yapabilirdik.

Bütün bunlar ne anlama geliyor? Artık root'a ait olan ve music grubuna okuma ve yazma erişimi sağlayan /usr/local/share/Music adlı bir dizine sahibiz. Music grubunun üyeleri janet ve tony'dir; dolayısıyla janet ve tony, /usr/local/share/Music dizininde dosya oluşturabilirler. Diğer kullanıcılar dizinin içeriğini listeleyebilirler, ancak orada dosya oluşturamazlar.

Ancak hala bir sorunumuz var. Bu sistemdeki varsayılan umask 0022'dir ve bu, grup üyelerinin grubun diğer üyelerine ait dosyalara yazmasını engeller. Paylaşılan dizin yalnızca dosyalar içeriyorsa bu bir sorun olmaz, ancak bu dizin müzik depolayacağı ve müzik genellikle sanatçılar ve albümler hiyerarşisinde düzenlendiği için, grup üyeleri diğer üyeler tarafından oluşturulan dizinlerin içinde dosya ve dizinler oluşturabilme yeteneğine sahip olmalıdır. Janet ve Tony tarafından kullanılan umask değerini 0002 olarak değiştirmemiz gerekiyor.

janet umask değerini 0002 olarak ayarlar ve yeni bir test dosyası ve dizini oluşturur:

```
[janet@linuxbox ~]$ umask 0002
[janet@linuxbox ~]$ > /usr/local/share/Music/test_file
[janet@linuxbox ~]$ mkdir /usr/local/share/Music/test_dir
[janet@linuxbox ~]$ ls -l /usr/local/share/Music
drwxrwsr-x 2 janet    music 4096 2025-03-24 20:24 test_dir
-rw-rw-r-- 1 janet    müzik     0 2025-03-24 20:22 test_file
```

```
[janet@linuxbox ~]$
```

Artık hem dosyalar hem de dizinler, `müzik` grubunun tüm üyelerinin `Müzik` dizini içinde dosya ve dizin oluşturabilmeleri için doğru izinlerle oluşturulmuştur.

Geriye kalan tek sorun `umask`'tır. Gerekli ayar, oturumun sonuna kadar geçerlidir ve yeniden ayarlanması gerekir. 11. bölümde, `umask`'ı kalıcı olarak değiştirmeyi inceleyeceğiz.

Şifrenizi Değiştirme

Bu bölümde ele alacağımız son konu, kendimiz için (ve süper kullanıcı ayrıcalıklarına erişimimiz varsa diğer kullanıcılar için) şifreler belirlemektir. Şifre belirlemek veya değiştirmek için `passwd` komutu kullanılır. Komutun sözdizimi şöyledir:

```
passwd [kullanıcı]
```

Şifremizi değiştirmek için `passwd` komutunu gireriz. Eski şifremiz ve yeni şifremiz sorulur.

```
[me@linuxbox ~]$ passwd Şifremi
değiştiriyorum. Mevcut şifre:
Yeni şifre:
Yeni şifreyi tekrar girin:
passwd: şifre başarıyla güncellendi
```

`passwd` komutu, "güçlü" şifrelerin kullanılmasını zorunlu kılmaya çalışır. Bu, çok kısa, önceki şifrelere çok benzeyen, sözlükte bulunan veya çok kolay tahmin edilebilen şifreler için bir uyarı vereceği anlamına gelir.

```
[me@linuxbox ~]$ passwd Şifremi
değiştiriyorum. Mevcut şifre:
Yeni şifre:
KÖTÜ ŞİFRE: eskisine çok benziyor Yeni şifre:
YANLIŞ ŞİFRE: çok kısa Yeni şifre:
```

```
YANLIŞ ŞİFRE: sözlük kelimesine dayanıyor
```

Süper kullanıcı ayrıcalıklarına sahipsek, başka bir kullanıcının şifresini ayarlamak için passwd komutuna bir kullanıcı adı argümanı olarak belirtebilirsiniz. Süper kullanıcı, hesap kilitleme, şifre süresinin dolması vb. işlemleri yapmak için başka seçeneklere de sahiptir. Ayrıntılar için `passwd` man sayfasına bakın.

`passwd`, `addgroup` ve `usermod` komutları, shadow-utils paketindeki bir komut grubunun parçasıdır. Tablo 9-8, bu pakette bulunan bazı komutları listelemektedir:

Tablo 9-8: `shadow-utils` Komutları

Komut	Açıklama
`lastlog`	Tüm kullanıcıların veya belirli bir kullanıcının en son oturum açma bilgilerini bildirir.
`useradd`	Yeni bir kullanıcı oluşturur veya varsayılan yeni kullanıcı bilgilerini günceller.
`userdel`	Bir kullanıcı hesabını ve ilgili dosyaları siler.
`usermod`	Bir kullanıcı hesabını değiştirir.
`groupadd`	Yeni bir grup oluşturun.
`groupdel`	Bir grubu sil.
`groupmod`	Sistemdeki bir grup tanımını değiştirir.

Bu komutlar bu kitabın kapsamı dışında kaldığı için ayrıntılı olarak ele almayacağız. Daha fazla bilgi için, her komutun man sayfasına bakın.

Özet

Bu bölümde, Linux gibi Unix benzeri sistemlerin, dosya ve dizinlere okuma, yazma ve yürütme erişimi sağlamak için kullanıcı izinlerini nasıl yönettiğini gördük. Bu izin sisteminin temel fikirleri, Unix'in ilk günlerine kadar uzanır ve zamanın testinden oldukça iyi geçmiştir. Ancak Unix benzeri sistemlerdeki yerel izin mekanizması, daha modern sistemlerin ince ayrıntılarına sahip değildir.

Daha fazla bilgi

- Wikipedia'da kötü amaçlı yazılımlarla ilgili iyi bir makale bulunmaktadır: http://en.wikipedia.org/wiki/Malware

10 – İşlemler

Modern işletim sistemleri genellikle *çoklu* görevlidir, yani bir programdan diğerine hızla geçerek aynı anda birden fazla şey yapıyormuş gibi bir izlenim yaratırlar. Linux çekirdeği bunu *işlemler* kullanarak yönetir. İşlemler, Linux'un CPU'da sırasını bekleyen farklı programları düzenleme şeklidir.

Bazen bilgisayar yavaşlar veya bir uygulama yanıt vermeyi durdurur. Bu bölümde, hangi programların ne yaptığını incelememizi ve hatalı çalışan işlemleri sonlandırmamızı sağlayan komut satırında kullanılabilen bazı araçlara bakacağız.

Bu bölümde aşağıdaki komutlar tanıtılacaktır:

- `ps` – Mevcut işlemlerin anlık görüntüsünü raporlar.
- `top` – Görevleri görüntüler.
- `jobs` – Etkin işleri listeler.
- `bg` – Bir işi arka plana yerleştirir.
- `fg` – Bir işi ön plana yerleştirir.
- `kill` – Bir sürece sinyal gönderir.
- `killall` – Adına göre işlemleri sonlandır.
- `nice` - Değiştirilmiş zamanlama önceliği ile bir programı çalıştırır.
- `renice` - Çalışan işlemlerin önceliğini değiştirir.
- `nohup` - Bağlantı kesilmesine karşı bağışık bir komut çalıştırın.
- `halt/poweroff/reboot` - Sistemi durdurun, kapatın veya yeniden başlatın.
- `shutdown` – Sistemi kapatın veya yeniden başlatın.

Bir İşlemin Çalışma Şekli

Sistem başlatıldığında, çekirdek kendi faaliyetlerinden birkaçını işlem olarak başlatır ve `init` adlı bir programı çalıştırır. `init` ise tüm sistem hizmetlerini başlatan `systemd`'yi başlatır. Eski Linux dağıtımlarında `init`, bir dizi kabuk betiği (

/etc) benzer bir işlevi yerine getirmek için *init komut dosyaları* olarak adlandırılır. Birçok sistem hizmeti, arka planda çalışan ve herhangi bir kullanıcı arayüzü olmadan işlerini yapan *daemon programları* olarak uygulanır. Dolayısıyla, oturum açmamış olsak bile, sistem en azından rutin işleri yerine getirmekle biraz meşguldür.

Bir programın başka programları başlatabilmesi, işlem şemasında bir *ana işlemin* bir *alt işlem* üretmesi olarak ifade edilir.

Çekirdek, işleri düzenli tutmak için her işlemle ilgili bilgileri saklar. Örneğin, her işleme bir *işlem kimliği* (*PID*) adı verilen bir numara atanır. PID'ler artan sırada atanır ve init her zaman PID 1'i alır. Çekirdek ayrıca her işleme atanan belleği ve işlemlerin yürütmeye devam etmeye hazır olup olmadığını da takip eder. Dosyalar gibi, işlemlerin de sahipleri ve kullanıcı kimlikleri, etkin kullanıcı kimlikleri vb. vardır.

İşlemleri Görüntüleme

İşlemleri görüntülemek için en sık kullanılan araç (birkaç tane vardır) ps komutudur. ps programı birçok seçeneğe sahiptir, ancak en basit haliyle şu şekilde kullanılır:

```
[me@linuxbox ~]$ ps
  PID TTY          ZAMAN CMD
 5198 pts/1    00:00:00 bash
10129 pts/1    00:00:00 ps
```

Bu örnekteki sonuç, sırasıyla bash ve ps olan 5198 ve 10129 numaralı iki işlemi listelemektedir. Gördüğümüz gibi, varsayılan olarak ps bize çok fazla bilgi göstermez, sadece mevcut terminal oturumu ile ilişkili işlemleri gösterir. Daha fazlasını görmek için bazı seçenekler eklememiz gerekir, ancak bunu yapmadan önce ps tarafından üretilen diğer alanlara bakalım. TTY, "teletype" kelimesinin kısaltmasıdır ve işlem için *kontrol terminalini* ifade eder. Unix burada yaşını gösteriyor. TIME alanı, işlem tarafından tüketilen CPU süresidir. Gördüğümüz gibi, her iki işlem de bilgisayarı çok fazla zorlamıyor.

Bir seçenek eklersek, sistemin ne yaptığını daha geniş bir perspektiften görebiliriz.

```
[me@linuxbox ~]$ ps x
  PID TTY      STAT   ZAMAN KOMUT
 2799 ?        Ssl    0:00 /usr/libexec/bonobo-activation-server -ac
 2820 ?        Sl     0:01 /usr/libexec/evolution-data-server-1.10 --
15647 ?        Ss     0:00 /bin/sh /usr/bin/startkde
15751 ?        Ss     0:00 /usr/bin/ssh-agent /usr/bin/dbus-launch --
15754 ?        S      0:00 /usr/bin/dbus-launch --exit-with-session
15755 ?        Ss     0:01 /bin/dbus-daemon --fork --print-pid 4 -pr
```

```
15774 ?        Ss     0:02 /usr/bin/gpg-agent -s -daemon
15793 ?        S      0:00 start_kdeinit --new-startup +kcminit_start
15794 ?        Ss     0:00 kdeinit Çalışıyor...
15797 ?        S      0:00 dcopserver -nosid
ve daha fazlası...
```

"x" seçeneğini eklemek (başında tire işareti olmadığını unutmayın), `ps'ye` hangi terminal tarafından kontrol edildiklerine bakılmaksızın tüm işlemlerimizi göstermesini söyler. `TTY` sütununda "?" işareti varsa, kontrol terminali olmadığı anlamına gelir. Bu seçeneği kullanarak, sahip olduğumuz tüm işlemlerin listesini görebiliriz.

Sistem çok sayıda işlem çalıştırdığı için, `ps` uzun bir liste üretir. Daha kolay görüntüleme için `ps` çıktısını `less'e` yönlendirmek genellikle yararlıdır. Bazı seçenek kombinasyonları da uzun çıktı satırları üretir, bu nedenle terminal emülatör penceresini büyütmek de iyi bir fikir olabilir.

Çıktıya `STAT` başlıklı yeni bir sütun eklenmiştir. `STAT`, "durum"un kısaltmasıdır ve Tablo 10-1'de gösterildiği gibi işlemin mevcut durumunu gösterir.

Tablo 10-1: İşlem Durumları

Durum	Anlam
`R`	Çalışıyor. Bu, işlemin çalıştığı veya çalışmaya hazır olduğu anlamına gelir.
`S`	Uyku. İşlem çalışmıyor; daha ziyade, tuş vuruşu veya ağ paketi gibi bir olay bekliyor.
`D`	Kesintisiz uyku. İşlem, disk sürücüsü gibi bir I/O beklemektedir.
`T`	Durduruldu. İşlemin durdurulması talimatı verilmiştir. Bu konu hakkında daha fazla bilgi bu bölümün ilerleyen kısımlarında verilecektir.
`Z`	İşlevsiz veya "zombi" işlem. Bu, sonlandırılmış ancak üst işlemi tarafından temizlenmemiş bir alt işlemdir.
`<`	Yüksek öncelikli bir işlem. Bir işleme daha fazla önem verilerek CPU'da daha fazla zaman tanınabilir. Bir işlemin bu özelliğine *niceness (nezaket)* denir. Yüksek öncelikli bir işlemin *nezaketi* daha azdır, çünkü CPU'nun zamanının daha fazlasını alır ve diğerlerine daha az zaman kalır.
`N`	Düşük öncelikli bir işlem. Düşük öncelikli bir işlem ("nice" işlem), daha yüksek öncelikli diğer işlemler tamamlandıktan sonra işlemci zamanı alır.

İşlem durumunun ardından başka karakterler gelebilir. Bunlar çeşitli egzotik işlem özelliklerini gösterir. Daha fazla ayrıntı için `ps` man sayfasına bakın.

Bir başka popüler seçenek grubu ise "aux"dur (başında tire işareti yoktur). Bu seçenek bize daha da fazla bilgi verir.

```
[me@linuxbox ~]$ ps aux
KULLANICI    PID  %CPU  %MEM     VSZ    RSS TTY      STAT BAŞLANGIÇ  ZAMAN KOMUT
root       1   0,0     0    2136    644 ?        Ss   Mar05    0:31 init
root       2     0     0       0      0 ?        S<   Mar05    0:00 [kt]
root       3   0,0     0       0      0 ?        S<   Mar05    0:00 [mi]
root       4   0,0     0       0      0 ?        S<   Mar05    0:00 [ks]
root       5   0,0     0       0      0 ?        S<   05 Mart  0:06 [wa]
root       6   0,0     0       0      0 ?        S<   Mar05    0:36 [ev]
root       7   0,0     0       0      0 ?        S<   Mar05    0:00 [kh]
ve daha fazlası...
```

Bu seçenekler kümesi, her kullanıcıya ait işlemleri görüntüler. Başında tire işareti olmayan seçenekleri kullanmak, komutu "BSD stili" davranışıyla çağırır. Linux sürümündeki `ps`, çeşitli Unix uygulamalarında bulunan ps programının davranışını taklit edebilir. Tablo 10-2, en popüler BSD seçeneklerini gösterir.

Tablo 10-2: Popüler BSD Stili `ps` Seçenekleri

Seçenek	İşlev
`x`	Çalışan işlemlerimizi listeler.
`ax`	Çalışan tüm işlemleri listeler.
`w`	Tam komut adlarını dahil eder.
`u`	Ayrıntılı liste.

`aux` seçenekleriyle, Tablo 10-3'te gösterilen ek sütunlar elde edilir.

Tablo 10-3: BSD Stili `ps` Sütun Başlıkları

Başlık	Anlam
`KULLANICI`	Kullanıcı Kimliği. Bu, işlemin sahibidir.

%CPU	Yüzde olarak CPU kullanımı.
%MEM	Yüzde olarak bellek kullanımı.
VSZ	Sanal bellek boyutu.
RSS	Yerleşik set boyutu. Bu, işlemin kullandığı fiziksel bellek (RAM) miktarıdır (kilobayt cinsinden).
START	İşlemin başladığı zaman. 24 saatten uzun değerler için tarih kullanılır.
TIME	İşlem tarafından tüketilen CPU süresi.

Aşağıdaki örnekte gösterildiği gibi, bir komut argümanı olarak PID ekleyerek tek bir işlemin ayrıntılı bir anlık görüntüsünü oluşturmak da mümkündür.

```
[me@linuxbox ~]$ ps uw 44719
KULLANICI PID                  %CPU           STAT           ZAMAN KOMUTU
%MEM VSZ RSS TTY
ben  44719 0,0 0,0 13480 6492 puan/1         S  BAŞLANGIÇ   0:00 bash
```

`top` ile İşlemleri Dinamik Olarak Görüntüleme

`ps` komutu, makinenin ne yaptığını büyük ölçüde ortaya çıkarabilir, ancak `ps` komutunun yürütüldüğü anda makinenin durumunun sadece bir anlık görüntüsünü sağlar. Makinenin faaliyetlerini daha dinamik bir şekilde görmek için `top` komutunu kullanırız:

```
[me@linuxbox ~]$ top
```

`top` programı, sistem işlemlerini işlem etkinliğine göre sıralayarak sürekli olarak güncellenen (varsayılan olarak her üç saniyede bir) bir ekran görüntüsü sunar. *top* adı, `top` programının sistemdeki "en üstteki" işlemleri görmek için kullanılması gerçeğinden gelmektedir. `top` ekranı iki bölümden oluşur: ekranın üst kısmında sistem özeti, ardından CPU etkinliğine göre sıralanmış işlem tablosu:

```
top - 14:59:20 açık  6:30, 2 kullanıcı, ortalama yük: 0,07, 0,02, 0,00
Görevler: toplam     1 çalışıyor, 106 uyku modunda, 0            2 zombi
CPU(lar  0,7%us,  1,0%sy,  0,0%ni, 98,3% id, 0,0% wa, 0,0% hi, 0,0% si
): Mem:  319496k toplam,   314860k              4636k boş,          19392k
Takas:   875500k toplam,   kullanılmış,       726372k boş,          tampon
                           149128k                                 114676k
  PID KULLANICI  PR NI VIRT RES SHR S %CPU %MEM          ZAMAN+ KOMUT
 6244 ben        39  19 31752 3124 2188 S  6.3 1.0 16:24.42 trackerd
   11071         20   0 2304 1092  840 R  1.3 0.3     0:00.14 top
      ben        20   0 2700 1100  772 S  O ,7        0:03.66 dbus-dae
 6180 ben        20   0 20944 7248 6560 S  O ,7        2:51.38 multiloa
 6321 ben        20   0  104m 9668 5776 S  0,3        2:19.39 Xorg
   4955 kök      20   0  2976  528  476 S  0,0        0:03.14 init
      1 kök      15  -5     0    0    0     0,0        0:00,00 kthreadd
      2 kök      RT  -      0    0    0 S   0,0        0:00.00 göç
      3 kök      15  -5     0    0    0 S   0,0        0:00.72 ksoftirq
      4 kök      RT  -5     0    0    0 S   0,0        0:00.04 watchdog
      5 kök      15  -5     0    0    0 S   0,0        0:00.42 olaylar/0
      6 kök      15  -5     0    0    0 S   0,0        0:00.06 khelper
      7 kök      15  -5     0    0    0 S   0,0        0:01.08 kblockd/
     41 kök      15  -5     0    0    0 S   0,0        0:00.00 kseriod
     67 kök      20   0     0    0    0 S   0,0        0:01.62 pdflush
    114 kök      15  -5     0    0    0 S   0,0        0:02.44 kswapd0
    116 kök
```

Sistem özeti birçok yararlı bilgi içerir. Tablo 10-4 bir özet sunmaktadır.

Tablo 10-4: `En üstteki` Bilgi Alanları

Satır	Alan	Anlam
1	`üst`	Programın adı.
	`14:59:20`	Gün içindeki mevcut saat.
	`yukarı 6:30`	Buna *çalışma süresi* denir. Makinenin son başlatılmasından bu yana geçen süreyi ifade eder. Bu örnekte, sistem altı buçuk saattir çalışır durumda.
	`2 kullanıcı`	İki kullanıcı oturum açmıştır.
	`ortalama yük:`	*Ortalama yük*, çalışmayı bekleyen işlemlerin sayısını, yani çalışabilir durumda olan ve CPU'yu paylaşan işlemlerin sayısını ifade eder. Her biri farklı bir zaman aralığına ait üç değer gösterilir.

		farklı bir zaman dilimini temsil eder. İlki son 60 saniyenin ortalaması, ikincisi önceki 5 dakikanın ortalaması ve sonuncusu önceki 15 dakikanın ortalamasıdır. 1,0'dan küçük değerler, makinenin meşgul olmadığını gösterir.
2	`Görevler:`	Bu, işlemlerin sayısını ve çeşitli işlem durumlarını özetler.
3	`Cpu(lar):`	Bu satır, CPU'nun gerçekleştirdiği faaliyetlerin niteliğini tanımlar.
	`0,7%us`	CPU'nun %0,7'si *kullanıcı işlemleri* için kullanılıyor. Bu, çekirdek dışındaki işlemleri ifade eder.
	`1,0%sy`	CPU'nun %1,0'ı *sistem* (çekirdek) işlemleri için kullanılıyor.
	`0,0%ni`	CPU'nun %0,0'ı "nice" (düşük öncelikli) işlemler tarafından kullanılıyor.
	`98,3%id`	CPU'nun %98,3'ü boşta.
	`0,0%wa`	CPU'nun %0,0'ı I/O bekliyor.
4	`Mem:`	Bu, fiziksel RAM'in nasıl kullanıldığını gösterir.
5	`Takas:`	Bu, takas alanı (sanal bellek) nasıl kullanıldığını gösterir.

`Üst` program bir dizi klavye komutunu kabul eder. En ilginç olan ikisi `h`, programın yardım ekranını görüntüler ve `q`, `top'u` kapatır.

Her iki büyük masaüstü ortamı da `top'a` benzer bilgileri görüntüleyen grafiksel uygulamalar sunar (Windows'taki Görev Yöneticisi'nin çalıştığı şekilde), ancak `top` grafiksel sürümlerden daha iyidir çünkü daha hızlıdır ve çok daha az sistem kaynağı tüketir. Sonuçta, sistem izleme programımız, izlemeye çalıştığımız sistem yavaşlamasının kaynağı olmamalıdır.

İşlemleri Kontrol Etme

Artık süreçleri görebiliyor ve izleyebiliyoruz, şimdi de bunları kontrol etmeye başlayalım. Deneylerimiz için, `xlogo` adlı küçük bir programı denek olarak kullanacağız. `xlogo` programı, X Window System (ekranımızdaki grafikleri oluşturan temel motor, ancak Wayland lehine modası geçmeye başlamıştır

Wayland'ın lehine modası geçmek üzere olan) ile birlikte sağlanan örnek bir programdır ve X logosunu içeren yeniden boyutlandırılabilir bir pencere görüntüler. İlk olarak, test nesnemizi tanıyacağız.

```
[me@linuxbox ~]$ xlogo
```

Komutu girdikten sonra, logoyu içeren küçük bir pencere ekranın bir yerinde görünmelidir (Şekil 3). Bazı sistemlerde, `xlogo` bir uyarı mesajı görüntüleyebilir, ancak bu mesaj güvenle yok sayılabilir.

İpucu: Sisteminizde `xlogo` programı yoksa, bunun yerine `gedit` veya `kwrite programını` kullanmayı deneyin.

`xlogo'nun` çalıştığını, penceresinin boyutunu değiştirerek doğrulayabiliriz. Logo yeni boyutta yeniden çizilirse, program çalışıyor demektir.

Kabuk istemimizin geri dönmediğini fark ettiniz mi? Bunun nedeni, kabuğun programı bitirmesini beklemesi, tıpkı şimdiye kadar kullandığımız diğer tüm programlar gibi. `xlogo` penceresini kapatırsak, istem geri döner.

Şekil 3: xlogo programı

Bir Süreci Kesintiye Uğratma

`xlogo'yu` tekrar çalıştırdığımızda ne olduğunu gözlemleyelim. Önce, `xlogo` komutunu girin ve programın çalıştığını doğrulayın. Ardından, terminal penceresine dönün ve `Ctrl-c tuşlarına` basın.

```
[me@linuxbox ~]$ xlogo
[me@linuxbox ~]$
```

Terminalde `Ctrl-c tuşlarına` basmak, programı *keser*. Bu, programdan kibarca sonlandırılmasını istediğimiz anlamına gelir. `Ctrl-c tuşlarına` bastıktan sonra, `xlogo` penceresi kapandı ve kabuk istemine geri döndük.

Bu teknik kullanılarak birçok (ancak tümü değil) komut satırı programı kesilebilir.

Bir Süreci Arka Plana Alma

Diyelim ki `xlogo` programını sonlandırmadan kabuk istemini geri almak istiyoruz. Bunu, programı *arka plana* alarak yapabiliriz. Terminali, *ön plan* (kabuk istemcisi gibi yüzeyde görünen öğeler) ve arka plan (yüzeyin arkasında gizli olan öğeler) olarak düşünün. Bir programı hemen arka plana yerleştirmek için, komutu bir ve işareti (&) karakteri ile takip ederiz.

```
[me@linuxbox ~]$ xlogo &
[1] 28236
[me@linuxbox ~]$
```

Komutu girdikten sonra, `xlogo` penceresi belirdi ve kabuk istemcisi geri döndü, ancak bazı garip sayılar da yazdırıldı. Bu mesaj, *iş kontrolü* adı verilen bir kabuk özelliğinin parçasıdır. Bu mesajla, kabuk bize 1 numaralı işi (`[1]`) başlattığımızı ve bunun PID 28236 olduğunu söylüyor. `ps komutunu` çalıştırırsak, işlemimizi görebiliriz.

```
[me@linuxbox ~]$ ps
 PID TTY          ZAMAN CMD
10603 pts/1    00:00:00 bash
28236 pts/1    00:00:00 xlogo
28239 puan/1   00:00:00 ps
```

Kabuğun iş kontrol özelliği, terminalimizden başlatılan işleri listelememizi de sağlar. `jobs` komutunu kullanarak bu listeyi görebiliriz:

```
[me@linuxbox ~]$ jobs
[1]+ Çalışıyor                 xlogo &
```

Sonuçlar, 1 numaralı bir işin çalıştığını ve komutun `xlogo &` olduğunu göstermektedir.

Burada gösterilen kısayolu kullanarak arka planda birden fazla komut çalıştırabileceğimizi unutmayın:

```
me@linuxbox:~$ xlogo & gedit &
[1] 47211
[2] 47212
```

Bir Süreci Ön Plana Geri Getirme

Arka planda çalışan bir işlem, `Ctrl-c` ile kesintiye uğratma girişimleri dahil olmak üzere terminal klavye girişlerinden etkilenmez. Bir işlemi ön plana döndürmek için `fg` komutunu şu şekilde kullanın:

```
[me@linuxbox ~]$ jobs
[1]+ Çalıştırma                 xlogo &
[me@linuxbox ~]$ fg %1
xlogo
```

`fg` komutunun ardından yüzde işareti ve iş numarası (*jobspec* olarak adlandırılır) yazılır. Tek bir arka plan işimiz varsa, jobspec isteğe bağlıdır. `xlogo'yu` sonlandırmak için `Ctrl-c tuşlarına` basın.

Bir Süreci Durdurma (Askıya Alma)

Bazen bir işlemi sonlandırmadan durdurmak isteyebiliriz. Bu genellikle ön plandaki bir işlemin arka plana taşınması için yapılır. Ön plandaki bir işlemi durdurmak ve arka plana taşımak için `Ctrl-z tuşlarına` basın. Bunu deneyelim. Komut isteminde `xlogo` yazın, `ENTER tuşuna` basın ve ardından `Ctrl-z tuşlarına` basın:

```
[me@linuxbox ~]$ xlogo
[1]+ Durduruldu                 xlogo
[me@linuxbox ~]$
```

`xlogo'yu` durdurduktan sonra, `xlogo` penceresinin boyutunu değiştirmeye çalışarak programın durduğunu doğrulayabiliriz. Pencerenin boyutunu değiştirebileceğimizi, ancak logonun yeniden çizilmediğini ve programın durduğunu göreceğiz. Programın

ön planda yürütmeye devam edebiliriz veya `bg` komutuyla programın arka planda yürütülmesine devam edebiliriz:

```
[me@linuxbox ~]$ bg %1
[1]+ xlogo &
[me@linuxbox ~]$
```

fg komutunda olduğu gibi, tek bir iş varsa iş tanımı isteğe bağlıdır.

Bir işlemi ön plandan arka plana taşımak, komut satırından grafiksel bir program başlattığımızda, ancak sonuna `&` ekleyerek arka plana yerleştirmeyi unuttuğumuzda kullanışlıdır.

Neden komut satırından grafiksel bir program başlatmak isteyelim ki? Bunun iki nedeni vardır.

- Çalıştırmak istediğimiz program, pencere yöneticisinin menülerinde listelenmemiş olabilir (örneğin `xlogo`).
- Komut satırından bir programı başlatarak, program grafiksel olarak başlatıldığında görünmeyecek olan hata mesajlarını görebiliriz. Bazen, bir program grafiksel menüden başlatıldığında başlatılamaz. Bunun yerine komut satırından başlatarak, sorunu ortaya çıkaran bir hata mesajı görebiliriz. Ayrıca, bazı grafiksel programların ilginç ve kullanışlı komut satırı seçenekleri vardır.

İşlem Önceliklerini Değiştirme

ps komutunun çıktısında (ve `top` komutunda) gördüğümüz gibi, bir sürece verilen zamanlama önceliğini ifade eden *niceness* adlı bir süreç özniteliği vardır. Video kod dönüştürme veya CPU tabanlı ışın izleme gibi belirli durumlarda, bir sürece daha fazla öncelik (daha az niceness) vermek isteyebiliriz veya alternatif olarak, bir sürecin daha az CPU zamanı kullanmasını istiyorsak, ona daha fazla niceness verebiliriz. Niceness, `nice` ve renice komutlarıyla ayarlanabilir. Yalnızca süper kullanıcının bir işlemin önceliğini artırabileceğini ve normal kullanıcıların yalnızca kendilerine ait işlemlerin önceliğini azaltabileceğini unutmamak önemlidir.

`nice` komutu, belirtilen niceness ile bir işlemi başlatır. Niceness ayarlamaları -20 (en uygun) ile 19 (en uygun olmayan) arasında ifade edilir ve varsayılan değer sıfırdır (ayarlamasız). Bunun nasıl çalıştığını görelim. Normal 20'den daha düşük bir öncelikle çalıştırmak istediğimiz `cpu-hog` adlı bir programımız olduğunu varsayalım. Programı `nice` ile şu şekilde başlatabiliriz:

```
[me@linuxbox ~]$ nice -n 10 cpu-hog
```

Aynı şekilde, daha fazla CPU önceliği verilmesi gereken `must-run-fast` adlı bir programımız varsa, (süper kullanıcı olarak) şunu yapabiliriz:

```
[me@linuxbox ~]$ sudo nice -n -10 must-run-fast
```

Öncelik artırılmış bir komutu çalıştırmak nadiren gereklidir ve bunu yapmak, gerekli CPU zamanından mahrum kalma riskiyle karşı karşıya kalınmasına neden olur, bu nedenle dikkatli olun.

`renice` komutu, çalışan bir işlemin önceliğini ayarlar. Örneğin, `cpu-hog` programını başlattıktan sonra niceness değerini artırmak istersek, şunu yapabiliriz:

```
[me@linuxbox ~]$ ps
    PID TTY           ZAMAN CMD
 379087 pts/9     00:00:00 bash
 379215 pts/9     00:00:00 cpu-hog
 379223 puan/9    00:00:00 ps
[me@linuxbox ~]$ renice -n 19 379215
```

İlk olarak, çalışan `cpu-hog` programının işlem kimliğini belirlemek için `ps` komutunu çalıştırıyoruz, ardından istenen niceness seviyesi ve işlem kimliği ile `renice` komutunu çalıştırıyoruz. 19 (maksimum değer) niceness seviyesi, işlemlerin başka hiçbir şey beklemediğinde CPU döngülerini kullanmasını sağladığı için kullanışlıdır.

Sinyaller

`kill` komutu, sonlandırılması gereken (yani, bir tür duraklatma veya sonlandırma) programları sonlandırmak için kullanılır. İşte bir örnek:

```
[me@linuxbox ~]$ xlogo &
[1] 28401
[me@linuxbox ~]$ kill 28401
[1]+ Sonlandırıldı               xlogo
```

Önce `xlogo'yu` arka planda başlatıyoruz. Kabuk, iş spesifikasyonunu ve arka plan işleminin PID'sini yazdırır. Ardından, `kill` komutunu kullanarak sonlandırmak istediğimiz işlemin PID'sini belirtiriz. İş spesifikasyonunu kullanarak da işlemi belirtebilirdik (örneğin, %1).

`%1` gibi) kullanarak da belirtebilirdik.

Bu çok basit gibi görünse de, aslında bundan daha fazlası var. `Kill` komutu işlemleri tam olarak "öldürmez"; daha çok onlara *sinyal* gönderir. Sinyaller, işletim sisteminin programlarla iletişim kurmasının birkaç yolundan biridir. `Ctrl-c` ve `Ctrl-z` tuşlarını kullanarak sinyallerin nasıl çalıştığını daha önce görmüştük. Terminal bu tuşlardan birini aldığında, ön plandaki programa bir sinyal gönderir. `Ctrl-c` durumunda, `INT` (kesme) adlı bir sinyal gönderilir; `Ctrl-z` durumunda ise `TSTP` (terminal durdurma) adlı bir sinyal gönderilir. Programlar ise sinyalleri "dinler" ve alındıklarında bunlara göre hareket edebilir. Bir programın sinyalleri dinleyip bunlara göre hareket edebilmesi, programa sonlandırma sinyali gönderildiğinde devam eden işi kaydetme gibi işlemleri yapma olanağı verir.

`kill` ile İşlemlere Sinyal Gönderme

`kill` komutu, programlara sinyal göndermek için kullanılır. En yaygın sözdizimi şöyledir:

```
kill [-signal] PID...
```

Komut satırında herhangi bir sinyal belirtilmezse, varsayılan olarak `TERM` (sonlandırma) sinyali gönderilir. `kill` komutu en sık aşağıdaki sinyalleri göndermek için kullanılır:

Tablo 10-5: Yaygın Sinyaller

Sayı	Ad	Anlam
1	HUP	Hangup. Bu, terminallerin telefon hatları ve modemlerle uzak bilgisayarlara bağlı olduğu eski güzel günlerin bir kalıntısıdır. Bu sinyal, programlara kontrol terminali "kapandı" mesajını iletmek için kullanılır. Bu sinyalin etkisi, bir terminal oturumunu kapatarak gösterilebilir. Terminalde çalışan ön plan programı bu sinyali alır ve sonlandırılır. Bu sinyal, birçok arka plan programı tarafından yeniden başlatma işlemini gerçekleştirmek için de kullanılır. Bu, bir arka plan programına bu sinyal gönderildiğinde, programın yeniden başlatılacağı ve yapılandırma dosyasını yeniden okuyacağı anlamına gelir. Apache web sunucusu, `HUP` sinyalini bu şekilde kullanan bir arka plan programı örneğidir.

		Bir işlemi `nohup` komutuyla (bu bölümün ilerleyen kısımlarında ele alınacaktır) başlatarak `HUP` sinyaline karşı bağışık hale getirmek mümkündür.
2	`INT`	Kesme. Bu, terminalden gönderilen `Ctrl-c` ile aynı işlevi yerine getirir. Genellikle bir programı sonlandırır.
9	`KILL`	Kill. Bu sinyal özeldir. Programlar kendilerine gönderilen sinyalleri farklı şekillerde işlemeyi seçebilirler, hatta hepsini tamamen yok sayabilirler, ancak `KILL` sinyali hedef programa asla gönderilmez. Bunun yerine, çekirdek işlemi hemen sonlandırır. Bir işlem bu şekilde sonlandırıldığında, kendi ardından "temizlik" yapma veya çalışmasını kaydetme fırsatı verilmez. Bu nedenle, `KILL` sinyali yalnızca diğer sonlandırma sinyalleri başarısız olduğunda son çare olarak kullanılmalıdır.
15	`TERM`	Sonlandır. Bu, `kill` komutu tarafından gönderilen varsayılan sinyaldir. Bir program sinyalleri alacak kadar hala "canlı" ise, sonlandırılır.
18	`CONT`	Devam. Bu, `STOP` veya `TSTP` sinyalinden sonra bir işlemi geri yükler. Bu sinyal `bg` ve `fg` komutları tarafından gönderilir.
19	`STOP`	Durdur. Bu sinyal, bir işlemi sonlandırmadan duraklatır. `KILL` sinyali gibi, hedef işleme gönderilmez ve bu nedenle yok sayılamaz.
20	`TSTP`	Terminal durdurma. Bu, `Ctrl-z` tuşuna basıldığında terminal tarafından gönderilen sinyaldir. `STOP` sinyalinden farklı olarak, `TSTP` sinyali program tarafından alınır, ancak program bunu yok saymayı seçebilir.

`kill` komutunu deneyelim:

```
[me@linuxbox ~]$ xlogo &
[1] 13546
```

```
[me@linuxbox ~]$ kill -1 13546
[1]+ Bağlantı kesildi            xlogo
```

Bu örnekte, `xlogo` programını arka planda başlatıyoruz ve ardından `kill` ile bir `HUP` sinyali gönderiyoruz. `xlogo` programı sonlandırılıyor ve kabuk, arka plan işleminin bir hangup sinyali aldığını gösteriyor. Mesajın görünmesi için `ENTER tuşuna` birkaç kez basmamız gerekebilir. Sinyallerin, `SIG` harfleriyle başlayan isimler de dahil olmak üzere, numara veya isimle belirtilebileceğini unutmayın.

```
[me@linuxbox ~]$ xlogo &
[1] 13601
[me@linuxbox ~]$ kill -INT 13601
[1]+ Kesinti                     xlogo
[me@linuxbox ~]$ xlogo &
[1] 13608
[me@linuxbox ~]$ kill -SIGINT 13608
[1]+ Kesinti                     xlogo
```

Önceki örneği tekrarlayın ve diğer sinyalleri deneyin. Unutmayın, PID'lerin yerine iş özelliklerini de kullanabiliriz.

Dosyalarda olduğu gibi işlemlerin de sahipleri vardır ve `kill` ile sinyal göndermek için işlemin sahibi (veya süper kullanıcı) olmanız gerekir.

`kill` ile en sık kullanılan yukarıdaki sinyal listesine ek olarak, Tablo 10-5'te listelenen sistem tarafından sıkça kullanılan başka sinyaller de vardır.

Tablo 10-6: Diğer Yaygın Sinyaller

Numara	Ad	Anlam
3	QUIT	Bırak.
11	SEGV	Bölümleme ihlali. Bu sinyal, bir program belleği yasadışı bir şekilde kullandığında, yani yazma izni olmayan bir yere yazmaya çalıştığında gönderilir.
28	WINCH	Pencere değişikliği. Bu, bir pencerenin boyutu değiştiğinde sistem tarafından gönderilen sinyaldir. `Top` ve `less` gibi bazı programlar, yeni pencere boyutlarına uyacak şekilde kendilerini yeniden çizerek bu sinyale yanıt verir.

Merak edenler için, sinyallerin tam listesi aşağıdaki komutla görüntülenebilir:

```
[me@linuxbox ~]$ kill -l
```

Bir Süreci Askıya Alma Koruması Yapma

Yukarıda bahsettiğimiz gibi, birçok komut satırı programı, kontrol terminali "kapatıldığında" (yani kapatıldığında veya bağlantısı kesildiğinde) `HUP` sinyaline yanıt vererek sonlandırılır. Bu davranışı önlemek için, programı `nohup` komutuyla başlatabiliriz. İşte bir örnek.

```
[me@linuxbox ~]$ xlogo
```

`xlogo` programını tekrar başlatıp terminal penceremizi kapatırsak, `xlogo` programı, kontrol terminali kapatıldığında bir `HUP` sinyali gönderildiği için sonlandırılır. Bunu önlemek için `xlogo'yu nohup` komutuyla şu şekilde başlatabiliriz:

```
[me@linuxbox ~]$ nohup xlogo
```

Artık terminal penceresini kapattığımızda `xlogo` çalışmaya devam edecektir.

`killall` ile Birden Çok İşleme Sinyal Gönderme

`killall` komutunu kullanarak, belirli bir programa veya kullanıcı adına uyan birden fazla işleme sinyal göndermek de mümkündür. Komutun sözdizimi şöyledir:

```
killall [-u kullanıcı] [-sinyal] ad...
```

Bunu göstermek için, xlogo programının birkaç örneğini başlatıp sonra bunları sonlandıracağız.

```
[me@linuxbox ~]$ xlogo &
[1] 18801
[me@linuxbox ~]$ xlogo &
```

```
[2] 18802
[me@linuxbox ~]$ killall xlogo
[1]- Sonlandırıldı                xlogo
[2]+ Sonlandırıldı                xlogo
```

Unutmayın, `kill` komutunda olduğu gibi, bize ait olmayan işlemlere sinyal göndermek için süper kullanıcı ayrıcalıklarına sahip olmamız gerekir.

Sistemi Kapatma

Sistemi kapatma işlemi, sistemdeki tüm işlemlerin düzenli bir şekilde sonlandırılmasının yanı sıra, sistem kapanmadan önce bazı önemli bakım işlerinin (örneğin, tüm takılı dosya sistemlerinin senkronize edilmesi) gerçekleştirilmesini içerir. Bu işlevi gerçekleştirebilen dört komut vardır. Bunlar `halt`, `poweroff`, `reboot` ve `shutdown` komutlarıdır. İlk üçü oldukça açıklayıcıdır ve genellikle herhangi bir komut satırı seçeneği olmadan kullanılır. İşte bir örnek:

```
[me@linuxbox ~]$ sudo reboot
```

`Shutdown` komutu biraz daha ilginçtir. Bu komutla, hangi eylemin gerçekleştirileceğini (durdurma, kapatma veya yeniden başlatma) belirleyebilir ve kapatma işlemi için bir gecikme süresi belirleyebiliriz. Çoğu zaman sistemi durdurmak için şu şekilde kullanılır:

```
[me@linuxbox ~]$ sudo shutdown -h now
```

Sistemi yeniden başlatmak için şu şekilde kullanılır:

```
[me@linuxbox ~]$ sudo shutdown -r now
```

Gecikme çeşitli şekillerde belirtilebilir. Ayrıntılar için `shutdown` man sayfasına bakın. `shutdown` komutu yürütüldüğünde, oturum açmış tüm kullanıcılara yaklaşan olay hakkında uyarı içeren bir mesaj "yayınlanır".

İşlemlerle İlgili Diğer Komutlar

İşlemleri izlemek önemli bir sistem yönetimi görevi olduğundan, bunun için birçok komut vardır. Tablo 10-6, bunlardan bazılarını listelemektedir:

Tablo 10-7: Diğer İşlemle İlgili Komutlar

Komut	Açıklama
`pstree`	İşlemler arasındaki ana-alt ilişkilerini gösteren ağaç benzeri bir düzenle işlem listesini görüntüler.
`vmstat`	Bellek, takas ve disk G/Ç dahil olmak üzere sistem kaynak kullanımının anlık görüntüsünü çıkarır. Sürekli görüntüleme için, komutu güncellemeler için bir zaman gecikmesi (saniye cinsinden) ile takip edin. İşte bir örnek: `vmstat 5`. `Ctrl-c` ile çıktıyı sonlandırın.
`xload`	Zaman içinde sistem yükünü gösteren bir grafik çizen grafiksel bir programdır.
`tload`	xload programına benzer, ancak grafiği terminalde çizer. `Ctrl-c` ile çıktıyı sonlandırın.

Özet

Çoğu modern sistem, birden fazla işlemi yönetmek için bir mekanizmaya sahiptir. Linux, bu amaç için zengin bir araç seti sunar. Linux'un dünyanın en çok kullanılan sunucu işletim sistemi olduğu düşünüldüğünde, bu çok mantıklıdır. Ancak, diğer bazı sistemlerin aksine, Linux işlem yönetimi için öncelikle komut satırı araçlarına dayanır. Linux için grafiksel işlem araçları olsa da, komut satırı araçları hızları ve az yer kaplamaları nedeniyle büyük ölçüde tercih edilir. GUI araçları güzel görünebilir, ancak genellikle sistemde çok fazla yük oluştururlar, bu da bir şekilde amacını boşa çıkarır.

Bölüm 2 – Yapılandırma ve Ortam

11 – Ortam

Daha önce de bahsettiğimiz gibi, kabuk oturumu sırasında kabuk, *ortam* adı verilen bir bilgi kümesini tutar. Programlar, sistemin yapılandırmasıyla ilgili bilgileri belirlemek için ortamda depolanan verileri kullanır. Çoğu program, program ayarlarını depolamak için *yapılandırma dosyalarını* kullanırken, bazı programlar davranışlarını ayarlamak için ortamda depolanan değerleri de arar. Bunu bilerek, ortamları kabuk deneyimimizi özelleştirmek için kullanabiliriz.

Bu bölümde, aşağıdaki komutlarla çalışacağız:

- `printenv` – Ortamın bir kısmını veya tamamını yazdırır.
- `set` – Kabuk seçeneklerini ayarlar.
- `export` – Ortamı daha sonra yürütülecek programlara aktarır.
- `alias` – Bir komut için bir takma ad oluşturun.
- `source` – Geçerli kabukta bir dosyadan komutları çalıştırın.

Ortamda Neler Depolanır?

Kabuk, ortamda iki temel veri türü depolar; ancak `bash` ile bu türler ilk bakışta büyük ölçüde ayırt edilemez. Bunlar *ortam değişkenleri* ve *kabuk değişkenleridir*. Kabuk değişkenleri, `bash'ın` geçerli örneği tarafından oraya yerleştirilen veri parçalarıdır ve ortam değişkenleri ise diğer her şeydir. Değişkenlere ek olarak, kabuk bazı programatik verileri, yani *takma adları* ve *kabuk işlevlerini* depolar. Takma adları 5. bölümde ele aldık ve kabuk işlevlerini (kabuk betikleriyle ilgili olan) 4. bölümde ele alacağız.

Ortamı İnceleme

Çevrede neyin depolandığını görmek için `bash'ın` yerleşik `set` komutunu veya `printenv` programını kullanabiliriz. `set` komutu hem kabuk hem de çevre değişkenlerini gösterirken, `printenv` sadece çevre değişkenlerini gösterir. Çevre içeriğinin listesi oldukça uzun olacağından, her iki komutun çıktısını `less'e` yönlendirmek en iyisidir.

```
[me@linuxbox ~]$ printenv | lessBunu yaptığımızda, aşağıdaki gibi bir çıktı
almalıyız:
KULLANICI=ben
PAGER=less
LSCOLORS=Gxfxcxdxbxegedabagacad
XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/usr/share/upstart/xdg:/etc/xdg
PATH=/home/me/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/
sbin:/bin:/usr/games:/usr/local/games
DESKTOP_SESSION=ubuntu
QT_IM_MODULE=ibus
QT_QPA_PLATFORMTHEME=appmenu-qt5
JOB=dbus
PWD=/home/me
GNOME_KEYRING_PID=1850
LANG=en_US.UTF-8
GDM_LANG=en_US
MANDATORY_PATH=/usr/share/gconf/ubuntu.mandatory.path MASTER_HOST=linuxbox
IM_CONFIG_PHASE=1
COMPIZ_CONFIG_PROFILE=ubuntu
GDMSESSION=ubuntu
SESSIONTYPE=gnome-session
XDG_SEAT=seat0
HOME=/home/me
SHLVL=2
LANGUAGE=en_US
GNOME_DESKTOP_SESSION_ID=bu-kullanımdan-
kaldırıldı LESS=-R
LOGNAME=ben
COMPIZ_BIN_PATH=/usr/bin/
LC_CTYPE=en_US.UTF-8
XDG_DATA_DIRS=/usr/share/ubuntu:/usr/share/gnome:/usr/local/share/:/
usr/share/
QT4_IM_MODULE=xim
DBUS_SESSION_BUS_ADDRESS=unix:abstract=/tmp/dbus-IwaesmWaT0 LESSOPEN=|
/usr/bin/lesspipe %s
INSTANCE=
```

Gördüğümüz şey, ortam değişkenlerinin ve değerlerinin bir listesidir. Örneğin, `me` değerini içeren `USER` adlı bir değişken görüyoruz. `printenv` komutu, belirli bir değişkenin değerini de listeleyebilir.

```
[me@linuxbox ~]$ printenv USER me
```

`set` komutu, seçenekler veya argümanlar olmadan kullanıldığında, hem kabuk hem de ortam değişkenlerini ve tanımlanmış kabuk işlevlerini görüntüler. `printenv`'den farklı olarak, çıktısı alfabetik sırayla sıralanır.

```
[me@linuxbox ~]$ set | less
```

`Echo` komutunu kullanarak bir değişkenin içeriğini şu şekilde görüntülemek de mümkündür:

```
[me@linuxbox ~]$ echo $HOME
/home/me
```

Ne `set` ne de `printenv komutunun` göstermediği bir ortam öğesi de takma adlardır. Bunları görmek için, argüman olmadan `alias` komutunu girin.

```
[me@linuxbox ~]$ alias
alias l.='ls -d .* --color=auto'
alias ll='ls -l --color=auto' alias
ls='ls --color=auto'
alias vi='vim'
alias which='alias | /usr/bin/which --tty-only --read-alias --show-dot
--show-tilde'
```

Bazı İlginç Değişkenler

Ortamda oldukça fazla değişken bulunur ve ortam burada sunulan ortamdan farklı olsa da, ortamımızda Tablo 11-1'de listelenen değişkenleri muhtemelen göreceğiz.

Tablo 11-1: Ortam Değişkenleri

Değişken	İçerik
DISPLAY	Grafiksel bir ortam çalıştırıyorsak ekranın adı. Genellikle bu ":0"dır, yani X sunucusu tarafından oluşturulan ilk ekran anlamına gelir. Wayland ekran birleştiricisini çalıştıran sistemlerde

	`WAYLAND_DISPLAY` adlı ilgili bir değişken vardır.
`EDITOR`	Metin düzenleme için kullanılacak programın adı.
`SHELL`	Kullanıcının varsayılan kabuk programının adı.
`HOME`	Ana dizininizin yol adı.
`LANG`	İnsan dilimizin karakter kümesini ve sıralama düzenini tanımlar.
`OLDPWD`	Önceki çalışma dizini.
`PAGER`	Sayfalama çıktısı için kullanılacak programın adı. Genellikle `/usr/bin/less` olarak ayarlanır.
`PATH`	Yürütülebilir bir programın adını girdiğimizde aranan, iki nokta üst üste ile ayrılmış dizinlerin listesi.
`PS1`	"prompt string 1" (komut istemi dizesi 1) anlamına gelir. Bu, kabuk komut isteminin içeriğini tanımlar. 13. bölümde göreceğimiz gibi, bu kapsamlı bir şekilde özelleştirilebilir.
`PWD`	Geçerli çalışma dizini.
`TERM`	Terminal türünüzün adı. Unix benzeri sistemler birçok terminal protokolünü destekler; bu değişken, terminal emülatörümüzle kullanılacak protokolü ayarlar.
`TZ`	Saat dilimimizi belirtir. Unix benzeri sistemler, bilgisayarın dahili saatini *Koordineli Evrensel Saat* (UTC) ile tutar ve ardından bu değişkenle belirtilen farkı uygulayarak yerel saati görüntüler.
`KULLANICI`	Kullanıcı adımız.

Bu değerlerin bazıları eksik olsa da endişelenmeyin. Dağıtımlara göre değişiklik gösterirler.

Ortam Nasıl Oluşturulur?

Sisteme giriş yaptığımızda, `bash` programı başlar ve tüm kullanıcılar tarafından paylaşılan varsayılan ortamı tanımlayan bir dizi yapılandırma komut dosyası olan *başlangıç dosyalarını* okur. Bunu, kişisel ortamımızı tanımlayan ana dizindeki diğer başlangıç dosyaları izler. Tam sırası, başlatılan kabuk oturumunun türüne bağlıdır. İki tür vardır.

- **Giriş kabuğu oturumu** Giriş kabuğu oturumu, kullanıcı adımızı ve şifremizi girmemizi isteyen oturumdur. Bu, örneğin grafiksel bir ortama giriş yaptığımızda gerçekleşir.

. Ayrıca, sanal konsol oturumu başlattığımızda da yapılır.

- **Giriş yapmayan kabuk oturumu** Giriş yapmayan kabuk oturumu genellikle terminal emülatörümüzle GUI'de bir terminal oturumu başlattığımızda gerçekleşir.

Giriş kabukları, Tablo 11-2'de gösterildiği gibi bir veya daha fazla başlangıç dosyasını okur.

Tablo 11-2: Oturum Açma Kabuğu Oturumları için Başlangıç Dosyaları

Dosya	İçerik
`/etc/profile`	Tüm kullanıcılara uygulanan genel bir yapılandırma betiği.
`~/.bash_profile` ayarları genişletmek	Kullanıcının kişisel başlangıç dosyası. Bu, genel yapılandırma betiğindeki veya genel yapılandırma betiğindeki ayarları geçersiz kılmak için kullanılabilir.
`~/.bash_login`	`~/.bash_profile` bulunmazsa, `bash` bu komut dosyasını okuyun.
`~/.profile`	Ne `~/.bash_profile` ne de `~/.bash_login` bulunmazsa, `bash` bu dosyayı okumaya çalışır. Bu, Ubuntu gibi Debian tabanlı dağıtımlarda varsayılan ayardır.

Giriş yapmayan kabuk oturumları, Tablo 11-3'te listelenen başlangıç dosyalarını okur.

Tablo 11-3: Oturum Açma Kabuğu Oturumları için Başlangıç Dosyaları

Dosya	İçerik
`/etc/bash.bashrc`	Tüm kullanıcılara uygulanan genel bir yapılandırma komut dosyası.
`~/.bashrc`	Kullanıcının kişisel başlangıç dosyası. Küresel yapılandırma betiğindeki ayarları genişletmek veya geçersiz kılmak için kullanılabilir. genel yapılandırma betiğindeki ayarları geçersiz kılmak için kullanılabilir.

Tablo 11-3'teki başlangıç dosyalarını okumaya ek olarak, oturum açma kabuğu olmayan kabuklar, genellikle bir oturum açma kabuğu olan üst işlemlerinden ortam değişkenlerini devralır.

Bir göz atın ve bu başlangıç dosyalarından hangilerinin yüklü olduğunu görün. Unutmayın — önceki dosya adlarının çoğu nokta ile başladığından (yani gizli olduklarından), `ls komutunu` kullanırken "-a" seçeneğini kullanmamız gerekecektir.

`~/.bashrc` dosyası, neredeyse her zaman okunduğu için sıradan kullanıcıların bakış açısından muhtemelen en önemli başlangıç dosyasıdır. Oturum açma olmayan kabuklar varsayılan olarak bu dosyayı okur ve oturum açma kabukları için çoğu başlangıç dosyası da `~/.bashrc` dosyasını okuyacak şekilde yazılmıştır.

Başlangıç Dosyasında Neler Vardır?

Tipik bir `.bash_profile` dosyasının içine bakarsak, şuna benzer bir şey görürüz:

```
# .bash_profile

# Takma adları ve işlevleri al if
[ -f ~/.bashrc ]; then
        . ~/.bashrc
fi

# Kullanıcıya özgü ortam ve başlangıç programları

PATH=$PATH:$HOME/bin
export PATH
```

"#" ile başlayan satırlar *yorum*lardır ve kabuk tarafından yürütülmez. Bunlar, insanların okuyabilmesi için vardır. İlk ilginç şey dördüncü satırda, aşağıdaki kodla ortaya çıkar:

```
if [ -f ~/.bashrc ]; then
        . ~/.bashrc
fi
```

Bu, *if bileşik komutu* olarak adlandırılır ve 4. Bölümde kabuk betiklerine geldiğimizde ayrıntılı olarak ele alacağız, ancak şimdilik çevirisi şöyledir:

```
"~/.bashrc" dosyası varsa, "~/.bashrc"
    dosyasını okuyun.
```

Bu kod parçasının, bir oturum açma kabuğunun `.bashrc` dosyasının içeriğini nasıl aldığını gösterdiğini görebiliriz. Başlangıç dosyamızdaki bir sonraki şey, `PATH` değişkeniyle ilgilidir.

Komut satırına komutları girdiğimizde kabuğun komutları nerede bulacağını nasıl bildiğini hiç merak ettiniz mi? Örneğin, `ls komutunu` girdiğimizde, kabuk `/bin/ls` (`ls` komutunun tam yol adı) bulmak için tüm bilgisayarı aramaz; bunun yerine, `PATH` değişkeninde bulunan dizinlerin listesini arar.

`PATH` değişkeni genellikle (dağıtımına bağlı olarak her zaman değil) `.bash_profile` başlangıç dosyası tarafından şu kodla ayarlanır:

```
PATH=$PATH:$HOME/bin
```

PATH, listenin sonuna $HOME/bin dizinini eklemek için değiştirilir. Bu, 7. bölümde değindiğimiz parametre genişletmesinin bir örneğidir. Bunun nasıl çalıştığını göstermek için aşağıdakileri deneyin:

```
[me@linuxbox ~]$ foo="Bu bir " [me@linuxbox ~]$
echo $foo
Bu bir
[me@linuxbox ~]$ foo=$foo"metin."
[me@linuxbox ~]$ echo $foo
Bu bir metindir.
```

Bu tekniği kullanarak, bir değişkenin içeriğinin sonuna metin ekleyebiliriz.

PATH değişkeninin içeriğinin sonuna $HOME/bin dizesini ekleyerek, $HOME/bin dizini bir komut girildiğinde aranan dizinler listesine eklenir. Bu, kendi özel programlarımızı depolamak için ana dizinde bir dizin oluşturmak istediğimizde, kabuğun bunu yapmaya hazır olduğu anlamına gelir. Tek yapmamız gereken ona bin adını vermek ve hazırız.

Not: Birçok dağıtım bu PATH ayarını varsayılan olarak sağlar. Ubuntu gibi Debian tabanlı dağıtımlar, oturum açıldığında ~/bin dizinini kontrol eder ve dizin bulunursa PATH değişkenine dinamik olarak ekler.

Son olarak, şunu elde ederiz:

```
export PATH
```

export komutu, kabuğa PATH içeriğini bu kabuğun alt süreçlerine kullanılabilir hale getirmesini söyler. Bir anlamda, kabuk değişkenini bir ortam değişkenine dönüştürür.

Alt Süreçlerin Ortamlarını Nasıl Miras Aldıklarını Keşfetmek

Bu son nokta biraz daha ayrıntılı olarak ele alınmayı hak ediyor. Kabuk değişkenleri, kabuğun geçerli örneğine özeldir ve kabuğun başlattığı alt süreçlere kopyalanmaz. Bunu gösterelim.

İlk olarak, mevcut kabuğumuzda bir kabuk değişkeni ayarlayacağız.

```
[me@linuxbox ~]$ foo="bar"
```

Ardından, kabuğun başka bir kopyasını başlatacağız.

```
[me@linuxbox ~]$ bash
[me@linuxbox ~]$
```

Şu anda hiçbir şey olmamış gibi görünüyor, ancak aslında kabuğun başka bir örneğini çalıştırıyoruz. Bunu `ps` komutunun sonuçlarına bakarak görebiliriz.

```
[me@linuxbox ~]$ ps
    PID TTY          ZAMAN CMD
1011638 pts/9     00:00:00 bash
1011650 pts/9     00:00:00 bash
1011662 puan/9    00:00:00 ps
```

Burada iki kopya `bash` çalıştırdığımızı görüyoruz. Yeni örneği başlattığımızda arka plana almadığımız için, şimdi ön planda çalışan görev bu ve orijinal örnek bu yeni kabuğun bitmesini bekliyor. Şimdi az önce ayarladığımız `foo` değişkeninin değerini görüntülemeye çalışalım.

```
[me@linuxbox ~]$ echo $foo

[me@linuxbox ~]$
```

Sonuç olmaması, `foo` değişkeninin boş olduğunu gösterir. Bunun nedeni, bu kabuk örneğinde ona bir değer vermemiş olmamızdır. Kabuk değişkenleri, alt işlem oluşturulduğunda kopyalanıp alt işleme verilmez. Öte yandan, ortam değişkenleri kopyalanarak alt işlemin ortamı haline gelir.

İşlemlerin ve ortamlarının bir başka eğlenceli özelliğini göstermek için, bu kabuk örneğinde `foo'yu` tanımlayalım.

```
[me@linuxbox ~]$ foo="barbar"
```

Ardından, bu `bash` örneğinden çıkıp, arka planda çalıştırmadığımız diğer programlarda olduğu gibi, alt işlemin sonlanmasını sabırla bekleyen üst örneğe geri döneceğiz.

```
[me@linuxbox ~]$ exit
[me@linuxbox ~]$
```

`ps komutunu` tekrar çalıştırarak ilk örneğe geri döndüğümüzü göreceğiz.

```
[me@linuxbox ~]$ ps
   PID TTY          ZAMAN CMD
1011638 pts/9     00:00:00 bash
1014900 pts/9     00:00:00 ps
```

Şimdi bu örnekte `foo` değerine bakalım.

```
[me@linuxbox ~]$ echo $foo bar
[me@linuxbox ~]$
```

Gördüğümüz gibi, hala ona verdiğimiz değer var, alt örnekte ayarladığımız yeni değer değil. Bu, alt süreçlerle ilgili önemli bir kuralı gösterir: *bir alt süreç*, *üst sürecinin ortamını değiştiremez*. Bir alt süreç sona erdiğinde, ortamını da beraberinde götürür. Bu gerçek, kabuk betikleri yazmaya başladığımızda önemli hale gelecektir.

Geçici Ortamla Bir Programı Başlatma

Kabuğun sağladığı bir başka kullanışlı özellik, bir komutu çalıştırıp ona geçici bir ortam değişkeni atayabilmesidir. Bazen bir programı çalıştırıp ona özel bir ortam değeri atamak isteriz. Bunun iyi bir örneği, `man` komutudur. Bu komut, `man'a` çıktısını ne kadar genişlikte biçimlendireceğini söyleyen `MANWIDTH` adlı bir ortam değişkenini arar. Örneğin, `man`'ın çıktısını en fazla 75 karakter genişliğinde biçimlendirmesi için (kolay okuma için kullanışlı bir ayar) şunu yapabiliriz:

```
[me@linuxbox ~]$ MANWIDTH=75 man ls
```

Bu, `ls` komutunun man sayfasını rahat bir genişlikte güzelce biçimlendirilmiş olarak görüntüler. Bu arada, bu bir alias olarak kullanmak için iyi bir şeydir.

```
[me@linuxbox ~]$ alias man='MANWIDTH=75 man'
```

Artık `man` komutunu kullandığımızda satır uzunluğu her zaman 75 karakterle sınırlandırılacaktır. Daha gelişmiş geçici ortamlar için `env` komutuna bakın.

Ortamı Değiştirme

Başlangıç dosyalarının nerede olduğunu ve ne içerdiğini bildiğimiz için, ortamımızı özelleştirmek için bunları değiştirebiliriz.

Hangi Dosyaları Değiştirmeliyiz?

Genel bir kural olarak, `PATH'inize` dizinler eklemek veya ek ortam değişkenleri tanımlamak için, bu değişiklikleri `.bash_profile` dosyasına (veya dağıtımınıza göre eşdeğerine; örneğin, Ubuntu `.profile` kullanır) yerleştirin. Diğer her şey için, değişiklikleri `.bashrc` dosyasına yerleştirin.

Not: Sistem yöneticisi değilseniz ve sistemdeki tüm kullanıcılar için varsayılanları değiştirmeniz gerekmiyorsa, değişikliklerinizi ev dizininizdeki dosyalarla sınırlandırın. `/etc` dizinindeki `profile` gibi dosyaları değiştirmek elbette mümkündür ve çoğu durumda bunu yapmak mantıklı olabilir, ancak şimdilik güvenli olanı yapalım.

Metin Editörleri

Kabuğun başlangıç dosyalarını ve sistemdeki diğer yapılandırma dosyalarının çoğunu düzenlemek (yani değiştirmek) için *metin düzenleyici* adlı bir program kullanırız. Metin düzenleyici, bazı yönlerden kelime işlemciye benzer bir programdır, çünkü hareketli bir imleçle ekrandaki kelimeleri düzenlememize olanak tanır. Kelime işlemciden farklı olarak, yalnızca düz metni destekler ve genellikle program yazmak için tasarlanmış özellikler içerir. Metin editörleri, yazılım geliştiricilerin kod yazmak için ve sistem yöneticilerinin sistemi kontrol eden yapılandırma dosyalarını yönetmek için kullandıkları temel araçlardır.

Linux için birçok farklı metin editörü mevcuttur; çoğu sistemde birkaç tane yüklüdür. Neden bu kadar çok farklı editör var? Çünkü programcılar bunları yazmayı severler ve programcılar bunları yoğun olarak kullandıkları için, nasıl çalışması gerektiğine dair kendi isteklerini ifade etmek için editörler yazarlar.

Metin editörleri iki temel kategoriye ayrılır: grafiksel ve metin tabanlı. GNOME ve KDE, bazı popüler grafiksel editörler içerir. GNOME, genellikle GNOME menüsünde "Metin Editörü" olarak adlandırılan `gedit` adlı bir editörle birlikte gelir. KDE genellikle `kwrite` ve daha gelişmiş olan `kate` olmak üzere iki editörle birlikte gelir.

Birçok metin tabanlı editör vardır. Sıkça karşılaştığımız popüler olanlar `nano`, `vi` ve `emacs'tır`. `Nano` editörü, PINE e-posta paketi ile birlikte gelen `pico` editörünün yerine tasarlanmış, basit ve kullanımı kolay bir editördür. `vi` editörü (çoğu Linux sisteminde "vi improved"ın kısaltması olan `vim` adlı bir programla değiştirilmiştir), Unix benzeri sistemler için geleneksel bir editördür. Bu editör, bir sonraki bölümümüzün konusu olacaktır.

emacs editörü, orijinal olarak Richard Stallman tarafından yazılmıştır. Devasa, çok amaçlı, her şeyi yapan bir programlama ortamıdır. Her ne kadar kolayca elde edilebilir olsa da, çoğu Linux sisteminde varsayılan olarak nadiren yüklenir.

Metin Düzenleyiciyi Kullanma

Metin editörleri, komut satırından editörün adını ve ardından düzenlemek istediğimiz dosyanın adını yazarak çağrılır. Dosya henüz mevcut değilse, editör yeni bir dosya oluşturmak istediğimizi varsayar. İşte gedit kullanarak bir örnek:

```
[me@linuxbox ~]$ gedit some_file
```

Bu komut, gedit metin düzenleyicisini başlatır ve varsa "some_file" adlı dosyayı yükler.

Grafiksel metin editörleri oldukça açıklayıcıdır, bu yüzden burada bunlara değinmeyeceğiz. Bunun yerine, ilk metin tabanlı metin editörümüz olan nano'ya odaklanacağız. Nano'yu çalıştırıp .bashrc dosyasını düzenleyelim. Ancak bunu yapmadan önce, biraz "güvenli bilgisayar kullanımı" pratiği yapalım. Önemli bir yapılandırma dosyasını düzenlediğimizde, her zaman önce dosyanın yedek bir kopyasını oluşturmak iyi bir fikirdir. Bu, düzenleme sırasında dosyayı bozarsak bizi korur. .bashrc dosyasının yedeğini oluşturmak için şunu yapın:

```
[me@linuxbox ~]$ cp .bashrc .bashrc.bak
```

Yedek dosyasına ne isim verdiğimiz önemli değildir; anlaşılır bir isim seçin yeter. ".bak", ".sav", ".old" ve ".orig" uzantıları, yedek dosyasını belirtmek için yaygın olarak kullanılan yöntemlerdir. Unutmayın, cp komutu *mevcut dosyaları* sessizce *üzerine yazar*.

Artık bir yedekleme dosyamız olduğuna göre, düzenleyiciyi başlatacağız.

```
[me@linuxbox ~]$ nano .bashrc
```

nano başlatıldığında, şöyle bir ekran göreceğiz:

```
  GNU nano 8.6                Dosya:
                              .bashrc
# .bashrc

# Kaynak global tanımları
```

```
if [ -f /etc/bashrc ]; then
        . /etc/bashrc
fi

# Kullanıcıya özgü takma adlar ve işlevler









                                [ 8 satır oku ]
^G Yardım Al^O Yaz^R Dosya Oku^Y Önceki Sayfa^K Metni Kes^C Geçerli Konum
^X Çıkış  ^J Hizala ^W Nerede^V Sonraki Sayfa^U Kesmeyi Kaldır^T Yazım
Denetimi
```

Not: Sisteminizde `nano` yüklü değilse, bunun yerine grafiksel bir düzenleyici kullanabilirsiniz.

Ekranın üst kısmında bir başlık, ortasında düzenlenmekte olan dosyanın metni ve alt kısmında bir komut menüsü bulunur. `Nano`, e-posta istemcisiyle birlikte gelen metin düzenleyicinin yerini almak üzere tasarlanmış olduğundan, düzenleme özellikleri oldukça sınırlıdır.

Herhangi bir metin düzenleyicide öğrenmemiz gereken ilk komut, programdan nasıl çıkılacağıdır. `Nano'da` çıkmak için `Ctrl-x tuşlarına` basarız. Bu, ekranın altındaki menüde gösterilir. `^X` notasyonu `Ctrl-x` anlamına gelir. Bu, birçok programda kullanılan kontrol karakterleri için yaygın bir notasyonudur.

Bilmeniz gereken ikinci komut, çalışmanızı nasıl kaydedeceğinizdir. `Nano` ile bu komut `Ctrl-o` tuşlarıdır. Bu bilgiyle, düzenleme yapmaya hazırız. Aşağı ok ve/veya `PageDown` tuşlarını kullanarak imleci dosyanın sonuna getirin ve ardından `.bashrc` dosyasına ekleyin:

```
umask 0002
export HISTSIZE=1000
alias l.='ls -d .* --color=auto'
alias ll='ls -l --color=auto'
```

Not: Dağıtımınızda bunlardan bazıları zaten mevcut olabilir, ancak yinelemeler herhangi bir zarara yol açmaz.

Tablo 11-4, eklemelerimizin anlamını ayrıntılı olarak göstermektedir:

Tablo 11-4: `.bashrc` Dosyamızdaki Eklemeler

Satır	Anlam
`umask 0002`	9. Bölümde ele aldığımız paylaşımlı dizinlerle ilgili sorunu çözmek için `umask` ayarını yapar.
`export HISTCONTROL=ignoredups`	Aynı komut az önce kaydedilmişse, kabuğun geçmiş kaydetme özelliğinin komutu yok saymasına neden olur.
`export HISTSIZE=1000`	Komut geçmişinin boyutunu normalde varsayılan olan 500 satırdan 1.000 satıra çıkarır.
`alias l.='ls -d .* --color=auto'`	`l.` adlı yeni bir komut oluşturur ve nokta ile başlayan tüm dizin girişlerini görüntüler.
`alias ll='ls -l --color=auto'`	Uzun biçimli dizin listesini görüntüleyen `ll` adlı yeni bir komut oluşturur.

Gördüğümüz gibi, eklemelerimizin çoğu sezgisel olarak açık değildir, bu nedenle insanlara açıklamak için `.bashrc` dosyamıza bazı yorumlar eklemek iyi bir fikir olabilir. Editörü kullanarak eklemelerimizi şu şekilde değiştirin:

```
# Dizin paylaşımını kolaylaştırmak için umask'ı değiştirin
umask 0002

# Komut geçmişindeki yinelemeleri yok say ve # geçmiş
boyutunu 1000 satıra çıkar
export HISTCONTROL=ignoredups
export HISTSIZE=1000

# Bazı yararlı takma adlar ekle
alias l.='ls -d .* --color=auto'
alias ll='ls -l --color=auto'
```

Ah, çok daha iyi! Değişikliklerimizi tamamladıktan sonra, `Ctrl-o` tuşlarına basarak değiştirilen `.bashrc` dosyasını kaydetmek için Ctrl-o tuşlarına basın ve `nano'dan` çıkmak için `Ctrl-x` tuşlarına basın.

Yorumlar Neden Önemlidir?

Yapılandırma dosyalarını her değiştirdiğinizde, değişikliklerinizi belgelemek için bazı yorumlar eklemek iyi bir fikirdir. Elbette, yarın neyi değiştirdiğinizi hatırlayacaksınız, ama altı ay sonra ne olacak? Kendinize bir iyilik yapın ve bazı yorumlar ekleyin. Bunu yaparken, yaptığınız değişikliklerin bir kaydını tutmak da fena bir fikir değildir.

Shell betikleri ve `bash` başlangıç dosyaları, yorum başlangıcı için "#" sembolünü kullanır. Diğer yapılandırma dosyaları başka semboller kullanabilir. Çoğu yapılandırma dosyasında yorumlar bulunur. Bunları kılavuz olarak kullanın.

Yapılandırma dosyalarında, etkilenen program tarafından kullanılmalarını önlemek için *yorumlanmış* satırlar sıklıkla görürsünüz. Bu, okuyucuya olası yapılandırma seçenekleri veya doğru yapılandırma sözdizimi örnekleri için öneriler sunmak amacıyla yapılır. Örneğin, Ubuntu 24.04'ün `.bashrc` dosyası şu satırları içerir:

```
# bazı diğer ls takma
adları #alias ll='ls -
alF' #alias la='ls -A'
#alias l='ls -CF'
```

Son üç satır, yorumlanmış geçerli takma ad tanımlarıdır. Bu üç satırdan baştaki "#" sembollerini kaldırırsanız (*yorum kaldırma* adı verilen bir teknik), takma adları etkinleştirirsiniz. Tersine, bir satırın başına "#" sembolü eklerseniz

satırın başına bir "#" sembolü eklerseniz, içerdiği bilgileri koruyarak bir yapılandırma satırını devre dışı bırakabilirsiniz.

Değişikliklerimizi Etkinleştirme

`.bashrc` dosyasında yaptığımız değişiklikler, terminal oturumumuzu kapatıp yeni bir oturum başlatana kadar etkili olmaz, çünkü `.bashrc` dosyası yalnızca oturumun başında okunur. Ancak, aşağıdaki komutla `bash`'ı değiştirilen `.bashrc` dosyasını yeniden okumaya zorlayabiliriz:

```
[me@linuxbox ~]$ source ~/.bashrc
```

Bunu yaptıktan sonra, değişikliklerimizin etkisini görebilmeliyiz. Yeni takma adlardan birini deneyin.

```
[me@linuxbox ~]$ ll
```

Kaynak Hakkında Biraz Daha Fazla Bilgi

`Kaynak` komutu (kısaltması `.` olabilir), bir dosyayı içeriği klavyeden girilmiş gibi doğrudan geçerli kabuğa okuyan bir kabuk yerleşik komutudur. Evet, kabuk başlangıç dosyalarında gördüğümüz tüm o garip görünümlü şeyler, kabuğun anladığı ve üzerinde işlem yapabildiği şeylerdir. Birçok eski metin tabanlı işletim sistemi (DOS, CP/M vb.) esas olarak basit program başlatıcıları olarak işlev görürdü. Unix tarzı kabuklar da elbette bunu yapabilir, ancak çok daha fazlasını da yapabilirler.

Özet

Bu bölümde, metin düzenleyiciyle yapılandırma dosyalarını düzenleme gibi temel bir beceriyi öğrendik. İleride, komutların man sayfalarını okurken, komutların desteklediği ortam değişkenlerine dikkat edin. Birkaç değerli bilgi bulabilirsiniz. Sonraki bölümlerde, `bash` başlangıç dosyalarına ekleyerek özel komutlarınızın arzenalinize ekleyebileceğiniz güçlü bir özellik olan kabuk işlevlerini öğreneceğiz.

Daha Fazla Okuma

- `Bash` man sayfasının INVOCATION bölümü, `bash` başlangıç dosyalarını ayrıntılı olarak ele almaktadır.

12 – vi(m) 'ye Nazik Bir Giriş

New York'u ziyaret eden birinin, şehrin ünlü klasik müzik salonuna giden yolu bir yoldan geçen kişiye sorduğu eski bir fıkra vardır:

Ziyaretçi: Affedersiniz, Carnegie Hall'a nasıl giderim? Yoldan

geçen: Pratik, pratik, pratik!

Linux komut satırını öğrenmek, başarılı bir piyanist olmak gibi, bir öğleden sonra öğrenebileceğimiz bir şey değildir. Yıllarca pratik yapmak gerekir. Bu bölümde, Unix geleneğinin temel programlarından biri olan `vi` (telaffuz: "vee eye") metin editörünü tanıtacağız. `vi`, benzersiz kullanıcı arayüzüyle biraz kötü şöhretlidir, ancak bir ustanın klavyenin başına oturup "çalmaya" başladığını gördüğümüzde, gerçekten de büyük bir sanat eserine tanık olacağız. Bu bölümde usta olmayacağız, ancak bittiğinde `vi'de` "chopsticks"i nasıl çalacağımızı bileceğiz.

Neden vi'yi Öğrenmeliyiz

Grafiksel editörlerin ve `nano` gibi kullanımı kolay metin tabanlı editörlerin olduğu bu modern çağda, neden `vi` öğrenmeliyiz? Bunun üç iyi nedeni var.

- `vi` neredeyse her zaman kullanılabilir. Bu, uzak sunucu veya bozuk GUI yapılandırmasına sahip yerel sistem gibi grafik arayüzü olmayan bir sistemimiz varsa hayat kurtarıcı olabilir. `nano`, giderek popüler hale gelse de, hala evrensel değildir. Unix sistemlerinde program uyumluluğu için bir standart olan POSIX, `vi'nin` mevcut olmasını gerektirir.
- `vi` hafiftir ve hızlıdır. Birçok görev için, menülerde grafiksel metin editörünü bulup birkaç megabaytlık yüklemeyi beklemektense `vi'yi` açmak daha kolaydır. Ayrıca, `vi` yazma hızı için tasarlanmıştır. Göreceğimiz gibi, yetenekli `vi` kullanıcıları düzenleme sırasında parmaklarını klavyeden hiç kaldırmak zorunda kalmazlar.
- Diğer Linux ve Unix kullanıcılarının bizim korkak olduğumuzu

düşünmesini istemiyoruz. Tamam, belki iki iyi neden var.

Biraz Arka Plan

Vi'nin ilk sürümü 1976 yılında, daha sonra Sun Microsystems'ın kurucularından biri olan Berkeley Üniversitesi öğrencisi Bill Joy tarafından yazıldı. Vi, adını "visual" (görsel) kelimesinden almıştır, çünkü hareketli bir imleçle video terminalinde düzenleme yapmayı amaçlamaktaydı. *Görsel editörlerden* önce, tek seferde tek bir metin satırında çalışan *satır editörleri* vardı. Bir değişikliği belirtmek için, satır düzenleyicisine belirli bir satıra gitmesini ve metin ekleme veya silme gibi yapılacak değişikliği tanımlamasını söyleriz. Video terminallerin (teletype gibi yazıcı tabanlı terminallerin yerine) ortaya çıkmasıyla görsel düzenleme mümkün hale geldi. vi aslında ex adlı güçlü bir satır düzenleyicisini içerir ve vi kullanırken satır düzenleme komutlarını kullanabiliriz.

Çoğu Linux dağıtımı gerçek vi'yi içermez; bunun yerine, Bram Moolenaar tarafından yazılan vim ("vi improved"ın kısaltması) adlı gelişmiş bir yedekle birlikte gelir. vim, geleneksel Unix vi'ye göre önemli bir gelişmedir ve genellikle Linux sistemlerinde vi adına sembolik olarak bağlanır (veya takma ad verilir). Aşağıdaki tartışmalarda, vi adlı programın aslında vim olduğunu varsayacağız.

vi'yi Başlatma ve Durdurma

vi'yi başlatmak için, aşağıdakileri giriyoruz:

```
[me@linuxbox ~]$ vi
```

Bunun gibi bir ekran görünmelidir:

```
~
~
~                          VIM - Vi Improved
~
~                           sürüm 8.0.707
~                       Bram Moolenaar ve diğerleri
~                       tarafından.
~             Vim açık kaynaklıdır ve ücretsiz olarak
~             dağıtılabilir
~           yazın  :help sponsor<Enter>     bilgi için
~                        Vim geliştirilmesini destekleyin!
~           yazı   :q<Enter>                çıkmak için
~           n      :yardım<Enter> veya <F1>  çevrimiçi yardım
~           yazı   :yardım sürüm8<Enter>     için sürüm
~           n                                bilgisi
            yazın
```

```
~                    Vi uyumlu modda çalıştırma
~           :set nocp<Enter> yazın          Vim varsayılanları için
~           bu konuda bilgi için :help cp-default<Enter>
~           yazın
~
~
```

Daha önce `nano` ile yaptığımız gibi, öğrenmemiz gereken ilk şey nasıl çıkılacağıdır. Çıkmak için aşağıdaki komutu giriyoruz (iki nokta üst üste karakterinin komutun bir parçası olduğunu unutmayın):

```
:q
```

Kabuk istemine geri dönülmelidir. Herhangi bir nedenle `vi` çıkmazsa (genellikle henüz kaydedilmemiş bir dosyada değişiklik yaptığımız için), komuta ünlem işareti ekleyerek `vi`'ye gerçekten çıkmak istediğimizi belirtebiliriz.

```
:q!
```

İpucu: `vi'de` "kaybolursanız", `Esc tuşuna` iki kez basarak yolunuzu tekrar bulmaya çalışın.

Uyumluluk Modu

Yukarıdaki örnek başlangıç ekranında, "Vi uyumlu modda çalışıyor" metnini görüyoruz. Bu, `vim'in vim'in` gelişmiş davranışından ziyade `vi'nin` normal davranışına daha yakın bir modda çalışacağı anlamına gelir. Bu bölümün amaçları doğrultusunda, `vim'i` gelişmiş davranışıyla çalıştırmak isteyeceğiz. Bunu yapmak için birkaç seçeneğiniz var. `Vi` yerine `vim`'i çalıştırmayı deneyin. Bu işe yararsa, `.bashrc` dosyanıza `vi='vim' aliasını` eklemeyi düşünün. Alternatif olarak, `vim` yapılandırma dosyanıza bir satır eklemek için şu komutu kullanın:

```
echo "set nocp" >> ~/.vimrc
```

Farklı Linux dağıtımları `vim'i` farklı şekillerde paketler. Bazı dağıtımlar, varsayılan olarak

sınırlı sayıda `vim` özelliğini destekler. Aşağıdaki dersleri yaparken, eksik özelliklerle karşılaşabilirsiniz. Bu durumda, `vim'in` tam sürümünü yükleyin.

Düzenleme Modları

`Vi'yi` tekrar başlatalım, bu sefer ona var olmayan bir dosyanın adını verelim. `Vi` ile yeni bir dosya şu şekilde oluşturabiliriz:

```
[me@linuxbox ~]$ rm -f foo.txt
[me@linuxbox ~]$ vi foo.txt
```

Her şey yolunda giderse, şöyle bir ekran görmelisiniz:

```
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
"foo.txt" [Yeni Dosya]
```

Baştaki tilde karakterleri (~) o satırda metin olmadığını gösterir. Bu, dosyanın boş olduğunu gösterir. **Henüz hiçbir şey yazmayın!**

`Vi` hakkında öğrenilmesi gereken ikinci en önemli şey (çıkış yapmayı öğrendikten sonra), `vi`'nin *modal* bir *düzenleyici* olduğudur. `Vi` başlatıldığında, *normal modda* başlar. Bu modda, neredeyse her tuş bir komuttur, bu nedenle yazmaya başlarsak, `vi` çıldırır ve büyük bir karışıklık yaratır.

Ekleme Moduna Girme

Dosyamıza metin eklemek için önce *ekleme moduna* girmeliyiz. Bunu yapmak için `i harfini` yazıyoruz. Ardından, `vim` normal gelişmiş modunda çalışıyorsa ekranın altında aşağıdakileri görmeliyiz (`vi` uyumlu modda bu görünmez):

```
-- INSERT --
```

Şimdi metin girebiliriz. Şunu deneyin:

```
Hızlı kahverengi tilki tembel köpeğin üzerinden atlar.
```

Ekleme modundan çıkıp normal moda dönmek için `Esc tuşuna` basın.

Çalışmamızı Kaydetme

Dosyada yaptığımız değişikliği kaydetmek için *komut moduna* girmeliyiz. Bu, normal modda : tuşuna basarak yapılır. Bunu yaptıktan sonra, ekranın altında iki nokta üst üste işareti görünmelidir.

```
:
```

Değiştirdiğimiz dosyayı yazmak için, iki nokta üst üste işaretinden sonra `w` harfini yazıp `ENTER tuşuna` basarız.

```
:w
```

Dosya sabit diske yazılır ve ekranın altında aşağıdaki gibi bir onay mesajı alırız:

```
"foo.txt" [Yeni] 1L, 45C yazıldı
```

Not: `vim` üç ana düzenleme modunu *normal*, *insert* ve *com-mand* olarak adlandırırken, gerçek `vi` (ve belgeleri) bu modları sırasıyla *command*, *insert* ve *ex* olarak adlandırır. Çoğu çevrimiçi `vi` kaynağı bu şekilde atıfta bulunur ve evet, bu kafa karıştırıcı olabilir.

İmleci Hareket Ettirme

Normal modda `vi`, bazıları `less` ile ortak olan çok sayıda hareket komutu sunar. Tablo 12-1 bunların bir alt kümesini listeler.

Tablo 12-1: İmleç Hareket Tuşları

Tuş	İmleci hareket ettirir
`l` veya sağ ok	Bir karakter sağa.
`h` veya sol ok	Bir karakter kaldı.
`j` veya aşağı ok	Bir satır aşağı.
`k` veya yukarı ok	Bir satır yukarı.
`0` (sıfır)	Geçerli satırın başına.
`^`	Geçerli satırdaki ilk boşluk olmayan karaktere.
`$`	Geçerli satırın sonuna.
`w`	Bir sonraki kelimenin veya noktalama işaretinin başına.
`W`	Noktalama işaretlerini yok sayarak bir sonraki kelimenin başına.
`b`	Önceki kelimenin veya noktalama işaretinin başına.
`B`	Noktalama işaretlerini yok sayarak önceki kelimenin başına.
`Ctrl-f` veya `Sayfa Aşağı`	Bir sayfa aşağı.
`Ctrl-b` veya `Page Up`	Bir sayfa yukarı.
`numberG`	Satır *numarasına* gider. Örneğin, `1G` dosyasının ilk satırına gider.

Tuş	İmleci hareket ettirir
G	Dosyanın son satırına.

İmleç hareketleri için neden h, j, k ve l harfleri kullanılır? Vi ilk yazıldığında, tüm video terminallerinde ok tuşları bulunmuyordu ve yetenekli daktilocular, parmaklarını klavyeden kaldırmadan normal klavye tuşlarını kullanarak imleci hareket ettirebiliyorlardı.

Vi'deki birçok komutun önüne, daha önce listelenen G komutunda olduğu gibi bir sayı eklenebilir. Bir komutun önüne bir sayı ekleyerek, komutun kaç kez gerçekleştirileceğini belirtebiliriz. Örneğin, 5j komutu vi'nin imleci beş satır aşağı hareket ettirir.

Temel Düzenleme

Düzenleme işlemlerinin çoğu, metin ekleme, metin silme ve metni kesip yapıştırarak taşıma gibi birkaç temel işlemden oluşur. vi, elbette, tüm bu işlemleri kendine özgü bir şekilde destekler. vi ayrıca sınırlı bir geri alma işlevi de sunar. Normal moddayken u tuşuna basarsak, vi son yaptığınız değişikliği geri alır. Bu, temel düzenleme komutlarını denerken çok kullanışlı olacaktır.

Metin Ekleme

vi, ekleme moduna girmenin birkaç farklı yoluna sahiptir. Metin eklemek için i komutunu zaten kullandık.

Bir an için foo.txt dosyamıza geri dönelim.

```
Hızlı kahverengi tilki tembel köpeğin üzerinden atlar.
```

Bu cümlenin sonuna biraz metin eklemek istersek, i komutunun bunu yapamayacağını fark ederiz, çünkü imleci satırın sonunun ötesine taşıyamayız. vi, metin eklemek için mantıklı bir şekilde a komutu adında bir komut sağlar. İmleci satırın sonuna taşıyıp a yazarsak, imleç satırın sonunu geçecek ve vi ekleme moduna girecektir. Bu, biraz daha metin eklememizi sağlayacaktır.

```
Hızlı kahverengi tilki tembel köpeğin üzerinden atlar. Çok havalıydı.
```

Ekleme modundan çıkmak için Esc tuşuna basmayı unutmayın.

Neredeyse her zaman bir satırın sonuna metin eklemek isteyeceğimizden, `vi` mevcut satırın sonuna geçip ekleme yapmaya başlamak için bir kısayol sunar. Bu, `A` komutudur. Bunu deneyip dosyamızda birkaç satır daha ekleyelim.

İlk olarak, "0" (sıfır) komutunu kullanarak imleci satırın başına taşıyacağız. Şimdi `A` yazıp aşağıdaki metin satırlarını ekliyoruz:

```
Hızlı kahverengi tilki tembel köpeğin üzerinden atlar. Çok havalıydı.
Satır 2
Satır 3
Satır 4
Satır 5
```

Yine, `Esc tuşuna` basarak ekleme modundan çıkın.

Gördüğümüz gibi, `A` komutu ekleme moduna başlamadan önce imleci satırın sonuna taşıdığı için daha kullanışlıdır.

Satır Açma

Metin eklemenin bir başka yolu da bir satırı "açmaktır". Bu, mevcut iki satır arasına boş bir satır ekler ve ekleme moduna girer. Tablo 12-2'de açıklandığı gibi bunun iki çeşidi vardır.

Tablo 12-2: Satır Açma Tuşları

Komut	Açar
o	Mevcut satırın altındaki satır
O	Geçerli satırın üstündeki satır

Bunu şu şekilde gösterebiliriz: İmleci 3. satıra getirin ve `o` yazın.

```
Hızlı kahverengi tilki tembel köpeğin üzerinden atlar. Çok
havalıydı. 2. satır
3. satır

4. satır
5. satır
```

Üçüncü satırın altına yeni bir satır açıldı ve ekleme moduna girdik. Ekleme modundan çıkmak için

. Değişikliğimizi geri almak için `u tuşuna` basın.

İmlecin üzerindeki satırı açmak için `O tuşuna` basın.

```
Hızlı kahverengi tilki tembel köpeğin üzerinden atlar. Çok
havalıydı. Satır 2

Satır 3
Satır 4
Satır 5
```

`Esc tuşuna` basarak ekleme modundan çıkın ve `u tuşuna` basarak yaptığımız değişikliği geri alın.

Metin Silme

Beklenebileceği gibi, `vi` metin silmek için çeşitli yöntemler sunar ve bunların tümü iki tuş vuruşundan birini içerir. İlk olarak, `x` komutu imlecin bulunduğu konumdaki karakteri siler. `x`'in önüne, silinecek karakter sayısını belirten bir sayı yazılabilir. `d` komutu daha genel amaçlıdır. `x` gibi, bu komutun önüne de silme işleminin kaç kez gerçekleştirileceğini belirten bir sayı yazılabilir. Ayrıca, `d komutunun` ardından her zaman silme işleminin boyutunu kontrol eden bir hareket komutu gelir. Tablo 12-3'te bazı örnekler verilmiştir.

Tablo 12-3: Metin Silme Komutları

Komut	Siler
`x`	Geçerli karakter
`3x`	Geçerli karakter ve sonraki iki karakter
`dd`	Geçerli satır
`5dd`	Mevcut satır ve sonraki dört satır
`dW`	Geçerli imleç konumundan sonraki kelimenin başına kadar
`d$`	Geçerli imleç konumundan geçerli satırın sonuna kadar
`d0`	Mevcut imleç konumundan satırın başına kadar
`d^`	Geçerli imleç konumundan satırdaki ilk boşluk olmayan karaktere kadar

Komut	Siler
dG	Geçerli satırdan dosyanın sonuna kadar
d20G	Dosyanın mevcut satırından yirminci satırına kadar

İmleci metnimizin ilk satırındaki `It` kelimesinin üzerine getirin. Cümlenin geri kalan kısmı silinene kadar `x tuşuna` tekrar tekrar basın. Ardından, silme işlemi geri alınana kadar `u tuşuna` tekrar tekrar basın.

Not: Gerçek `vi` yalnızca tek seviyeli geri alma işlemini destekler. `vim ise` çoklu seviyeleri destekler.

Silme işlemini bu sefer `d` komutunu kullanarak tekrar deneyelim. Yine, imleci `It` kelimesinin üzerine getirin ve `dW` yazarak kelimeyi silin.

```
Hızlı kahverengi tilki tembel köpeğin üzerinden atlar.
güzeldi. Satır 2
Satır 3
Satır 4
Satır 5
```

İmleç konumundan satırın sonuna kadar silmek için `d$` yazın.

```
Hızlı kahverengi tilki tembel köpeğin üzerinden
atlar. Satır 2
Satır 3
Satır 4
Satır 5
```

`dG` yazarak mevcut satırdan dosyanın sonuna kadar silebilirsiniz.

```
~
~
~
~
~
```

Silme işlemini geri almak için `u tuşuna` üç kez basın.

Metin Kesme, Kopyalama ve Yapıştırma

`d` komutu metni silmekle kalmaz, aynı zamanda metni "keser" de. `d` komutunu her kullandığımızda, silinen metin yapıştırma tamponuna (panoya benzer) kopyalanır ve daha sonra p komutuyla tamponun içeriğini imlecin arkasına yapıştırmak veya P komutuyla imlecin önüne yapıştırmak için geri çağırabiliriz.

`y` komutu, `d` komutunun metni kesmek için kullanıldığı gibi metni "çekmek" (kopyalamak) için kullanılır. Tablo 12-4, `y` komutunu çeşitli hareket komutlarıyla birleştirmenin bazı örneklerini vermektedir.

Tablo 12- 4: Çekme Komutları

Komut	Kopyalar
`yy`	Geçerli satır
`5yy`	Geçerli satır ve sonraki dört satır
`yW`	Geçerli imleç konumundan sonraki kelimenin başına kadar
`y$`	Geçerli imleç konumundan geçerli satırın sonuna kadar
`y0`	Geçerli imleç konumundan satırın başına
`y^`	Mevcut imleç konumundan satırdaki ilk boşluk olmayan karaktere kadar
`yG`	Geçerli satırdan dosyanın sonuna kadar
`y20G`	Geçerli satırdan dosyanın yirminci satırına kadar

Biraz kopyala-yapıştır yapalım. İmleci metnin ilk satırına getirin ve `yy` yazarak mevcut satırı kopyalayın. Ardından, imleci son satıra (`G`) getirin ve `p` yazarak satırı mevcut satırın altına yapıştırın.

```
Hızlı kahverengi tilki tembel köpeğin üzerinden atlar. Çok
havalıydı. 2. satır
3. satır
4. satır
5. satır
Çevik kahverengi tilki tembel köpeğin üzerinden atlar. Çok havalıydı.
```

Daha önce olduğu gibi, u komutu yaptığımız değişikliği geri alır. İmleç hala dosyanın son satırında iken, P tuşuna basarak metni mevcut satırın üzerine yapıştırın.

```
Hızlı kahverengi tilki tembel köpeğin üzerinden atlar. Çok
havalıydı. Satır 2
3. satır
4. satır
Hızlı kahverengi tilki tembel köpeğin üzerinden atlar. Çok havalıydı.
5. satır
```

Tablo 12-4'teki diğer y komutlarını deneyin ve p ve P komutlarının davranışlarını öğrenin. İşiniz bittiğinde, dosyayı orijinal durumuna geri döndürün.

Satırları Birleştirme

vi, satır kavramı konusunda oldukça katıdır. Normalde, imleci bir satırın sonuna getirip satır sonu karakterini silerek bir satırı altındaki satırla birleştirmek mümkün değildir. Bu nedenle vi, satırları birleştirmek için özel bir komut olan J (imleci hareket ettirmek için kullanılan j ile karıştırılmamalıdır) sağlar.

İmleci 3. satıra getirip J komutunu yazarsak, şunlar olur:

```
Hızlı kahverengi tilki tembel köpeğin üzerinden atlar. Çok
havalıydı. 2. satır
3. satır 4. satır
5. satır
```

Arama ve Değiştir

vi, aramalara göre imleci belirli konumlara taşıma özelliğine sahiptir. Bunu tek bir satırda veya tüm dosya üzerinde yapabilir. Ayrıca, kullanıcının onayı ile veya onayı olmadan metin değiştirme işlemi gerçekleştirebilir.

Satır İçinde Arama

f komutu bir satırı arar ve imleci belirtilen karakterin bir sonraki örneğine taşır. Örneğin, fa komutu imleci geçerli satırdaki a karakterinin bir sonraki örneğine taşır. Bir satır içinde karakter araması yaptıktan sonra, noktalı virgül yazarak aramayı tekrarlayabilirsiniz.

Tüm Dosyayı Arama

İmleci bir kelime veya kelime öbeğinin bir sonraki geçtiği yere taşımak için / komutu kullanılır. Bu, daha önce less programında öğrendiğimizle aynı şekilde çalışır.
/ komutunu yazdığınızda, ekranın altında bir / işareti belirir. Ardından, aranacak kelimeyi veya kelime öbeğini yazıp ENTER tuşuna basın. İmleç, arama dizesini içeren bir sonraki konuma gider. n komutu ile önceki arama dizesini kullanarak arama tekrarlanabilir. İşte bir örnek:

```
Hızlı kahverengi tilki tembel köpeğin üzerinden atlar. Serindi.
Satır 2
Satır 3
Satır 4
Satır 5
```

İmleci dosyanın ilk satırına getirin. Şunu yazın ve ENTER tuşuna basın:

```
/Satır
```

İmleç 2. satıra hareket edecektir. Ardından, n yazın ve imleç 3. satıra hareket edecektir. n komutunu tekrarlamak, eşleşmeler bitene kadar imleci dosyanın aşağısına hareket ettirecektir. Şimdiye kadar arama desenlerimiz için sadece kelimeler ve kelime öbekleri kullandık, ancak vi, karmaşık metin desenlerini ifade etmenin güçlü bir yöntemi olan *düzenli ifadeleri* kullanmaya izin verir. Düzenli ifadeleri 19. bölümde ayrıntılı olarak ele alacağız.

Global Arama ve Değiştirme

vi, bir dizi satır veya tüm dosya üzerinde arama ve değiştirme işlemlerini (vi'da *ikame olarak* adlandırılır) gerçekleştirmek için komut modunu kullanır. Tüm dosya için Line kelimesini line olarak değiştirmek için aşağıdaki komutu gireriz:

```
:%s/Line/line/g
```

Bu komutu ayrı öğelere ayırıp her birinin ne işe yaradığını görelim (bkz. Tablo 12-5).

Tablo 12- 5: Global Arama ve Değiştirme Sözdizimi Örneği

Öğe	Anlam

`:`	İki nokta üst üste karakteri komut moduna girer.
`%`	Bu, işlem için satır aralığını belirtir. `%` ilk satırdan son satıra kadar anlamına gelen bir kısayoldur. Alternatif olarak, aralık `1,5` (dosyamız beş satır uzunluğunda olduğu için) veya `1,$` olarak da belirtilebilirdi, bu da "1. satırdan dosyanın son satırına kadar" anlamına gelir. Satır aralığı belirtilmezse, işlem yalnızca geçerli satırda gerçekleştirilir.
`s`	Bu, işlemi belirtir. Bu durumda, bu işlem ikame (ara ve değiştir) işlemidir.
`/Satır/satır/`	Bu, arama desenini ve değiştirme metnini belirtir.
`g`	Bu, arama ve değiştirme işleminin satırdaki arama dizesinin her örneğinde gerçekleştirildiği anlamında "global" anlamına gelir. Atlanırsa, her satırdaki arama dizesinin yalnızca ilk örneği değiştirilir.

Arama ve değiştirme komutunu uyguladıktan sonra dosyamız şöyle görünüyor:

```
Hızlı kahverengi tilki tembel köpeğin üzerinden atlar. Çok
havalıydı. satır 2
satır 3
satır 4
satır 5
```

Kullanıcı onayı ile bir değiştirme komutu da belirtebiliriz. Bu, komutun sonuna bir `c` ekleyerek yapılır. İşte bir örnek:

```
:%s/line/Line/gc
```

Bu komut, dosyamızı önceki haline geri döndürür; ancak, her değiştirme işleminden önce `vi` durur ve şu mesajla değiştirmeyi onaylamamızı ister:

```
Line ile değiştir (y/n/a/q/l/^E/^Y)?
```

Parantez içindeki karakterlerin her biri, Tablo 12-6'da açıklandığı gibi olası bir seçimdir.

Tablo 12-6: Değiştirme Onay Tuşları

Anahtar	Eylem
`y`	Değiştirme işlemini gerçekleştirir.
`n`	Desenin bu örneğini atlayın.
`a`	Bu ve sonraki tüm desen örneklerinde değiştirme işlemini gerçekleştirin.
`q` veya `Esc`	Değiştirmeyi sonlandır.
`l`	Bu değiştirmeyi gerçekleştirin ve ardından çıkın. Bu, "last" (son) kelimesinin kısaltmasıdır.
`Ctrl-e, Ctrl-y`	Sırasıyla aşağı ve yukarı kaydırın. Bu, önerilen değiştirmenin bağlamını görüntülemek için kullanışlıdır.

`y` yazarsanız, değiştirme işlemi gerçekleştirilir, `n` yazarsanız `vi` bu örneği atlayıp bir sonrakine geçer.

Birden Fazla Dosyayı Düzenleme

Aynı anda birden fazla dosyayı düzenlemek genellikle yararlıdır. Birden fazla dosyada değişiklik yapmanız veya bir dosyadaki içeriği başka bir dosyaya kopyalamanız gerekebilir. `Vi` ile komut satırında belirtmek suretiyle düzenlemek için birden fazla dosya açabiliriz.

```
vi dosya1 dosya2 dosya3...
```

Mevcut vi oturumumuzdan çıkıp düzenlemek için yeni bir dosya oluşturalım. Değiştirdiğimiz metni kaydedip `vi'dan` çıkmak için `:wq` yazın. Ardından, ana dizinde oynayabileceğimiz ek bir dosya oluşturacağız. Bu dosyayı `ls` komutunun çıktısını yakalayarak oluşturacağız.

```
[me@linuxbox ~]$ ls -l /usr/bin > ls-output.txt
```

Eski dosyamızı ve yeni dosyamızı `vi` ile düzenleyelim.

```
[me@linuxbox ~]$ vi foo.txt ls-output.txt
```

`vi` başlatılacak ve ekranda ilk dosya görünecektir.

```
Hızlı kahverengi tilki tembel köpeğin üzerinden atlar. Çok
havalıydı. Satır 2
Satır 3
Satır 4
Satır 5
```

Dosyalar Arasında Geçiş Yapma

Bir dosyadan diğerine geçmek için şu ex komutunu kullanın:

```
:bn
```

Önceki dosyaya geri dönmek için aşağıdakileri kullanın:

```
:bp
```

Bir dosyadan diğerine geçebilsek de, `vi`, geçerli dosyada kaydedilmemiş değişiklikler varsa dosya değiştirmeyi engelleyen bir politika uygular. `Vi'yi` dosya değiştirmeye ve değişikliklerinizi terk etmeye zorlamak için komuta ünlem işareti (!) ekleyin.

Yukarıda açıklanan geçiş yöntemine ek olarak, `vim` (ve `vi'nin` bazı sürümleri) birden fazla dosyayı daha kolay yönetmeyi sağlayan bazı komut modu komutları sunar. `:`buffers komutuyla düzenlenmekte olan dosyaların listesini görüntüleyebiliriz. Bunu yaptığımızda, ekranın altında dosyaların listesi görüntülenir.

```
:buffers
  1 %a    "foo.txt"                          satır 1
  2       "ls-output.txt"                    satır 0
Devam etmek için ENTER tuşuna basın veya
komutu yazın
```

Başka bir tampona (dosyaya) geçmek için, `:buffer` (daha hızlı olması için `:b` olarak kısaltılabilir) yazıp ardından düzenlemek istediğimiz tamponun numarasını yazın. Örneğin, `foo.txt` dosyasını içeren tampon 1'den `ls-output.txt` dosyasını içeren tampon 2'ye geçmek için şunu yazın:

```
:buffer 2
```

Ekranımızda şimdi ikinci dosya görüntüleniyor. Tamponları değiştirebileceğimiz başka bir yol da, daha önce bahsedilen
`:bn` (kısaltması "buffer next") ve `:bp` (kısaltması "buffer previous") komutlarını kullanmaktır.

Düzenlemek için Ek Dosyalar Açma

Mevcut düzenleme oturumumuza dosya eklemek de mümkündür. Komut modu komutu `:e` (kısaltması "edit") ve ardından bir dosya adı girildiğinde ek bir dosya açılır. Mevcut düzenleme oturumumuzu sonlandırıp komut satırına geri dönelim.

`Vi'yi` tek bir dosyayla yeniden başlatın.

```
[me@linuxbox ~]$ vi foo.txt
```

İkinci dosyamızı eklemek için aşağıdakileri girin:

```
:e ls-output.txt
```

Ekrana görünmelidir. İlk dosya hala mevcut, bunu şu şekilde doğrulayabiliriz:

```
:buffers
  1 #     "foo.txt"                          satır 1
  2 %a    "ls-output.txt"                    satır 0
Devam etmek için ENTER tuşuna basın veya
komutu yazın
```

Bir Dosyadan Diğerine İçerik Kopyalama

Çoğu zaman, birden fazla dosyayı düzenlerken, bir dosyanın bir bölümünü düzenlediğimiz başka bir dosyaya kopyalamak isteriz. Bu, daha önce kullandığımız normal kopyala-yapıştır komutlarıyla kolayca yapılabilir. Bunu şu şekilde gösterebiliriz. İlk olarak, iki dosyamızı kullanarak, şunu girerek tampon 1'e (`foo.txt`) geçin:

```
:tampon 1
```

Bu bize şunu vermelidir:

```
Hızlı kahverengi tilki tembel köpeğin üzerinden atlar. Çok
havalıydı. Satır 2
Satır 3
Satır 4
Satır 5
```

Ardından, imleci ilk satıra getirin ve `yy` yazarak satırı kopyalayın. Aşağıdakileri girerek ikinci arabelleğe geçin:

```
:buffer 2
```

Ekran şimdi aşağıdaki gibi bazı dosya listelerini içerecektir (burada sadece bir kısmı gösterilmektedir):

```
toplam
 -rwxr-xr-x  1 root root        31316  2024-12-05  08:58 [
 -rwxr-xr-x  1 root root         8240  2024-12-09  13:39 411toppm
 -rwxr-xr-x  1 root root       111276  2025-01-31  13:36 a2p
 -rwxr-xr-x  1 root root        25368  2023-10-06  20:16 a52dec
 -rwxr-xr-x  1 root root        11532  2024-05-04  17:43 aafire
 -rwxr-xr-x  1 root root         7292  2024-05-04  17:43 aainfo
```

İmleci ilk satıra getirin ve `p` komutunu yazarak önceki dosyadan kopyaladığımız satırı yapıştırın.

```
toplam 343700
 -rwxr-xr-x  1 root root        31316  2024-12-05  08:58 [        rdı.
 -rwxr-xr-x  1 root root         8240  2024-12-09  13:39 411toppm
 -rwxr-xr-x  1 root root       111276  2025-01-31  13:36 a2p
 -rwxr-xr-x  1 root root        25368  2023-10-06  20:16 a52dec
 -rwxr-xr-x  1 root root        11532  2024-05-04  17:43 aafire
 -rwxr-xr-x  1 root root         7292  2024-05-04  17:43 aainfo
```

Bir Dosyanın Tamamını Başka Bir Dosyaya Ekleme

Düzenlediğimiz dosyaya bir dosyayı tamamen eklemek de mümkündür. Bunun nasıl yapıldığını görmek için `vi` oturumumuzu sonlandırıp tek bir dosya ile yeni bir oturum açalım.

```
[me@linuxbox ~]$ vi ls-output.txt
```

Dosya listemizi tekrar göreceğiz.

```
toplam
 -rwxr-xr-x  1 root root        31316  2024-12-05  08:58 [
 -rwxr-xr-x  1 root root         8240  2024-12-09  13:39 411toppm
 -rwxr-xr-x  1 root root       111276  2025-01-31  13:36 a2p
 -rwxr-xr-x  1 root root        25368  2023-10-06  20:16 a52dec
 -rwxr-xr-x  1 root root        11532  2024-05-04  17:43 aafire
 -rwxr-xr-x  1 root root         7292  2024-05-04  17:43 aainfo
```

İmleci üçüncü satıra getirin ve ardından aşağıdaki komut modu komutunu girin:

```
:r foo.txt
```

`:r` komutu (kısaltması "read"dir) belirtilen dosyayı imleç konumunun altına ekler. Ekranımız şimdi şöyle görünmelidir:

```
toplam 343700
-rwxr-xr-x 1 root root        31316 2024-12-05 08:58 [
-rwxr-xr-x 1 root root         8240 2024-12-09 13:39 411toppm
Hızlı kahverengi tilki tembel köpeğin üzerinden atlar. Çok
havalıydı. Satır 2
Satır 3
Satır 4
Satır 5
-rwxr-xr-x 1 root root       111276 2025-01-31 13:36 a2p
-rwxr-xr-x 1 root root        25368 2023-10-06 20:16 a52dec
-rwxr-xr-x 1 kök kök          11532 2024-05-04 17:43 aafire
-rwxr-xr-x 1 root root         7292 2024-05-04 17:43 aainfo
```

Çalışmamızı Kaydetme

`Vi'deki` her şey gibi, düzenlediğimiz dosyaları kaydetmenin de birkaç farklı yolu vardır. `:w` komutunu daha önce ele aldık, ancak yararlı olabilecek başka komutlar da vardır.

Normal modda, `ZZ` yazdığınızda mevcut dosya kaydedilir ve `vi'den` çıkılır. Benzer şekilde,

`:wq` komutu, `:w` ve `:q` komutlarını birleştirerek dosyayı kaydeder ve çıkmanızı sağlar.

`:w` komutu isteğe bağlı bir dosya adı da belirtebilir. Bu, "Farklı Kaydet..." gibi çalışır. Örneğin, `foo.txt dosyasını` düzenliyor ve `foo1.txt` adlı alternatif bir sürümü kaydetmek istiyorsak, şunu gireriz:

```
:w foo1.txt
```

> **Not:** Bu komut dosyayı yeni bir adla kaydeder, ancak düzenlediğimiz dosyanın adını değiştirmez. Düzenlemeye devam ederken, `foo1.txt dosyasını` değil, `foo.txt` dosyasını düzenlemeye devam ederiz.

Bash da vi yapar.

8. bölümde, komut satırının içeriğini düzenlemenin çeşitli yollarını inceledik. `Bash'ın` kullandığı belirli düzenleme komutları rastgele değildir. Bunlar, emacs metin düzenleyicisinden esinlenerek oluşturulmuştur. Bu, `bash'ın` varsayılan ayarıdır, ancak `bash` vi tarzı komut satırı düzenlemeyi de destekler. Bu özellik, aşağıdaki komutla kolayca etkinleştirilebilir:

```
[me@linuxbox ~]$ set -o vi
```

Bu yapıldıktan sonra, öğrendiğimiz vi tarzı düzenleme komutlarının çoğunu kullanabiliriz. Hadi deneyelim. Komut istemine aşağıdaki örnek metni yazın:

```
[me@linuxbox ~]$ hızlı kahverengi tilki tembel köpeğin üzerinden atlar
```

Tıpkı eskisi gibi ok tuşlarıyla imleci hareket ettirebilir ve normal şekilde karakter yazabiliriz. Bu şekilde davranmasının nedeni, yeni bir komut satırı başlattığımızda düzenleyicinin ekleme modunda olması ve `vim` gibi davranmasıdır. Daha ilginç özelliklere ulaşmak için normal moda geçmemiz gerekir. `ESC tuşuna` basarak ekleme modundan çıkarız. Tüm hareket komutları, yank, delete ve paste, `vim'de` tek satırlık bir metin dosyasını düzenliyormuşuz gibi çalışır. Ekleme moduna dönmek için, `i` veya `A` gibi uygun normal mod komutunu kullanırız.

`Bash'ı` vi tarzı komut satırı düzenlemeyi kullanacak şekilde ayarlamak, vi klavye becerilerimizi güçlendirmek için iyi bir yoldur ve hatırlamamız gereken düzenleme komutlarının sayısını azaltma gibi ek bir avantajı da vardır. Bir deneyin. Bunu kalıcı hale getirmek için, .bashrc dosyamızın sonuna `set -o vi`

komutunu ekleyebiliriz.

Emacs tarzı düzenleme moduna dönmek için şu komutu girin:

```
[me@linuxbox ~]$ set -o emacs
```

Not: Bu özellik için birçok çevrimiçi öğretici mevcuttur, ancak çoğunun vim'in (normal, insert ve command) yerine geleneksel vi modu adlarını (command, insert ve ex) kullandığını unutmayın.

Özet

Bu temel becerilerle, artık tipik bir Linux sistemini korumak için gereken metin düzenleme işlemlerinin çoğunu gerçekleştirebiliriz. `Vim'i` düzenli olarak kullanmayı öğrenmek, uzun vadede karşılığını verecektir. Vi tarzı editörler Unix kültürüne o kadar derinlemesine yerleşmiştir ki, bu tasarımdan etkilenen birçok başka program da göreceğiz. `Less,` bu etkinin iyi bir örneğidir.

Daha Fazla Okuma

Bu bölümde ele aldığımız tüm konulara rağmen, `vim'in` yapabileceklerinin sadece çok küçük bir kısmını gördük. `Vi'yi` ustaca kullanma yolunda ilerlemeniz için kullanabileceğiniz birkaç çevrimiçi kaynak aşağıda verilmiştir:

- *Vim, with Vigor* – LinuxCommand.org'da yer alan ve okuyucuyu orta düzeyde beceriye ulaştıran bu öğreticinin devamı niteliğindedir. Şu adreste bulabilirsiniz: https://linuxcommand.org/lc3_adv_vimvigor.php
- *Vi Editörünü Öğrenmek* – Wikipedia'dan `vi` ve `vim` dahil olmak üzere benzer birkaç programa ilişkin kısa bir kılavuz sunan bir Wikibook. Şu adreste bulabilirsiniz: https://en.wikibooks.org/wiki/Learning_the_vi_Editor
- `Vi'nin` yaratıcısı Bill Joy hakkında bir Wikipedia makalesi: https://en.wikipedia.org/wiki/Bill_Joy
- `Vim'in` yazarı Bram Moolenaar hakkında Wikipedia makalesi: https://en.wikipedia.org/wiki/Bram_Moolenaar

13 – Komut İstemi Özelleştirme

Bu bölümde, görünüşte önemsiz bir ayrıntı olan kabuk istemimizi inceleyeceğiz. Bu inceleme, kabuk ve terminal emülatör programının bazı iç işleyişlerini ortaya çıkaracaktır.

Linux'taki pek çok şey gibi, kabuk istemcisi de son derece yapılandırılabilir ve biz bunu neredeyse doğal kabul etsek de, istemciyi kontrol etmeyi öğrendiğimizde gerçekten çok yararlı bir araç olduğunu görürüz.

İmlecin Anatomisi

Varsayılan istemimiz şuna benzer:

```
[me@linuxbox ~]$
```

Burada kullanıcı adımız, ana bilgisayar adımız ve mevcut çalışma dizini yer almaktadır, peki bu nasıl gerçekleşmiştir? Aslında çok basit. Komut istemi, `PS1` ("prompt string 1"in kısaltması) adlı bir ortam değişkeni ile tanımlanmıştır. `PS1'in` içeriğini `echo` komutu ile görüntüleyebiliriz.

```
[me@linuxbox ~]$ echo $PS1
[\u@\h \W]\$
```

> **Not:** Sonuçlarınız yukarıdaki örnekle aynı değilse endişelenmeyin. Her Linux dağıtımı komut istem dizesini biraz farklı tanımlar, bazıları oldukça egzotik bir şekilde.

Sonuçlardan, `PS1'in` komut istemimizde gördüğümüz köşeli parantez, at işareti ve dolar işareti gibi birkaç karakteri içerdiğini görebiliriz, ancak geri kalanı bir gizemdir. Aramızdaki zeki olanlar, bunların 7. Bölümde gördüğümüz gibi *ters eğik çizgi ile kaçan özel karakterler* olduğunu fark edecektir. Tablo 13-1, `bash'ın`

komut istem dizisinde özel olarak ele aldığı karakterlerin kısmi bir listesini sunmaktadır.

Tablo 13-1: Kabuk Komut Satırlarında Kullanılan Kaçış Kodları

Sıra	Görüntülenen Değer
\a	ASCII zil. Bu, karşılaştığında bilgisayarın bip sesi çıkarmasını sağlar.
\d	Gün, ay, tarih biçiminde geçerli tarih. Örneğin, "26 Mayıs Pazartesi".
\e	ASCII kaçış karakteri (033).
\h	Yerel makinenin ana bilgisayar adı, sondaki etki alanı adı hariç.
\H	Tam ana bilgisayar adı.
\j	Geçerli kabuk oturumunda çalışan işlerin sayısı.
\l	Geçerli terminal aygıtının adı.
\n	Satır sonu karakteri.
\r	Satır başı.
\s	Kabuk programının adı.
\t	24 saatlik saat:dakika:saniye biçiminde geçerli saat.
\T	12 saatlik formatta geçerli saat.
\@	12 saatlik AM/PM formatında geçerli saat.
\A	24 saatlik saat:dakika biçiminde geçerli saat.
\u	Geçerli kullanıcının kullanıcı adı.
\v	Kabuğun sürüm numarası.
\V	Kabuğun sürüm ve yayın numaraları.
\w	Geçerli çalışma dizininin adı.
\W	Geçerli çalışma dizini adının son kısmı.
\!	Geçerli komutun geçmiş numarası.
\#	Bu kabuk oturumu sırasında girilen komutların sayısı.
\$	Süper kullanıcı ayrıcalıklarımız yoksa, burada "$" karakteri görüntülenir. Bu durumda, bunun yerine "#" karakteri görüntülenir.
\[	Bir veya daha fazla yazdırılamayan karakter dizisinin başlangıcını belirtir. Bu, terminal emülatörünü bir şekilde manipüle eden, örneğin

Sıra	Değer Görüntülenen
	imleç veya metin renklerini değiştirmek. Bu, imlecin doğru konumlandırılması için komut isteminin uzunluğunu doğru bir şekilde hesaplamak için gereklidir.
\]	Yazdırılmayan karakter dizisinin sonunu belirtir.

Bazı Alternatif Komut İstemi Tasarımlarını Denemek

Bu özel karakter listesi ile komut istemini değiştirerek etkisini görebiliriz. Öncelikle, daha sonra geri yükleyebilmek için mevcut komut istemi dizesini yedekleyeceğiz. Bunu yapmak için, mevcut dizeyi kendimiz oluşturduğumuz başka bir kabuk değişkenine kopyalayacağız.

```
[me@linuxbox ~]$ ps1_old="$PS1"
```

`ps1_old` adlı yeni bir değişken oluşturuyoruz ve ona `PS1` değerini atıyoruz. `Echo` komutunu kullanarak dizgenin kopyalandığını doğrulayabiliriz.

```
[me@linuxbox ~]$ echo $ps1_old
[\u@\h \W]\$
```

Terminal oturumumuz sırasında istediğimiz zaman bu işlemi tersine çevirerek orijinal komut istemini geri yükleyebiliriz.

```
[me@linuxbox ~]$ PS1="$ps1_old"
```

Artık devam etmeye hazırız, boş bir komut istemi dizesi olduğunda ne olacağını görelim.

```
[me@linuxbox ~]$ PS1=
```

Komut istemi dizesine hiçbir şey atamazsak, hiçbir şey elde edemeyiz. Hiçbir komut istemi dizesi olmaz! Komut istemi hala oradadır, ancak istediğimiz gibi hiçbir şey göstermez. Bu durum biraz rahatsız edici olduğu için, onu minimal bir komut istemiyle değiştireceğiz.

```
PS1="\$ "
```

Böylesi daha iyi. En azından şimdi ne yaptığımızı görebiliyoruz. Çift tırnak işaretlerinin içindeki boşluğa dikkat edin

Bu, komut istemi görüntülendiğinde dolar işareti ile imleç arasında boşluk sağlar.

Komut istemimize bir zil sesi ekleyelim.

```
$ PS1="\[\a\]\$ "
```

Şimdi, komut istemi her görüntülendiğinde bir bip sesi duymamız gerekir, ancak bazı sistemler bu "özelliği" devre dışı bırakır. Bu can sıkıcı olabilir, ancak özellikle uzun süren bir komutun yürütüldüğünde bildirim almamız gerektiğinde yararlı olabilir. `\[` ve `\]` dizilerini eklediğimizi unutmayın. ASCII zil (`\a`) "yazdırılmadığından", yani imleci hareket ettirmediğinden, bash'ın komut isteminin uzunluğunu doğru bir şekilde belirleyebilmesi için bunu ona bildirmemiz gerekir.

Şimdi, bazı ana bilgisayar adı ve saat bilgilerini içeren bilgilendirici bir komut istemi oluşturmaya çalışalım.

```
$ PS1="\A \h \$ "
17:33 linuxbox $
```

Komut istemimize günün saatini eklemek, belirli görevleri ne zaman gerçekleştirdiğimizi takip etmemiz gerektiğinde yararlı olacaktır. Son olarak, orijinal komut istemimize benzer yeni bir komut istemi oluşturacağız.

```
17:37 linuxbox $ PS1="<\u@\h \W>\$ "
<me@linuxbox ~>$
```

Yukarıdaki tabloda listelenen diğer dizileri deneyin ve harika bir yeni komut istemi bulup bulamadığınızı görün.

Renk Ekleme

Çoğu terminal emülatör programı, karakter özelliklerini (renk, kalın metin ve korkunç yanıp sönen metin gibi) ve imleç konumunu kontrol etmek için belirli yazdırılmayan karakter dizilerine yanıt verir. İmleç konumunu birazdan ele alacağız, ancak önce renge bakacağız.

Terminal Karışıklığı

Eski zamanlarda, terminaller uzak bilgisayarlara bağlıyken, birçok rakip terminal markası vardı ve hepsi farklı şekilde çalışıyordu. Farklı klavyeleri vardı ve hepsi kontrol bilgilerini farklı şekillerde yorumluyordu. Unix ve Unix benzeri sistemler, terminal kontrolünün karmaşıklığıyla başa çıkmak için oldukça karmaşık iki alt sisteme sahiptir (`termcap` ve `terminfo` olarak adlandırılır). Terminal emülatör ayarlarınızın en derin köşelerine bakarsanız, terminal emülasyonunun türüne ilişkin bir ayar bulabilirsiniz.

Terminallerin bir tür ortak dil konuşmasını sağlamak amacıyla, Amerikan Ulusal Standartlar Enstitüsü (ANSI) video terminallerini kontrol etmek için standart bir karakter dizisi seti geliştirdi. Eski DOS kullanıcıları, bu kodların yorumlanmasını sağlamak için kullanılan `ANSI.SYS` dosyasını hatırlayacaktır.

Karakter rengi, terminal emülatörüne görüntülenecek karakter akışına gömülü bir *ANSI kaçış kodu* gönderilerek kontrol edilir. Kontrol kodu ekranda "yazdırılmaz"; bunun yerine, terminal tarafından bir komut olarak yorumlanır. Tablo 13-1'de gördüğümüz gibi, `\[` ve `\]` dizileri yazdırılmayan karakterleri kapsüllemek için kullanılır. Bir ANSI kaçış kodu `\e` (sekizlik 033, `Esc` tarafından üretilen kod) ile başlar, ardından isteğe bağlı bir karakter özniteliği ve bir komut gelir. Örneğin, metin rengini normal (öznitelik = 0), siyah metin olarak ayarlamak için kullanılan kod aşağıdaki gibidir:

```
\e[0;30m
```

Tablo 13-2, kullanılabilir metin renklerini listeler. Renklerin, "açık" renkler görünümü yaratan kalın karakter özniteliğinin (1) uygulanmasıyla iki gruba ayrıldığına dikkat edin.

Tablo 13- 2: Metin Renklerini Ayarlamak İçin Kullanılan Kaçış Dizileri

Dizi	Metin rengi	Dizi	Metin rengi
`\e[0;30m`	Siyah	`\e[1;30m`	Koyu gri
`\e[0;31m`	Kırmızı	`\e[1;31m`	Açık kırmızı
`\e[0;32m`	Yeşil	`\e[1;32m`	Açık yeşil
`\e[0;33m`	Kahverengi	`\e[1;33m`	Sarı
`\e[0;34m`	Mavi	`\e[1;34m`	Açık mavi
`\e[0;35m`	Mor	`\e[1;35m`	Açık mor
`\e[0;36m`	Camgöbeği	`\e[1;36m`	Açık camgöbeği
`\e[0;37m`	Açık gri	`\e[1;37m`	Beyaz

Kırmızı bir komut istemi yapmaya çalışalım. Başına kaçış kodunu ekleyeceğiz.

```
<me@linuxbox ~>$ PS1="\[\e[0;31m\]<\u@\h \W>\$ "
<me@linuxbox ~>$
```

Bu işe yarıyor, ancak komut isteminden sonra yazdığımız tüm metinlerin de kırmızı renkte görüntüleneceğini unutmayın. Bunu düzeltmek için, komut isteminin sonuna terminal emülatörüne önceki renge dönmesini söyleyen başka bir kaçış kodu ekleyeceğiz.

```
<me@linuxbox ~>$ PS1="\[\e[0;31m\]<\u@\h \W>\$\[\e[0m\] "
<me@linuxbox ~>$
```

Daha iyi oldu!

Tablo 13-3'te listelenen kodları kullanarak metin arka plan rengini de ayarlamak mümkündür. Arka plan renkleri kalın yazı özelliğini desteklemez.

Tablo 13-3: Arka Plan Rengini Ayarlamak İçin Kullanılan Kaçış Dizileri

Dizi	Arka plan rengi	Dizi	Arka plan rengi
`\e[0;40m`	Siyah	`\e[0;44m`	Mavi
`\e[0;41m`	Kırmızı	`\e[0;45m`	Mor
`\e[0;42m`	Yeşil	`\e[0;46m`	Camgöbeği
`\e[0;43m`	Kahverengi	`\e[0;47m`	Açık gri

İlk kaçış koduna basit bir değişiklik uygulayarak kırmızı arka planlı bir komut istemi oluşturabiliriz.

```
<me@linuxbox ~>$ PS1="\[\e[0;41m\]<\u@\h \W>\$\[\e[0m\] "
<me@linuxbox ~>$
```

Renk kodlarını deneyin ve neler yaratabileceğinizi görün!

Not: Normal (0) ve kalın (1) karakter özelliklerinin yanı sıra, metne alt çizgi (4), yanıp sönen (5) ve ters (7) özellikleri de verilebilir. Ancak, estetik nedenlerle birçok terminal emülatörü yanıp sönen özelliği desteklemez.

İmleci Hareket Ettirme

Kaçış kodları, imleci konumlandırmak için kullanılabilir. Bu, genellikle ekranın farklı bir konumunda, örneğin komut istemi her çizildiğinde üst köşede saat veya başka tür bilgiler sağlamak için kullanılır. Tablo 13-4, imleci konumlandıran kaçış kodlarını listeler.

Tablo 13-4: İmleç Hareketleri Kaçış Dizileri

Kaçış Kodu	Eylem
`\e[`*l*`;`*c*`H`	İmleci *l* satırına ve *c* sütununa taşıma
`\e[`*n*`A`	İmleci *n* satır yukarı taşıma
`\e[`*n*`B`	İmleci *n* satır aşağı taşıma
`\e[`*n*`C`	İmleci *n* karakter ileri hareket ettir
`\e[`*n*`D`	İmleci *n* karakter geriye hareket ettir
`\e[2J`	Ekranı temizleyin ve imleci sol üst köşeye (satır 0, sütun 0) taşıyın
`\e[K`	İmleç konumundan geçerli satırın sonuna kadar silin
`\e[s`	Geçerli imleç konumunu saklayın
`\e[u`	Kaydedilen imleç konumunu geri çağır

Tablo 13-4'teki kodları kullanarak, komut istemi her görüntülendiğinde ekranın üst kısmında saat (sarı metinle görüntülenir) içeren kırmızı bir çubuk çizen bir komut istemi oluşturacağız. Komut istemi için kod, şu zorlu görünümlü dizedir:

```
PS1="\[\e[s\e[0;0H\e[0;41m\e[K\e[1;33m\t\e[0m\e[u\]
<\u@\h \W>\$ "
```

Tablo 13-5, dizgenin her bir parçasının ne işe yaradığını özetlemektedir.

Tablo 13-5: Karmaşık Komut İstemi Dizisinin Ayrıntıları

Sıra	Eylem
`\[`	Yazdırılmayan karakter dizisini başlatır. Bunun amacı, `bash'ın` görünür komut isteminin boyutunu doğru bir şekilde hesaplamasını sağlamaktır. Doğru bir hesaplama yapılmazsa, komut satırı düzenleme özellikleri imleci doğru şekilde konumlandıramaz.
`\e[s`	İmleç konumunu saklayın. Bu, ekranın üst kısmına çubuk ve saat çizildikten sonra komut istemine geri dönmek için gereklidir. *Bazı terminal emülatörlerinin bu kodu tanımadığını unutmayın.*
`\e[0;0H`	İmleci sol üst köşeye, yani satır 0, sütun 0'a taşır.
`\e[0;41m`	Arka plan rengini kırmızı olarak ayarlayın.
`\e[K`	İmlecin bulunduğu konumdan (sol üst köşe) satırın sonuna kadar silin. Arka plan rengi artık kırmızı olduğu için satır bu renge silinir ve çubuğumuz oluşturulur. Satırın sonuna kadar silinmesi imlecin konumunu değiştirmez, imleç sol üst köşede kalır.
`\e[1;33m`	Metin rengini sarı olarak ayarlayın.
`\t`	Geçerli saati gösterir. Bu bir "yazdırma" öğesi olmasına rağmen, `bash'ın` görüntülenen komut isteminin gerçek boyutunu hesaplarken saati dahil etmesini istemediğimiz için, bunu komut isteminin yazdırılmayan kısmına dahil ediyoruz.
`\e[0m`	Rengi kapatır. Bu, hem metni hem de arka planı etkiler.
`\e[u`	Daha önce kaydedilen imleç konumunu geri yükler.
`\]`	Yazdırılmayan karakter dizisini sonlandırın.
`<\u@\h \W>\$`	Komut istemi dizesi.

İstem Kaydetme

Elbette, bu uzun komutu her seferinde yazmak istemiyoruz, bu yüzden komut istemini bir yere kaydetmek istiyoruz. Komut istemini `.`bashrc dosyasına ekleyerek kalıcı hale getirebiliriz. Bunu yapmak için, dosyaya şu iki satırı ekleyin:

```
PS1="\[\e[s\e[0;0H\e[0;41m\e[K\e[1;33m\t\e[0m\e[u\]
```

```
<\u@\h \W>\$ "

export PS1
```

Özet

İster inanın ister inanmayın, burada ele almadığımız kabuk işlevleri ve komut dosyaları içeren komut istemleriyle yapılabilecek çok daha fazla şey var, ancak bu iyi bir başlangıç. Varsayılan komut istemi genellikle tatmin edici olduğundan, herkes komut istemini değiştirmek için yeterince önem vermeyecektir. Ancak, kurcalamayı sevenler için kabuk, saatlerce süren rahat bir eğlence fırsatı sunar.

Daha Fazla Okuma

- Linux Belgeleme Projesi'nin *Bash Prompt HOWTO'su,* kabuk komut isteminin neler yapabileceğine dair oldukça kapsamlı bir tartışma sunar. Şu adreste bulabilirsiniz: https://tldp.org/HOWTO/Bash-Prompt-HOWTO/
- Wikipedia'da ANSI Kaçış Kodları hakkında iyi bir makale bulunmaktadır: https://en.wikipedia.org/wiki/ANSI_escape_code

Bölüm 3 – Yaygın Görevler ve Temel Araçlar

14 – Paket Yönetimi

Linux topluluğunda biraz zaman geçirirsek, birçok Linux dağıtımından hangisinin "en iyi" olduğuna dair birçok görüş duyarız. Genellikle bu tartışmalar, masaüstü arka planının güzelliği (bazı insanlar varsayılan renk şeması nedeniyle Ubuntu kullanmazlar!) ve diğer önemsiz konular gibi şeylere odaklanarak gerçekten saçma bir hal alır.

Dağıtımın kalitesini belirleyen en önemli faktör, *paketleme sistemi* ve dağıtımın destek topluluğunun canlılığıdır. Linux ile daha fazla zaman geçirdikçe, yazılım ortamının son derece dinamik olduğunu görürüz. Her şey sürekli değişiyor. En üst düzey Linux dağıtımlarının çoğu altı ayda bir yeni sürümler yayınlar ve birçok program her gün güncellenir. Bu yazılım fırtınasına ayak uydurmak için, *paket yönetimi* için iyi araçlara ihtiyacımız var.

Paket yönetimi, sisteme yazılım yükleme ve bakımını yapma yöntemidir. Günümüzde çoğu kişi, Linux dağıtımcısından *paketler* yükleyerek tüm yazılım ihtiyaçlarını karşılayabilir. Bu durum, yazılım yüklemek için *kaynak kodunu* indirip derlemek zorunda olduğumuz Linux'un ilk günleriyle tezat oluşturur. Kaynak kodunu derlemenin yanlış bir yanı yoktur; aslında, kaynak koduna erişim Linux'un en büyük mucizesidir. Bu, bize (ve herkese) sistemi inceleme ve geliştirme olanağı verir. Sadece önceden derlenmiş bir pakete sahip olmak daha hızlı ve daha kolaydır.

Bu bölümde, paket yönetimi için kullanılan bazı komut satırı araçlarına bakacağız. Tüm büyük dağıtımlar, sistemi korumak için güçlü ve sofistike grafik programları sağlasa da, komut satırı programları hakkında da bilgi edinmek önemlidir. Bu programlar, grafiksel karşılıklarıyla yapmak zor (veya imkansız) olan birçok görevi yerine getirebilirler.

Paketleme Sistemleri

Farklı dağıtımlar farklı paketleme sistemleri kullanır ve genel bir kural olarak, bir dağıtım için tasarlanmış bir paket başka bir dağıtımla uyumlu değildir. Çoğu dağıtım, iki paketleme teknolojisi grubundan birine girer: Debian .deb grubu ve Red Hat .rpm grubu. Gentoo, Slackware ve Arch gibi bazı önemli istisnalar vardır, ancak çoğu diğerleri Tablo 14-1'de gösterildiği gibi bu iki temel sistemden birini kullanır.

Tablo 14-1: Başlıca Paketleme Sistemi Aileleri

Paketleme sistemi	Dağıtımlar (kısmi liste)
Debian Stili (.deb)	Debian, Ubuntu, Linux Mint, Raspberry Pi OS
Red Hat Stili (.rpm)	Fedora, CentOS, Red Hat Enterprise Linux, OpenSUSE

Paket Sistemi Nasıl Çalışır?

Tescilli yazılım endüstrisinde kullanılan yazılım dağıtım yöntemi genellikle, "kurulum diski" gibi bir kurulum ortamı satın almayı veya bir satıcının web sitesini ziyaret edip ürünü indirip ardından "kurulum sihirbazı"nı çalıştırarak sisteme yeni bir uygulama yüklemeyi içerir.

Linux bu şekilde çalışmaz. Linux sistemi için neredeyse tüm yazılımlar İnternet'te bulunabilir. Bunların çoğu dağıtım satıcısı tarafından *paket dosyaları* biçiminde sağlanır, geri kalanı ise manuel olarak kurulabilen kaynak kodu biçiminde mevcuttur. Kaynak kodunu derleyerek yazılımın nasıl kurulacağına 23. bölümde değineceğiz.

Paket Dosyaları

Bir paketleme sistemindeki yazılımın temel birimi *paket dosyasıdır*. Paket dosyası, yazılım paketini oluşturan dosyaların sıkıştırılmış bir koleksiyonudur. Bir paket, programları destekleyen çok sayıda program ve veri dosyasından oluşabilir. Yükleme yapılacak dosyalara ek olarak, paket dosyası ayrıca paket ve içeriği hakkında metin açıklaması gibi paketle ilgili meta verileri de içerir. Ayrıca, birçok paket, paket yüklemesinden önce ve sonra yapılandırma görevlerini gerçekleştiren yükleme öncesi ve sonrası komut dosyaları içerir.

Paket dosyaları, genellikle (ancak her zaman değil) dağıtım satıcısının bir çalışanı olan *paket bakımcısı* olarak bilinen kişi tarafından oluşturulur. Paket bakımcısı, yazılımı kaynak kodu biçiminde *yukarı akış sağlayıcısından* (programın yazarı) alır, derler ve paket meta verilerini ve gerekli kurulum komut dosyalarını oluşturur. Genellikle, paket bakımcısı, programın Linux dağıtımının diğer bölümleriyle entegrasyonunu iyileştirmek için orijinal kaynak kodunda değişiklikler yapar.

Depolar

Bazı yazılım projeleri kendi paketleme ve dağıtımlarını yapmayı tercih ederken, günümüzde çoğu paket dağıtım satıcıları ve ilgilenen üçüncü taraflar tarafından oluşturulmaktadır. Paketler, dağıtım kullanıcılarına, her biri dağıtım için özel olarak oluşturulmuş ve bakımı yapılan binlerce paket içerebilen merkezi depolarda sunulur.

Bir dağıtım, yazılım geliştirme yaşam döngüsünün farklı aşamaları için birkaç farklı depo tutabilir.

Örneğin, genellikle, genel dağıtım için yayınlanmadan önce hataları bulmak isteyen cesur kişiler tarafından kullanılmak üzere, yeni oluşturulmuş paketleri içeren bir "test" deposu bulunur. Bir dağıtım, genellikle, dağıtımın bir sonraki büyük sürümüne dahil edilmek üzere geliştirilmekte olan paketlerin tutulduğu bir "geliştirme" deposuna sahiptir.

Bir dağıtım, ilgili üçüncü taraf depolarına da sahip olabilir. Bunlar genellikle, patentler veya DRM anti-atlatma sorunları gibi yasal nedenlerle dağıtıma dahil edilemeyen yazılımları sağlamak için gereklidir. Belki de en iyi bilinen örnek, Amerika Birleşik Devletleri'nde yasal olmayan şifreli DVD desteğidir. Üçüncü taraf depoları, yazılım patentleri ve anti-atlatma yasalarının geçerli olmadığı ülkelerde faaliyet gösterir. Bu depolar genellikle destekledikleri dağıtımdan tamamen bağımsızdır ve bunları kullanmak için, bunların varlığından haberdar olmak ve bunları paket yönetim sisteminin yapılandırma dosyalarına manuel olarak eklemek gerekir.

Bağımlılıklar

Programlar nadiren "bağımsız"dır; aksine, işlerini yapmak için diğer yazılım bileşenlerinin varlığına bağlıdırlar. Örneğin giriş/çıkış gibi yaygın faaliyetler, birçok program tarafından paylaşılan rutinler tarafından gerçekleştirilir. Bu rutinler, birden fazla programa temel hizmetler sağlayan *paylaşılan kütüphaneler* olarak adlandırılan yerlerde saklanır. Bir paket, paylaşılan kütüphane gibi paylaşılan bir kaynağa ihtiyaç duyuyorsa, *bağımlılığı* olduğu söylenir. Modern paket yönetim sistemlerinin tümü, bir paket yüklendiğinde tüm bağımlılıklarının da yüklenmesini sağlamak için bir tür *bağımlılık çözme* yöntemi sunar.

Yüksek ve Düşük Seviyeli Paket Araçları

Paket yönetim sistemleri genellikle iki tür araçtan oluşur.

- Paket dosyalarının yüklenmesi ve kaldırılması gibi görevleri gerçekleştiren düşük seviyeli araçlar
- Meta veri arama ve bağımlılık çözümü gerçekleştiren üst düzey araçlar

Bu bölümde, Debian tarzı sistemlerle (Ubuntu ve diğerleri gibi) birlikte sağlanan araçları ve Red Hat ürünlerinde kullanılan araçları inceleyeceğiz. Tüm Red Hat tarzı dağıtımlar aynı düşük seviyeli programa (`rpm`) dayanmakla birlikte, farklı yüksek seviyeli araçlar kullanırlar. Bu bölümde, Red Hat Enterprise Linux, CentOS ve Fedora tarafından kullanılan yüksek seviyeli `dnf` programını ele alacağız. Diğer Red Hat tarzı dağıtımlar da benzer özelliklere sahip yüksek seviyeli araçlar sağlar (bkz. Tablo 14-2).

Tablo 14- 2: Paketleme Sistemi Araçları

Dağıtımlar	Düşük seviyeli araçlar	Üst düzey araçlar

Debian stili	`dpkg`	`apt, apt-get, aptitude`
Fedora, Red Hat Enterprise Linux, CentOS	`rpm`	`dnf, yum`

Yaygın Paket Yönetimi Görevleri

Komut satırı paket yönetim araçlarıyla birçok işlem gerçekleştirilebilir. En yaygın olanlara bakacağız. Düşük seviyeli araçların, bu kitabın kapsamı dışında kalan bir faaliyet olan paket dosyalarının oluşturulmasını da desteklediğini unutmayın.

Aşağıdaki tartışmada, *`package_name`* terimi, paketi içeren dosyanın adı olan *`package_file`* terimi yerine, paketin gerçek adını ifade eder. Ayrıca, herhangi bir paket işlemi gerçekleştirilmeden önce, paket deposu sorgulanarak veritabanının yerel kopyası senkronize edilmelidir. Red Hat'in `dnf` programı bunu otomatik olarak yapar ve son güncellemeden bu yana çok fazla zaman geçmişse yerel veritabanını günceller. Öte yandan, Debian'ın `apt` programı, yerel veritabanını açıkça güncellemek için update komutuyla çalıştırılmalıdır. Bu, sık sık yapılmalıdır. Aşağıdaki örneklerde, `apt update` komutu herhangi bir işlemden önce yapılır, ancak gerçek hayatta bu, güvenliği sağlamak için sadece birkaç saatte bir yapılmalıdır.

Sistem genelinde yazılım yükleme veya kaldırma işlemleri bir yönetim görevi olduğundan, paket yönetim aracından bağımsız olarak süper kullanıcı ayrıcalıkları gereklidir.

Depoda Paket Bulma

Yüksek düzey araçları kullanarak depo meta verilerini aramak suretiyle, bir paket adı veya açıklamasına göre bulunabilir (bkz. Tablo 14-3).

Tablo 14-3: Paket Arama Komutları

Stil	Komut(lar)
Debian	`apt update; apt search arama_dizesi`
Red Hat	`dnf search arama_dizesi`

Örneğin, emacs metin editörü için bir `dnf` deposunda arama yapmak için şu komutu kullanabiliriz:

```
dnf search emacs
```

Depodan Paket Yükleme

Üst düzey araçlar, bir paketin depodan indirilmesini ve tüm bağımlılıkların çözülmesiyle kurulmasını sağlar (bkz. Tablo 14-4).

Tablo 14-4: Paket Yükleme Komutları

Stil	Komut(lar)
Debian	`apt update; apt install` *`paket_adı`*
Red Hat	`dnf install` *`paket_adı`*

Örneğin, Debian sisteminde `apt` deposundan emacs metin düzenleyicisini yüklemek için şu komutu kullanabiliriz:

```
apt güncelle; apt emacs yükle
```

Paket Dosyasından Paket Yükleme

Bir paket dosyası depo dışındaki bir kaynaktan indirilmişse, düşük seviyeli bir araç kullanılarak (bağımlılık çözümü olmadan) doğrudan yüklenebilir (bkz. Tablo 14-5).

Tablo 14-5: Düşük Seviyeli Paket Yükleme Komutları

Stil	Komut(lar)
Debian	`dpkg -i` *`paket_dosyası`*
Red Hat	`rpm -i` *`paket_dosyası`*

Örneğin, `emacs-22.1-7.fc7-i386.rpm` paket dosyası depo dışındaki bir siteden indirilmişse, şu şekilde kurulur:

```
rpm -i emacs-22.1-7.fc7-i386.rpm
```

Not: Bu teknik, kurulumu gerçekleştirmek için düşük seviyeli `rpm` programını kullandığından, bağımlılık çözümü gerçekleştirilmez. `Rpm,` eksik bir bağımlılık tespit ederse

bağımlılık tespit ederse, `rpm` bir hata ile sonlandırılır.

Paket Kaldırma

Paketler, üst düzey veya alt düzey araçlar kullanılarak kaldırılabilir. Tablo 14-6, üst düzey araçları göstermektedir.

Tablo 14-6: Paket Kaldırma Komutları

Stil	Komut(lar)
Debian	`apt remove` *`paket_adı`*
Red Hat	`dnf erase` *`paket_adı`*

Örneğin, Debian tarzı bir sistemden emacs paketini kaldırmak için şu komutu kullanabiliriz:

```
apt remove emacs
```

Depodan Paketleri Güncelleme

En yaygın paket yönetimi görevi, sistemi en son paket sürümleriyle güncel tutmaktır. Üst düzey araçlar bu önemli görevi tek bir adımda gerçekleştirebilir (bkz. Tablo 14-7).

Tablo 14-7: Paket Güncelleme Komutları

Stil	Komut(lar)
Debian	`apt güncelle; apt yükselt`
Red Hat	`dnf update`

Örneğin, Debian tarzı bir sistemde yüklü paketlere mevcut tüm güncellemeleri uygulamak için şu komutu kullanabiliriz:

```
apt güncelle; apt yükselt
```

Paket Dosyasından Paket Yükseltme

Bir paketin güncellenmiş sürümü depo dışı bir kaynaktan indirilmişse, önceki sürümü değiştirerek yüklenebilir (bkz. Tablo 14-8).

Tablo 14-8: Düşük Seviyeli Paket Yükseltme Komutları

Stil	Komut(lar)
Debian	`dpkg -i paket_dosyası`
Red Hat	`rpm -U paket_dosyası`

Örneğin, Red Hat sisteminde mevcut emacs kurulumunu `emacs-22.1-7.fc7-i386.rpm` paket dosyasında bulunan sürüme güncellemek için şu komutu kullanabiliriz:

```
rpm -U emacs-22.1-7.fc7-i386.rpm
```

Not: `dpkg, rpm'nin` yaptığı gibi bir paketi yükseltmek için özel bir seçeneğe sahip değildir.

Yüklü Paketleri Listeleme

Tablo 14-9, sistemde kurulu tüm paketlerin listesini görüntülemek için kullanabileceğimiz komutları listeler.

Tablo 14-9: Paket Listeleme Komutları

Stil	Komut(lar)
Debian	`dpkg -l`
Red Hat	`rpm -qa`

Bir Paketin Yüklü Olup Olmadığını Belirleme

Tablo 14-10, belirli bir paketin yüklü olup olmadığını görüntülemek için kullanabileceğimiz düşük seviyeli araçları listeler.

Tablo 14-10: Paket Durumu Komutları

Stil	Komut(lar)
Debian	`dpkg -s paket_adı`
Red Hat	`rpm -q paket_adı`

Örneğin, emacs paketinin Debian tarzı bir sistemde kurulu olup olmadığını belirlemek için şu komutu kullanabiliriz:

```
dpkg --status emacs
```

Yüklü Bir Paket Hakkında Bilgi Görüntüleme

Yüklü bir paketin adı biliniyorsa, Tablo 14-11'deki komutları kullanarak paketin açıklamasını görüntüleyebiliriz.

Tablo 14-11: Paket Bilgisi Komutları

Stil	Komut(lar)
Debian	`apt show paket_adı`
Red Hat	`dnf info paket_adı`

Örneğin, Debian tarzı bir sistemde emacs paketinin açıklamasını görmek için şu komutu kullanabiliriz:

```
apt-cache show emacs
```

Bir Dosyayı Yükleyen Paketi Bulma

Belirli bir dosyanın yüklenmesinden hangi paketin sorumlu olduğunu belirlemek için Tablo 14-12'deki komutları kullanabiliriz.

Tablo 14-12: Paket Dosyası Tanımlama Komutları

Stil	Komut(lar)
Debian	`dpkg -S dosya_adı`

Red Hat	`rpm -qf dosya_adı`

Örneğin, Red Hat sisteminde `/usr/bin/vim` dosyasını hangi paketin yüklediğini görmek için aşağıdakileri kullanabiliriz:

```
rpm -qf /usr/bin/vim
```

Dağıtımdan Bağımsız Paket Biçimleri

Son birkaç yıldır dağıtım satıcıları, belirli bir Linux dağıtımına bağlı olmayan evrensel paket formatları piyasaya sürdüler. Bunlar arasında Snaps (Canonical tarafından geliştirilen ve tanıtılan), Flatpaks (Red Hat tarafından geliştirilen, ancak şu anda yaygın olarak kullanılabilen) ve AppImages bulunmaktadır. Her biri biraz farklı çalışsa da, amaçları bir uygulamayı ve tüm bağımlılıklarını bir araya getirip tek bir parça halinde yüklemektir. Bu tamamen yeni bir fikir değildir. Linux'un ilk günlerinde (1990'ların sonlarını düşünün) bir uygulamayı ve gerekli kütüphaneleri tek bir büyük ikili dosyada birleştiren *statik bağlama* adı verilen bir teknik vardı.

Bu paketleme yaklaşımının bazı avantajları vardır. Bunların başında, bir uygulamayı dağıtmak için gereken çabayı azaltması gelir. Uygulamayı, dağıtımın temel sisteminde bulunan kütüphaneler ve diğer destek dosyalarıyla çalışacak şekilde uyarlamak yerine, uygulama bir kez derlenir ve herhangi bir sisteme yüklenebilir. Bu formatlardan bazıları, ek güvenlik sağlamak için uygulamayı konteynerize bir sanal alanda çalıştırır.

Ancak bazı ciddi dezavantajları da vardır. Bu şekilde paketlenmiş uygulamalar büyüktür. Bazen *gerçekten çok* büyüktür. Bunun iki etkisi vardır. İlk olarak, depolamak için çok fazla disk alanı gerektirirler. İkincisi, büyük boyutları nedeniyle yüklemeleri çok yavaş olabilir. Bu, modern ultra hızlı donanımlarda çok büyük bir sorun olmayabilir, ancak eski makinelerde gerçek bir sorundur. Bir sonraki teknik sorun, dağıtım entegrasyonu ile ilgilidir. Bu uygulamalar tüm içeriklerini beraberlerinde getirdikleri için, altta yatan dağıtımın olanaklarından yararlanamazlar. Bazen konteynerleştirilmiş uygulama, optimum performans için gerekli sistem kaynaklarına erişemez.

Bir de felsefi sorunlar vardır. Bu hepsi bir arada uygulama paketlerinden belki de en çok yararlananlar, özel yazılım satıcılarıdır. Bir kez Linux sürümü oluşturduklarında, tüm dağıtımlar bunu kullanabilir. Uygulamalarını farklı dağıtımlar için özel olarak uyarlamalarına gerek yoktur.

Kullanıcılar bu paketleme formatlarını istemiyorlardı ve bu formatlar açık kaynak topluluğunu geliştirmek için çok az şey yapıyorlar. Bu nedenle, çeşitli performans sorunları çözülene kadar bu formatların kullanılması önerilmez.

Özet

İlerleyen bölümlerde, çok çeşitli uygulama alanlarını kapsayan birçok farklı programı inceleyeceğiz. Bu programların çoğu varsayılan olarak yüklenmiş olsa da, gerekli programlar sağlanmamışsa ek paketler yüklememiz gerekebilir. Paket yönetimi konusunda edindiğimiz yeni bilgiler (ve takdir) sayesinde, ihtiyacımız olan programları yüklemek ve yönetmek konusunda hiçbir sorun yaşamayız.

Linux Yazılım Yükleme Efsanesi

Diğer platformlardan geçiş yapan kişiler bazen, Linux altında yazılım yüklemenin bir şekilde zor olduğu ve farklı dağıtımlar tarafından kullanılan çeşitli paketleme şemalarının bir engel olduğu mitine kapılırlar. Evet, bu bir engeldir, ancak yalnızca gizli yazılımlarının yalnızca ikili sürümlerini dağıtmak isteyen tescilli yazılım satıcıları için.

Linux yazılım ekosistemi, açık kaynak kodu fikrine dayanmaktadır. Bir program geliştiricisi bir programın kaynak kodunu yayınlarsa, dağıtımla ilişkili bir kişi programı paketleyip kendi deposuna ekleyecektir. Bu yöntem, programın dağıtıma iyi bir şekilde entegre olmasını sağlar ve kullanıcıya, her programın web sitesini aramak yerine, yazılım için "tek noktadan alışveriş" kolaylığı sunar. Son zamanlarda, büyük özel platform satıcıları bu fikri taklit eden uygulama mağazaları kurmaya başladı.

Aygıt sürücüleri de benzer şekilde ele alınır, ancak dağıtımın deposunda ayrı öğeler olmak yerine Linux çekirdeğinin bir parçası olurlar. Genel olarak, Linux'ta "sürücü diski" diye bir şey yoktur. Çekirdek bir aygıtı destekler ya da desteklemez ve Linux çekirdeği birçok aygıtı destekler. Aslında, Windows'tan çok daha fazlasını destekler. Tabii ki, ihtiyacınız olan belirli bir aygıt desteklenmiyorsa, bu bir teselli değildir. Böyle bir durumda, nedenini araştırmanız gerekir. Sürücü desteğinin olmaması genellikle üç nedenden birine bağlıdır:

1. Aygıt çok yenidir. Birçok donanım satıcısı Linux geliştirmesini aktif olarak desteklemediğinden, çekirdek sürücü kodunu yazmak Linux topluluğunun bir üyesine düşer. Bu zaman alır.

2. Aygıt çok egzotik. Tüm dağıtımlar, olası her aygıt sürücüsünü içermez. Her dağıtım kendi çekirdeklerini oluşturur ve çekirdekler çok yapılandırılabilir olduğundan (bu da Linux'un kol saatlerinden ana bilgisayarlara kadar her şeyde çalışmasını mümkün kılar), belirli bir aygıt gözden kaçmış olabilir. Sürücünün kaynak kodunu bulup indirerek, sürücüyü kendiniz derleyip yükleyebilirsiniz (evet, siz). Bu işlem çok zor değildir, ancak oldukça karmaşıktır. Yazılım derlemeyi sonraki bir bölümde ele alacağız.

3. Donanım satıcısı bir şey saklıyor. Linux sürücüsü için kaynak kodunu yayınlamamış ve birisinin sürücü oluşturması için teknik belgeleri de yayınlamamış. Bu, donanım satıcısının cihaza ait programlama arayüzlerini gizli tutmaya çalıştığı anlamına gelir. Bilgisayarlarımızda gizli cihazlar istemediğimiz için, bu tür ürünlerden kaçınmanız en iyisidir.

Daha Fazla Okuma

Dağıtımınızın paket yönetim sistemini tanımak için biraz zaman ayırın. Her dağıtım, paket yönetim araçları için belgeler sağlar. Ayrıca, daha genel kaynaklar da vardır:

- *Debian GNU/Linux SSS* bölümünün paket yönetimi ile ilgili kısmı, Debian sistemlerinde paket yönetiminin genel bir özetini sunar: https://www.debian.org/doc/manuals/debian-faq/pkg-basics.en.html
- RPM projesinin ana sayfası: http://www.rpm.org
- Biraz arka plan bilgisi için, Wikipedia'da meta verilerle ilgili bir makale bulunmaktadır: http://en.wikipedia.org/wiki/Metadata
- Snap, Flatpak ve AppImage formatlarını karşılaştıran iyi bir makale: https://www.baeldung.com/linux/snaps-flatpak-appimage

15 – Depolama Ortamları

Önceki bölümlerde dosya düzeyinde veri işlemeyi inceledik. Bu bölümde, aygıt düzeyinde verileri ele alacağız. Linux, sabit diskler, ağ depolama gibi fiziksel depolama aygıtları veya RAID (Redundant Array of Independent Disks) ve LVM (Logical Volume Manager) gibi sanal depolama aygıtlarını yönetme konusunda inanılmaz yeteneklere sahiptir.

Ancak, bu kitap sistem yönetimi hakkında olmadığı için, bu konuyu derinlemesine ele almayacağız. Yapmaya çalışacağımız şey, depolama aygıtlarını yönetmek için kullanılan bazı kavramları ve temel komutları tanıtmaktır.

Bu bölümdeki alıştırmaları yapmak için bir USB flash sürücü, bir USB sabit disk sürücüsü ve bir DVD/CD-RW diski (CD-ROM yazıcı ile donatılmış sistemler için) kullanacağız.

Aşağıdaki komutları inceleyeceğiz:

- `mount` – Dosya sistemini bağlar
- `umount` – Dosya sistemini bağlantısını kesmek
- `parted` – Bölüm işleme programı
- `mkfs` – Dosya sistemi oluşturma
- `fsck` – Dosya sistemini kontrol etme ve onarma
- `dd` – Dosya dönüştürme ve kopyalama
- `genisoimage` – ISO 9660 görüntü dosyası oluşturma
- `wodim` – Optik depolama ortamına veri yazma
- `sha256sum` – SHA256 sağlama toplamlarını hesaplama ve kontrol etme

Depolama Aygıtlarını Takma ve Çıkarma

Linux masaüstündeki son gelişmeler, masaüstü kullanıcıları için depolama aygıtı yönetimini son derece kolay hale getirmiştir. Çoğu durumda, bir aygıtı sistemimize bağladığımızda "sadece çalışır". Eskiden (örneğin 2004'te), bu işlemlerin manuel olarak yapılması gerekiyordu. Masaüstü olmayan sistemlerde (yani sunucularda) bu işlem hala büyük ölçüde manuel bir prosedürdür, çünkü sunucular genellikle aşırı

aşırı depolama ihtiyaçları ve karmaşık yapılandırma gereksinimleri vardır.

Bir depolama aygıtını yönetmenin ilk adımı, aygıtı dosya sistemi ağacına eklemektir. *Bağlama olarak* adlandırılan bu işlem, aygıtın işletim sistemi ile etkileşime girmesini sağlar. Bölüm 2'de hatırladığımız gibi, Linux gibi Unix benzeri işletim sistemleri, çeşitli noktalara bağlı aygıtlarla tek bir dosya sistemi ağacı tutar. Bu, her aygıt için ayrı dosya sistemi ağaçları tutan MS-DOS ve Windows gibi diğer işletim sistemlerinden farklıdır (örneğin `C:\`, `D:\`, vb.).

`/etc/fstab` ("dosya sistemi tablosu"nun kısaltması) adlı bir dosya, önyükleme sırasında bağlanacak aygıtları (genellikle sabit disk bölümleri) listeler. İşte eski bir Fedora sisteminden bir `/etc/fstab` dosyası örneği:

```
LABEL=/12           /                       ext4    varsayılan        1 1
LABEL=/home         /home                   ext4    varsayılan        1 2
LABEL=/boot         /boot                   ext4    varsayılan        1 2
tmpfs               /dev/shm                tmpfs   varsayılan        0 0
devpts              /dev/pts                devpts  gid=5,mod=620     0 0
sysfs               /sys                    sysfs   varsayılanlar     0 0
proc                /proc                   proc    varsayılan        0 0
LABEL=SWAP-sda3     swap                    swap    varsayılan        0 0
```

Bu örnek dosyada listelenen dosya sistemlerinin çoğu sanaldır ve bizim konumuzla ilgisi yoktur. Bizim amacımız için ilginç olanlar ilk üçüdür:

```
LABEL=/12           /                       ext4    varsayılan        1 1
LABEL=/home         /home                   ext4    varsayılan        1 2
LABEL=/boot         /boot                   ext4    varsayılan        1 2
```

Bunlar sabit disk bölümleridir. Dosyanın her satırı, Tablo 15-1'de açıklandığı gibi altı alandan oluşur.

Tablo 15-1: `/etc/fstab` Alanları

Alan	İçerik	Açıklama
1	Aygıt	Geleneksel olarak, bu alan fiziksel cihazla ilişkili /dev/sda1 (ilk algılanan ilk bölüm) gibi fiziksel aygıtla ilişkili aygıt dosyasının gerçek adını içerir. `/dev/sda1` (algılanan ilk sabit diskin ilk bölümü sabit diskin ilk bölümü). Ancak, günümüzün bilgisayarlarında USB sürücüler gibi birçok cihaz, birçok modern Linux dağıtımı, bir aygıtı

		bir metin etiketi ile ilişkilendirir. Bu etiket (biçimlendirildiğinde depolama ortamına eklenir) basit bir metin etiketi veya rastgele oluşturulan bir UUID (Evrensel Benzersiz Tanımlayıcı) olabilir. Bu etiket, aygıt sisteme bağlandığında işletim sistemi tarafından okunur. Bu şekilde, gerçek fiziksel aygıta hangi aygıt dosyası atanmış olursa olsun, yine de doğru şekilde tanımlanabilir.
2	Bağlantı noktası	Aygıtın dosya sistemi ağacına bağlandığı dizin.
3	Dosya sistemi türü	Linux, birçok dosya sistemi türünün bağlanmasına izin verir. Çoğu Linux sistemi, Fourth Extended Filesystem (`ext4`) adlı yerel bir Linux dosya sistemi kullanır, ancak FAT16 (`msdos`), FAT32 (`vfat`), NTFS (`ntfs`), CD-ROM (`iso9660`) vb.
4	Seçenekler	Dosya sistemleri çeşitli seçeneklerle bağlanabilir. Örneğin, dosya sistemlerini salt okunur olarak bağlamak veya bunlardan herhangi bir programın çalıştırılmasını engellemek mümkündür (çıkarılabilir ortamlar için yararlı bir güvenlik özelliği).
5	Sıklık	Bir dosya sisteminin `dump` komutuyla yedeklenip yedeklenmeyeceğini ve ne zaman yedekleneceğini belirten tek bir sayı.
6	Sıra	fsck komutuyla dosya sistemlerinin hangi sırayla kontrol edileceğini belirten tek bir sayı. Bu konu hakkında daha fazla bilgi bu bölümün ilerleyen kısımlarında verilecektir.

Takılı Dosya Sistemlerinin Listesini Görüntüleme

`mount` komutu, dosya sistemlerini bağlamak için kullanılır. Komutu argüman olmadan girdiğinizde, o anda bağlı olan dosya sistemlerinin listesi görüntülenir:

```
[me@linuxbox ~]$ mount
/dev/sda2 on / type ext4 (rw)
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
devpts on /dev/pts type devpts (rw,gid=5,mode=620)
```

```
/dev/sda5 /home üzerinde ext4 (rw) türü
/dev/sda1 /boot üzerinde ext4 (rw)
tmpfs /dev/shm üzerinde tmpfs (rw)
none on /proc/sys/fs/binfmt_misc type binfmt_misc (rw)
sunrpc on /var/lib/nfs/rpc_pipefs type rpc_pipefs (rw)
fusectl on /sys/fs/fuse/connections type fusectl (rw)
/dev/sdd1 on /media/disk type vfat (rw,nosuid,nodev,noatime,
uhelper=hal,uid=500,utf8,shortname=lower)
twin4:/musicbox /misc/musicbox üzerinde nfs4 (rw,addr=192.168.1.4) türü
```

İpucu: Ubuntu kullanıcıları, bu listede sistemde yüklü snap paketleri tarafından oluşturulan "loop" aygıtlarının da yer aldığını göreceklerdir. Daha temiz bir liste görmek için şu komutları deneyin: `mount | grep -v snap` veya `mount | grep ^/dev`

Listenin formatı şu şekildedir: *aygıt* on *mount_point* type *file_system_type* (*op-tions*). Örneğin, ilk satırda `/dev/sda2` aygıtının kök dosya sistemi olarak bağlandığı, ext4 tipi olduğu ve hem okunabilir hem de yazılabilir olduğu ("rw" seçeneği) gösterilmektedir. Bu listenin altında da iki ilginç giriş bulunmaktadır. Sonuncu girişten önceki giriş, `/media/disk` konumuna takılı bir kart okuyucudaki 2 GB'lık bir flash bellek kartını gösterir ve son giriş ise `/misc/musicbox` konumuna takılı bir ağ sürücüsüdür.

İlk denememiz için 4 GB'lık bir flash sürücü kullanacağız. Öncelikle, sürücü takılmadan önce Ubuntu 22.04 sistemine bir göz atalım:

```
[me@linuxbox ~]$ mount | grep /dev/sd
/dev/sda2 on / type ext4 (rw,relatime,errors=remount-ro)
/dev/sda1 /boot/efi üzerinde vfat türü
(rw,relatime,fmask=0077,dmask=0077, codepage=437,iocharset=iso8859-
1,shortname=mixed,errors=remount-ro)
/dev/sdb1 /home üzerinde ext4 (rw,relatime) türü
```

Netlik sağlamak için, çıktıyı `grep` aracılığıyla `/dev/sd*` aygıtlarını listelemek üzere yönlendirdik. Bu sistemde `/dev/sda` ve `/dev/sdb` olmak üzere iki sabit disk olduğunu görebiliriz. `/dev/sda` üzerinde iki bölüm vardır. Bunlar `/dev/sda1` ve `/dev/sda2` iken, `/dev/sdb` üzerinde tek bir bölüm, `/dev/sdb1` vardır.

Birçok modern Linux dağıtımı gibi, bu sistem de flash sürücüyü takıldıktan sonra otomatik olarak bağlamaya çalışacaktır. Sürücüyü taktıktan sonra şunu görürüz:

```
[me@linuxbox ~]$ mount | grep /dev/sd
```

```
dev/sda2 on / type ext4 (rw,relatime,errors=remount-ro)
/dev/sda1 /boot/efi üzerinde vfat türü
(rw,relatime,fmask=0077,dmask=0077,codepage=437,iocharset=iso8859-
1,shortname=mixed,errors=remount-ro)
/dev/sdb1 /home üzerinde ext4 türü (rw,relatime)
/dev/sdc /media/me/C911-C314 üzerinde vfat türü
(rw,nosuid,nodev,relatime,
uid=1000,gid=1000,fmask=0022,dmask=0022,codepage=437,iocharset=iso885 9-
1,shortname=mixed,showexec,utf8,flush,errors=remount-ro)
```

Sürücüyü taktıktan sonra, bir önceki listeye ek olarak bir giriş daha eklenmiş olarak aynı listeyi görüyoruz. Listenin sonunda, flash sürücünün (bu sistemde `/dev/sdc` aygıtı) `/media/me/C911-C314` konumuna takıldığını ve vfat (FAT32 olarak da bilinir, Windows uyumlu bir dosya sistemi) türü olduğunu görüyoruz. Deneyimizin amacı doğrultusunda, cihazın adı ile ilgileniyoruz. Bu deneyi kendiniz yaptığınızda, cihaz adı büyük olasılıkla farklı olacaktır.

Uyarı: Aşağıdaki örneklerde, sisteminizde kullanılan gerçek aygıt adlarına çok dikkat etmeniz ve **bu metinde kullanılan adları kullanmamanız** çok önemlidir**!**

Flash sürücünün cihaz adını öğrendiğimize göre, sürücüyü kaldırıp dosya sistemi ağacındaki başka bir konuma yeniden takalım. Bunu yapmak için, süper kullanıcı oluyoruz (sistemimize uygun komutu kullanarak) ve `umount` (yazımına dikkat edin) komutuyla sürücüyü kaldırıyoruz.

```
[me@linuxbox ~]$ sudo -i [sudo]
benim için şifre:
[root@linuxbox ~]# umount /dev/sdc
```

Bir sonraki adım, disk için yeni bir *bağlama noktası* oluşturmaktır. Bağlama noktası, dosya sistemi ağacındaki herhangi bir yerdeki bir dizindir. Özel bir yanı yoktur. Boş bir dizin olması bile gerekmez, ancak boş olmayan bir dizine bir aygıt bağlarsanız, aygıtı bağlantısını kesene kadar dizinin önceki içeriğini göremezsiniz. Amacımız için yeni bir dizin oluşturacağız.

```
[root@linuxbox ~]# mkdir /mnt/flash
```

Son olarak, flash sürücüyü yeni bağlama noktasına bağlarız. `-t` seçeneği, dosya sistemi türünü belirtmek için kullanılır.

```
[root@linuxbox ~]# mount -t vfat /dev/sdc /mnt/flash
```

Ardından, yeni bağlama noktası üzerinden flash sürücünün içeriğini inceleyebiliriz.

```
[root@linuxbox ~]# cd /mnt/flash
[root@linuxbox cdrom]# ls
```

Sürücüyü kaldırmaya çalıştığımızda ne olduğunu dikkat edin.

```
[root@linuxbox cdrom]# umount /dev/sdc
umount: /mnt/flash: aygıt meşgul
```

Bunun nedeni nedir? Bunun nedeni, bir aygıt başka biri veya başka bir işlem tarafından kullanılıyorsa, o aygıtın bağlantısını kesemememizdir. Bu durumda, çalışma dizinimizi flash sürücünün bağlantı noktasına değiştirdik, bu da aygıtın meşgul olmasına neden oldu. Çalışma dizinini bağlantı noktasından başka bir yere değiştirerek bu sorunu kolayca çözebiliriz:

```
[root@linuxbox cdrom]# cd [root@linuxbox
~]# umount /dev/sdc
```

Artık aygıt başarıyla sökülmüştür.

Bağlantıyı Kesmek Neden Önemlidir?

Bellek kullanımıyla ilgili istatistikleri gösteren `free` komutunun çıktısına bakarsanız, *buffers* adlı bir istatistik göreceksiniz. Bilgisayar sistemleri mümkün olduğunca hızlı çalışacak şekilde tasarlanmıştır. Sistem hızını engelleyen faktörlerden biri yavaş cihazlardır. Yazıcılar buna iyi bir örnektir. En hızlı yazıcı bile bilgisayar standartlarına göre son derece yavaştır. Bir bilgisayar, yazıcının bir sayfayı yazdırmasını beklemek zorunda kalırsa gerçekten çok yavaş olur. PC'lerin ilk günlerinde (çoklu görev özelliği olmadan önce) bu gerçek bir sorundu. Bir elektronik tablo veya metin belgesi üzerinde çalışıyorsanız, her yazdırdığınızda bilgisayar durur ve kullanılamaz hale gelirdi. Bilgisayar, verileri yazıcının kabul edebileceği en yüksek hızda yazıcıya gönderirdi, ancak yazıcılar çok hızlı yazdırmadıkları için bu işlem çok yavaştı. Bu sorun, bilgisayar ile yazıcı arasına yerleştirilen ve bir miktar RAM belleği içeren bir aygıt olan *yazıcı tamponunun* ortaya çıkmasıyla çözüldü.

Yazıcı arabelleği takıldığında, bilgisayar yazıcı çıktısını arabelleğe gönderir ve bu veriler hızlı RAM'de hızla depolanır, böylece bilgisayar beklemeden çalışmaya devam edebilir. Bu arada, yazıcı arabelleği, yazıcının kabul edebileceği hızda arabelleğin belleğinden yazıcıya verileri yavaşça *aktarır*.

Bu tamponlama fikri, bilgisayarları daha hızlı hale getirmek için yaygın olarak kullanılır. Ara sıra yavaş cihazlardan veri okuma veya yazma ihtiyacının sistemin hızını engellemesine izin vermeyin. İşletim sistemleri, daha yavaş cihazlarla etkileşime girmeden önce, depolama cihazlarından okunan ve yazılacak verileri mümkün olduğunca uzun süre bellekte saklar. Örneğin, bir Linux sisteminde, sistem ne kadar uzun süre kullanılırsa belleği o kadar doldurur gibi görünecektir. Bu, Linux'un tüm belleği "kullandığı" anlamına gelmez; Linux'un mevcut tüm belleği mümkün olduğunca tamponlama yapmak için kullandığı anlamına gelir.

Bu tamponlama, fiziksel aygıta yazma işleminin ileri bir zamana ertelenmesi sayesinde depolama aygıtlarına çok hızlı yazma yapılmasına olanak tanır. Bu arada, aygıta gönderilecek veriler bellekte birikir. İşletim sistemi zaman zaman bu verileri fiziksel aygıta yazar.

Bir aygıtın bağlantısını kesmek, aygıta kalan tüm verilerin yazılmasını gerektirir, böylece aygıt güvenli bir şekilde çıkarılabilir. Aygıtın bağlantısı kesilmeden çıkarılması durumunda, aygıta gönderilen tüm verilerin aktarılmamış olma ihtimali vardır. Bazı durumlarda, bu veriler önemli dizin güncellemelerini içerebilir ve bu da bilgisayar için en kötü durumlardan biri olan *dosya sistemi bozulmasına* yol açabilir.

Aygıt Adlarını Belirleme

Bazen bir aygıtın adını belirlemek zor olabilir. Eskiden bu çok zor değildi. Aygıtlar her zaman aynı yerdeydi ve değişmezdi. Unix benzeri sistemler bu şekilde çalışır. Unix geliştirildiğinde, "disk sürücüsünü değiştirmek" bilgisayar odasından çamaşır makinesi büyüklüğünde bir cihazı forkliftle çıkarmak anlamına geliyordu. Son yıllarda, tipik masaüstü donanım yapılandırması oldukça dinamik hale geldi ve Linux, öncüllerinden daha esnek hale geldi.

Yukarıdaki örneklerde, modern Linux masaüstünün cihazı "otomatik olarak" bağlama ve ardından adı belirleme özelliğinden yararlandık. Peki, bunu yapamayacağımız bir sunucu veya başka bir ortam yönetiyorsak ne yapmalıyız? Bunu nasıl anlayabiliriz?

Öncelikle, sistemin aygıtlara nasıl isim verdiğine bakalım. /dev dizinindeki (tüm aygıtların bulunduğu yer) içeriği listelersek, çok sayıda aygıt olduğunu görebiliriz.

```
[me@linuxbox ~]$ ls /dev
```

Bu listenin içeriği, aygıt adlandırma konusunda bazı kalıpları ortaya koymaktadır. Tablo 15-2, bu kalıplardan birkaçını özetlemektedir.

Tablo 15-2: Linux Depolama Aygıtı Adları

Örüntü	Aygıt
/dev/fd*	Disket sürücüleri.
/dev/hd*	Eski sistemlerdeki IDE (PATA) diskler. Bu sistemlerdeki anakartlar, her biri sürücüler için iki bağlantı noktasına sahip bir kabloya sahip iki IDE konektörü veya *kanalı* içerir. Kablodaki ilk sürücü *ana* aygıt, ikincisi ise *bağımlı* aygıt olarak adlandırılır. Aygıt adları şu şekilde sıralanır /dev/hda ilk kanaldaki ana aygıtı, /dev/hdb ilk kanaldaki slave aygıtıdır; /dev/hdc ikinci kanaldaki master aygıtıdır ve bu şekilde devam eder. Son rakam, aygıt üzerindeki bölüm numarasını gösterir. Örneğin, /dev/hda1 sistemdeki ilk sabit sürücünün ilk bölümünü, /dev/hda ise tüm sürücüyü ifade eder.
/dev/lp*	Yazıcılar.
/dev/sd*	SCSI diskler. Modern Linux sistemlerinde, çekirdek tüm disk benzeri aygıtları (PATA/SATA sabit diskler, flash sürücüler ve taşınabilir müzik çalarlar ve dijital kameralar gibi USB yığın depolama aygıtları dahil) SCSI diskler olarak ele alır. Adlandırma sisteminin geri kalanı, yukarıda açıklanan eski /dev/hd* adlandırma şemasına benzer.
/dev/sr*	Optik sürücüler (CD/DVD okuyucular ve yazıcılar).

Ayrıca, kolaylık olması açısından, gerçek aygıt dosyalarına işaret eden /dev/cdrom, /dev/dvd ve /dev/floppy gibi sembolik bağlantılar da sıklıkla görülür.

Çıkarılabilir aygıtları otomatik olarak bağlamayan bir sistemde çalışıyorsak, çıkarılabilir aygıtın takıldığında nasıl adlandırıldığını belirlemek için aşağıdaki tekniği kullanabiliriz. İlk olarak, /var/log/messages veya /var/log/syslog dosyasının gerçek zamanlı görüntüsünü başlatın (bunun için süper kullanıcı ayrıcalıkları gerekebilir).

```
[me@linuxbox ~]$ sudo tail -f /var/log/syslog
```

Modern systemd tabanlı sistemlerde, systemd günlüğünü takip etmek için şu komutu kullanın:

```
[me@linuxbox ~]$ sudo journalctl -f
```

Listenin son birkaç satırı görüntülenecek ve ardından duraklayacaktır. Ardından, çıkarılabilir aygıtı takın. Bu örnekte, flash sürücümüzü söküp çıkaracağız ve ardından tekrar takacağız. Hemen hemen anında, çekirdek aygıtı fark edecek ve onu test edecektir.

```
25 Temmuz 13:15:07 ratel mtp-probe[21318]: bus 3, cihaz 8 kontrol
ediliyor: "/sys/devices/pci0000:00/0000:00:14.0/usb3/3-6"
25 Temmuz 13:15:07 ratel mtp-probe[21318]: bus: 3, device: 8 bir MTP
cihazı değildi
25 Temmuz 13:15:07 ratel mtp-probe[21321]: otobüs 3, aygıt 8 kontrol
ediliyor: "/sys/devices/pci0000:00/0000:00:14.0/usb3/3-6"
25 Temmuz 13:15:07 ratel mtp-probe[21321]: bus: 3, device: 8 bir MTP
aygıtı değildi
25 Temmuz 13:15:08 ratel çekirdeği: scsi 6:0:0:0: Doğrudan Erişim  Genel
UDisk             5.00 PQ: 0 ANSI: 2
25 Temmuz 13:15:08 ratel çekirdek: sd 6:0:0:0: Bağlı scsi genel sg3 türü
0
25 Temmuz 13:15:08 ratel çekirdek: sd 6:0:0:0: [sdc] 7864320 512
baytlık mantıksal bloklar: (4,03 GB/3,75 GiB)
25 Temmuz 13:15:08 ratel çekirdek: sd 6:0:0:0: [sdc] Yazma Koruması
kapalı 25 Temmuz 13:15:08 ratel çekirdek: sd 6:0:0:0: [sdc] Mod Algılama:
0b 00 00
08
25 Temmuz 13:15:08 ratel çekirdek: sd 6:0:0:0: [sdc] Önbellekleme modu
sayfası bulunamadı
25 Temmuz 13:15:08 ratel çekirdeği: sd 6:0:0:0: [sdc] Sürücü önbelleği
varsayımı: yazma
25 Temmuz 13:15:08 ratel çekirdek: sdc:
25 Temmuz 13:15:08 ratel çekirdek: sd 6:0:0:0: [sdc] Takılı SCSI
çıkarılabilir disk
```

Ekran tekrar durakladıktan sonra, `Ctrl-c tuşlarına` basarak komut istemini geri getirin. Çıktının ilginç kısımları, SCSI disk aygıt adı beklentimizle eşleşen `[sdc]` referanslarının tekrar tekrar geçmesidir. Bunu bilerek, şu iki satır özellikle aydınlatıcı hale gelir:

```
25 Temmuz 13:15:08 ratel çekirdek: sdc:
25 Temmuz 13:15:08 ratel çekirdek: sd 6:0:0:0: [sdc] Takılı SCSI
```

```
çıkarılabilir disk çıkarılabilir disk
```

İpucu: `tail -f /var/log/syslog` veya `journalctl -f` tekniğini kullanmak, sistemin ne yaptığını neredeyse gerçek zamanlı olarak izlemek için harika bir yoldur.

Bir aygıt adını belirlemenin başka bir yolu daha vardır (Linux'ta her zaman birden fazla yol vardır!). `lsblk` komutunu kullanabiliriz. Bu komut, takılı olup olmadıklarına bakılmaksızın sisteme bağlı tüm blok aygıtlarını listeler.

Flash sürücümüzü mount'tan kaldırıp çıkardığımızı varsayarsak, bağlı blok aygıtlarının listesine bakabiliriz:

```
[me@linuxbox ~]$ lsblk
NAME    MAJ:MIN RM    BOYUT RO TÜR MONTAJ
                      NOKTALARI
sda       8:0     0 111,8G  0 disk
├─sda1    8:1     0   976M  0 bölüm /boot/efi
│                             /
└─sda3    8:3     0  14,9G  0 part [SWAP]
sdb       8:16    0 931,5G  0 disk
├─sdb1    8:17    0 922,2G  0 part /home
└─sdb2    8:18    0   9,3G  0 part
sr0      11:0     1  1024M  0 rom
```

Ardından, flash sürücüyü tekrar takıp `lsblk komutunu` tekrar çalıştıracağız:

```
[me@linuxbox ~]$ lsblk
NAME    MAJ:MIN RM    BOYUT RO TÜR MONTAJ
                      NOKTALARI
sda       8:0     0 111,8G  0 disk
├─sda1    8:1     0   976M  0 bölüm /boot/efi
│                             /
└─sda3    8:3     0  14,9G  0 bölüm [SWAP]
sdb       8:16    0 931,5G  0 disk
├─sdb1    8:17    0 922,2G  0 part /home
└─sdb2    8:18    0   9,3G  0 part
sdc       8:32    1   3,8G  0 disk
sr0      11:0     1  1024M  0 rom
```

Şimdi listeye yeni bir aygıt (sdc) eklendiğini görüyoruz. Aygıt adı, fiziksel olarak bilgisayara bağlı kaldığı ve bilgisayar

yeniden başlatılmadı.

Cihazımızın adını öğrendikten sonra, flash sürücüyü takabiliriz.

```
[me@linuxbox ~]$ sudo mount /dev/sdc /mnt/flash
[me@linuxbox ~]$ df
Dosya                  1K      Kullanılan Kullanılabilir
sistemi             bloklar      Kullanım Kullanılan Bağlandığı yer
tmpfs               1624184   45133696   48250332  49% /
/dev/sda2          98430196          0    8120908   0% /dev/shm
tmpfs               8120908          4       5116   1% /run/lock
/dev/sda1            997456       6220     991236    1  /boot/efi
/dev/sdb1         950690656  585574576  316749944  %65 /home
tmpfs               1624180         80    1624100    %  /run/user/120
tmpfs               1624180        192    1623988    %  /run/user/1000
/dev/sdc            3924444          4    3924440    1  /mnt/flash
```

Yeni Dosya Sistemleri Oluşturma

Tek bir FAT32 dosya sistemine sahip harici bir USB sabit diskimiz olduğunu ve sürücüyü iki bölüme ayırmak istediğimizi varsayalım: birincisi Linux'a özgü bir dosya sistemi (ext4), ikincisi ise Windows sisteminde kullanılmak üzere NTFS olarak biçimlendirilmiş. Bu işlem iki adımdan oluşur.

1. Yeni bir bölüm düzeni oluşturun.
1. Sürücüde yeni, boş dosya sistemleri oluşturun.

Uyarı! Aşağıdaki alıştırmada, harici bir sabit sürücüyü biçimlendireceğiz. Silineceği için, içinde önemli hiçbir şey bulunmayan bir sürücü kullanın! Tekrar **belirtmek gerekirse, metinde gösterilen değil, sisteminiz için doğru aygıt adını belirttiğinizden kesinlikle emin olun. Bu uyarıya uymamanız, yanlış sürücüyü biçimlendirmenize (yani silmenize) neden olabilir!**

`parted` ile Bölümleri Yönetme

`parted`, disk benzeri aygıtlarla (sabit disk sürücüleri ve flash sürücüler gibi) çok düşük bir düzeyde doğrudan etkileşim kurmamızı sağlayan bir dizi programdan (hem komut satırı hem de grafiksel) biridir. Sürücüyü takıp `lsblk komutunu` kullanarak adını öğreneceğiz:

```
[me@linuxbox ~]$ lsblk
NAME    MAJ:MIN  RM    SIZE RO  TÜR  MONTAJ NOKTALARI
sda       8:0     0 111,8G  0 disk
├─sda1    8:1     0   976M  0 parça /boot/efi
│                                   /
└─sda3    8:3     0  14,9G  0 part  [SWAP]
sdb       8:16    0 931,5G  0 disk
├─sdb1    8:17    0 922,2G  0 part  /home
└─sdb2    8:18    0   9,3G  0 part
sdd       8:32    0 232,9G  0 disk
└─sdd1    8:33    0 232,9G  0 part  /media/me/USB_DISK
sr0      11:0     1  1024M  0 rom
```

Gördüğümüz gibi, harici USB sabit disk takılı ve adı `/dev/sdd` ve `/dev/sdd1` adlı bir bölüm içeriyor.

`parted` ile, aygıt üzerindeki bölümleri düzenleyebilir, silebilir ve oluşturabiliriz. Sabit sürücümüzle çalışmak için, önce onu (gerekirse) ayırmalı ve ardından `parted` programını aşağıdaki gibi çalıştırmalıyız:

```
[me@linuxbox ~]$ sudo umount /dev/sdd1
[me@linuxbox ~]$ sudo parted /dev/sdd
```

Cihazı bölüm numarasıyla değil, cihazın tamamı olarak belirtmemiz gerektiğini unutmayın. Program başladıktan sonra, aşağıdaki komut istemi görünecektir:

```
GNU Parted 3.4
/dev/sdd
kullanılıyor
GNU Parted'e hoş geldiniz! Komutların listesini görmek için 'help' yazın.
(parted)
```

"help" yazdığınızda kullanılabilir komutların listesi görüntülenir.

```
(parted) yardım
  align-check TİP N                        N bölümünü TYPE(min|opt)
                                           hizalaması için kontrol
  yardım [KOMUT]                           et genel yardım veya
                                           KOMUT hakkında yardım
  mklabel,mktable LABEL-TYPE               yazdır
                                           yeni bir disk etiketi
                                           oluştur
```

Oluşturma

```
                                                    (bölüm tablosu) bölüm
  mkpart PART-TYPE [FS-TYPE] START END              oluştur
  name NUMBER NAME                                  ad bölüm NUMBER adıyla NAME
                                                    bölüm tablosunu,
  yazdır [aygıtlar|boş|liste,tümü|NUMARA]           kullanılabilir aygıtları,
                                                    boş alanı, bulunan tüm
                                                    bölümleri veya belirli bir
                                                    bölümü görüntüle
                                                    programdan çık
  çık
  kurtarma BAŞLANGIÇ BİTİŞ                          START ve END yakınındaki
                                                    kayıp bir bölümü kurtar
                                                    bölüm NUMBER'ın boyutunu
  resizepart NUMARA SON                             değiştir bölüm NUMBER'ı
  rm NUMARA                                         sil
  seç DEVICE disk_set                               düzenlenecek aygıtı seç
  FLAG DURUM                                        seçilen aygıtın FLAG'ını
                                                    değiştir
  disk_toggle [FLAG] set                            seçilen aygıtta FLAG
                                                    durumunu değiştir NUMBER
  NUMBER FLAG STATE                                 bölümündeki FLAG'ı
                                                    değiştir
  toggle [NUMBER [FLAG]]                            NUMBER bölümündeki FLAG
                                                    durumunu değiştir
  unit UNIT                                         varsayılan birimi UNIT
                                                    olarak ayarla
  sürüm                                             sürüm numarasını görüntüle
                                                    ve GNU Parted'in telif
                                                    hakkı bilgilerini göster
```

İlk olarak, mevcut bölüm düzenini incelemek istiyoruz. Bunu, aygıtın bölüm tablosunu yazdırmak için "print" komutunu girerek yapıyoruz.

```
(parted) print
Model: WDC WD25 00BEVT-00A0RT0 (scsi)
Disk /dev/sdd: 250GB
Sektör boyutu (mantıksal/fiziksel):
512B/512B Bölüm Tablosu: msdos
Disk Bayrakları:

Sayı Başlangıç     Bitiş   Boyut   Tür       Dosya sistemi
 1      1049 kB 250 GB 250 GB birincil fat32 Bayrakları    lba
```

Diskimizin tek bir FAT32 bölümünde 250 GB alan olduğunu görebiliriz.

Yeni bölümlerimizi oluşturmak için önce mevcut bölümü silmeliyiz. Bu, `rm` komutuyla kolayca yapılabilir.

```
(parted) rm 1
(parted) print
Model: WDC WD25 00BEVT-00A0RT0 (scsi)
Disk /dev/sdd: 250GB
Sektör boyutu (mantıksal/fiziksel): 512B/512B
Bölüm Tablosu: msdos
Disk Bayrakları:

Numara Başlangıç Bitiş Boyut Tür Dosya sistemi Bayraklar
```

Kaldırmak istediğimiz bölüm numarasını belirledikten sonra, bölüm tablosuna tekrar bakarak bölümün kaldırıldığını kontrol ederiz.

Ardından, yeni bölümlerimizi oluşturacağız. Bunu `mkpart` komutuyla yapıyoruz.

```
(parted) mkpart
Bölüm türü? birincil/genişletilmiş? birincil
Dosya sistemi türü? [ext2]? ext4
Başlangıç? 1
Bitiş? 120000
```

Bu diskte (MBR, ana önyükleme kaydı tarzı bölüm tablosuna sahip) bir bölüm oluştururken, birincil veya genişletilmiş bölüm belirleriz ve ardından varsayılan bir megabaytlık artışlarla bölümün başlangıcını ve sonunu belirtiriz. 120.000 MB'lık bir son değer kullanmak, sürücüdeki kullanılabilir alanın yaklaşık yarısını tüketir.

Ardından, aynı tekniği kullanarak ikinci bölümü oluşturacağız.

```
(parted) mkpart
Bölüm türü? birincil/genişletilmiş? birincil
Dosya sistemi türü? [ext2]? ntfs
Başlangıç? 120001
Bitiş? 240000
```

Son olarak, sonuçlarımızı görmek için bölüm tablosuna bakıyoruz.

```
(parted) yazdır
Model: WDC WD25 00BEVT-00A0RT0 (scsi)
Disk /dev/sdd: 250GB
Sektör boyutu (mantıksal/fiziksel): 512B/512B
Bölüm Tablosu: msdos
Disk Bayrakları:
Sayı    Başlangıç  Bitiş   Boyut   Tür       Dosya sistem   Bayraklar
1       1049 kB  120 GB  120 GB  birincil  ext4           lba
2       120 GB   240 GB  120 GB  birincil  ntfs           lba
```

Bölümleme işlemi tamamlandığına göre, `parted` programından çıkabiliriz.

```
(parted) çık
```

`lsblk komutunu` tekrar çalıştırırsak, sürücümüzü ve iki bölümünü görebiliriz.

```
AD      MAJ:MIN  RM    SIZE  RO  TÜR  MONTAJ NOKTALARI
sda       8:0     0  111,8G   0 disk
├─sda1    8:1     0    976M   0 parça /boot/efi
│                                    /
└─sda3    8:3     0   14,9G   0 part  [SWAP]
sdb       8:16    0  931,5G   0 disk
├─sdb1    8:17    0  922,2G   0 part  /home
└─sdb2    8:18    0    9,3G   0 part
sdd       8:48    0  232,9G   0 disk
├─sdd1    8:49    0  111,8G   0 parça
└─sdd2    8:50    0  111,8G   0 parça
sr0      11:0     1   1024M   0 rom
```

`mkfs` ile Yeni Dosya Sistemi Oluşturma

Bölüm düzenlememiz tamamlandıktan sonra, sürücümüzde yeni dosya sistemlerini oluşturma (yani biçimlendirme) zamanı geldi. Bunu yapmak için, çeşitli formatlarda dosya sistemleri oluşturabilen `mkfs` (kısaltması "make filesystem") komutunu kullanacağız. Sürücüde ext4 dosya sistemini oluşturmak için, ext4 dosya sistemi türünü belirtmek için `-t` seçeneğini kullanarak `ext4` dosya sistemi türünü, ardından açıklayıcı bir birim etiketini ve biçimlendirmek istediğimiz bölümü içeren aygıtın adını belirtiriz.

```
[me@linuxbox ~]$ sudo mkfs -t ext4 -L EXT4_Disk /dev/sdd1 mke2fs
1.46.5 (30-Aralık-2021)
29296640 4k blok ve 7331840 inode ile dosya sistemi oluşturuluyor
Dosya sistemi UUID: 3365d135-d8d9-4b93-a57a-ec3567c8548a
Bloklarda depolanan süper blok yedeklemeleri:
    32768, 98304, 163840, 229376, 294912, 819200, 884736, 1605632,
    2654208, 4096000, 7962624, 11239424, 20480000, 23887872

Grup tabloları ayrılıyor:
                           tamamla
ndı İnode tabloları yazılıyor:
                           tamam
Günlük oluşturuluyor (131072 blok): tamam
Süper bloklar ve dosya sistemi hesap bilgileri yazılıyor:         tamam
```

Şimdi NTFS bölümünü oluşturacağız.

```
[me@linuxbox ~]$ sudo mkfs -t ntfs --quick -L NTFS_Disk /dev/sdd2 Küme
boyutu otomatik olarak 4096 bayta ayarlandı.
NTFS birim yapıları oluşturuluyor.
mkntfs başarıyla tamamlandı. İyi günler.
```

Yine bir dosya sistemi türü, açıklayıcı bir birim etiketi ve bir aygıt belirtiyoruz. Kötü blok kontrolünü atlamak için `--quick` seçeneğini ekliyoruz, çünkü bu işlemin gerçekleştirilmesi *uzun* zaman alıyor.

Farklı dosya sistemi türlerinin farklı seçenekleri desteklediğini unutmayın. Ayrıntılar için `mkfs` man sayfasına bakın.

Çalışmamız tamamlandığından, sürücüyü çıkaralım ve tekrar takalım, böylece sistem sürücüyü otomatik olarak bağlasın. `lsblk komutunu` bir kez daha çalıştırarak sonuçları görelim.

```
[me@linuxbox ~]$ lsblk
NAME    MAJ:MIN RM    BOYUT RO TÜR MONTAJ
                      NOKTALARI
sda       8:0    0 111,8G  0 disk
├─sda1    8:1    0   976M  0 bölüm /boot/efi
│                              /
└─sda3    8:3    0  14,9G  0 bölüm [SWAP]
sdb       8:16   0 931,5G  0 disk
├─sdb1    8:17   0 922,2G  0 part /home
└─sdb2    8:18   0   9,3G  0 part
sdd       8:48   0 232,9G  0 disk
├─sdd1    8:49   0 111,8G  0 part /media/me/EXT4_Disk
```

```
└─sdd2    8:50    0 111,8G 0 bölüm /media/me/NTFS_Disk
sr0      11:0     1 1024M 0 rom
```

Görüldüğü gibi, birim etiketleri, çıkarılabilir depolama aygıtlarının otomatik olarak bağlandığı `/me-dia` dizininde bağlama noktası adlarını oluşturmak için kullanılır.

Dosya Sistemlerini Test Etme ve Onarma

Daha önce `/etc/fstab` dosyası hakkında yaptığımız tartışmada, her satırın sonunda bazı gizemli rakamlar gördük. Sistem her önyükleme yaptığında, dosya sistemlerini bağlamadan önce rutin olarak bütünlüklerini kontrol eder. Bu, `fsck` programı ("filesys-tem check"in kısaltması) tarafından yapılır. Her `fstab` girişindeki son sayı, aygıtların kontrol edileceği sırayı belirtir. Önceki örneğimizde, kök dosya sisteminin ilk olarak kontrol edildiğini, ardından `home` ve `boot` dosya sistemlerinin kontrol edildiğini görüyoruz. Son rakamı sıfır olan aygıtlar rutin olarak kontrol edilmez.

Dosya sistemlerinin bütünlüğünü kontrol etmenin yanı sıra, `fsck`, hasarın derecesine bağlı olarak, bozuk dosya sistemlerini de çeşitli derecelerde başarıyla onarabilir. Unix benzeri dosya sistemlerinde, kurtarılan dosya bölümleri, her dosya sisteminin kökünde bulunan `lost+found` dizinine yerleştirilir.

`EXT4_Disk` bölümümüzü kontrol etmek için (önce bağlantısı kesilmelidir) aşağıdakileri yapabiliriz:

```
[me@linuxbox ~]$ sudo umount /dev/sdd1
[me@linuxbox ~]$ sudo fsck /dev/sdd1
util-linux 2.37.2'den fsck
e2fsck 1.46.5 (30 Aralık 2021)
EXT4_Disk: temiz, 11/7331840 dosya, 606693/29296640 blok
```

Günümüzde, disk sürücüsünün arızalanması gibi bir donanım sorunu olmadığı sürece dosya sistemi bozulması oldukça nadirdir. Çoğu sistemde, önyükleme sırasında tespit edilen dosya sistemi bozulması, sistemin durmasına ve devam etmeden önce `fsck` komutunu çalıştırmanıza neden olur.

`fsck` nedir?

Unix kültüründe, *fsck* kelimesi genellikle üç harfini paylaştığı popüler bir kelimenin yerine kullanılır. Bu özellikle uygun bir durumdur, çünkü `fsck'yi` çalıştırmak zorunda kaldığınız bir durumda muhtemelen yukarıda bahsedilen kelimeyi söyleyeceksiniz.

Verileri Doğrudan Aygıtlara ve Aygıtlardan Taşıma

Bilgisayarlarımızdaki verileri genellikle dosya halinde düzenlenmiş olarak düşünürüz, ancak bu verileri "ham" olarak da düşünebiliriz. Örneğin, bir disk sürücüsüne baktığımızda, işletim sisteminin dizinler ve dosyalar olarak gördüğü çok sayıda veri "bloğundan" oluştuğunu görürüz. Ancak, bir disk sürücüsünü sadece büyük bir veri bloğu koleksiyonu olarak ele alabilirsek, cihazları klonlamak gibi yararlı görevleri gerçekleştirebiliriz.

`dd` programı bu görevi yerine getirir. Veri bloklarını bir yerden başka bir yere kopyalar. (Tarihsel nedenlerden dolayı) benzersiz bir sözdizimi kullanır ve genellikle şu şekilde kullanılır:

```
dd if=giriş_dosyası of=çıkış_dosyası [bs=blok_boyutu [count=bloklar]]
```

Uyarı! `dd` komutu çok güçlüdür. Adı "veri tanımı"ndan gelse de, kullanıcılar genellikle `if` veya `of` belirtimini yanlış yazdıkları için bazen "disk yok et" olarak da adlandırılır. **`ENTER` tuşuna basmadan önce giriş ve çıkış belirtimlerinizi daima iki kez kontrol edin!**

Aynı boyutta iki USB flash sürücümüz olduğunu ve ilk sürücüyü ikinci sürücüye tam olarak kopyalamak istediğimizi varsayalım. Her iki sürücüyü de bilgisayara bağladığımızda ve bunlar sırasıyla `/dev/sdb` ve `/dev/sdc` aygıtlarına atandığında, ilk sürücüdeki her şeyi ikinci sürücüye aşağıdaki komutla kopyalayabiliriz:

```
[me@linuxbox ~]$ sudo dd if=/dev/sdb of=/dev/sdc
```

Alternatif olarak, bilgisayara yalnızca ilk aygıt bağlıysa, içeriğini daha sonra geri yüklemek veya kopyalamak için sıradan bir dosyaya kopyalayabiliriz.

```
[me@linuxbox ~]$ sudo dd if=/dev/sdb of=flash_drive.img
```

CD-ROM Görüntüleri Oluşturma

Kaydedilebilir bir CD-ROM (CD-R veya CD-RW) yazmak iki adımdan oluşur.

1. CD-ROM'un tam dosya sistemi görüntüsü olan bir *ISO görüntü dosyası* oluşturmak
2. Görüntü dosyasını CD-ROM ortamına yazma

CD-ROM'un Görüntü Kopyasını Oluşturma

Mevcut bir CD-ROM'un ISO görüntüsünü oluşturmak istiyorsak, `dd` komutunu kullanarak CD-ROM'daki tüm veri bloklarını okuyup bunları yerel bir dosyaya kopyalayabiliriz. Diyelim ki bir Ubuntu CD'miz var ve daha sonra daha fazla kopya oluşturmak için kullanabileceğimiz bir ISO dosyası oluşturmak istiyoruz. CD'yi takıp aygıt adını belirledikten sonra (`/dev/cdrom olduğunu` varsayalım), ISO dosyasını şu şekilde oluşturabiliriz:

```
dd if=/dev/cdrom of=ubuntu.iso
```

Bu teknik veri DVD'leri için de geçerlidir, ancak ses CD'leri için geçerli değildir, çünkü ses CD'leri depolama için dosya sistemi kullanmaz. Ses CD'leri için `cdrdao` komutuna bakın.

Bir Dosya Koleksiyonundan Görüntü Oluşturma

Bir dizinin içeriğini içeren bir ISO görüntü dosyası oluşturmak için `genisoimage` programını kullanırız. Bunu yapmak için, önce görüntüye dahil etmek istediğimiz tüm dosyaları içeren bir dizin oluştururuz ve ardından `genisoimage` komutunu çalıştırarak görüntü dosyasını oluştururuz. Örneğin, `~/cd-rom-files` adlı bir dizin oluşturup CD-ROM'umuz için dosyaları bu dizine doldurduysak, aşağıdaki komutla `cd-rom.iso` adlı bir görüntü dosyası oluşturabiliriz:

```
genisoimage -o cd-rom.iso -R -J ~/cd-rom-files
```

`-R` seçeneği, uzun dosya adlarının ve POSIX tarzı dosya izinlerinin kullanımına olanak tanıyan *Rock Ridge uzantıları* için meta veriler ekler. Benzer şekilde, `-J` seçeneği Windows için uzun dosya adlarına izin veren *Joliet uzantılarını* etkinleştirir.

Başka Bir Adla Bir Program...

CD-ROM ve DVD gibi optik ortamları oluşturmak ve yazmak için çevrimiçi öğreticilere bakarsanız, `mkisofs` ve `cdrecord` adlı iki programa sık sık rastlayacaksınız. Bu programlar, Jörg Schilling tarafından yazılan `cdr-tools` adlı popüler bir paketin parçasıydı. 2006 yazında, Schilling, `cdrtools` paketinin bir kısmının lisansını değiştirdi ve bu, Linux topluluğundaki birçok kişinin görüşüne göre, GNU GPL ile lisans uyumsuzluğu yarattı. Sonuç olarak, cdrtools projesinin bir *çatalı* başlatıldı ve bu çatal, artık sırasıyla `wodim` ve `genisoimage` adlı `cdrecord` ve `mkisofs` için yer değiştirme programlarını içeriyor.

CD-ROM Görüntüleri Yazma

ISO görüntü dosyasını elde ettikten sonra, bunu optik ortama yazabiliriz. Aşağıda ele alacağımız komutların çoğu, hem yazılabilir CD-ROM hem de DVD ortamlarına uygulanabilir.

ISO Görüntüsünü Doğrudan Bağlama

ISO görüntüsünü hala sabit diskimizdeyken bağlamak ve sanki zaten optik medyada varmış gibi işlemek için kullanabileceğimiz bir püf noktası vardır. `Bağlamak` için "-o loop" seçeneğini (gerekli "-t iso9660" dosya sistemi türüyle birlikte) ekleyerek, görüntü dosyasını sanki bir aygıtmış gibi bağlayabilir ve dosya sistemi ağacına ekleyebiliriz.

```
mkdir /mnt/iso_image
mount -t iso9660 -o loop image.iso /mnt/iso_image
```

Yukarıdaki örnekte, `/mnt/iso_image` adlı bir bağlama noktası oluşturduk ve ardından `image.iso` görüntü dosyasını bu bağlama noktasına bağladık. Görüntü bağlandıktan sonra, gerçek bir CD-ROM veya DVD gibi kullanılabilir. *Artık gerekmediğinde görüntüyü bağlamayı unutmayın*.

Yeniden Yazılabilir CD-ROM'u Boşaltma

Yeniden yazılabilir CD-RW ortamları, yeniden kullanılmadan önce silinmeli veya *boşaltılmalıdır*. Bunu yapmak için, CD yazıcının aygıt adını ve gerçekleştirilecek boşaltma türünü belirterek `wodim` komutunu kullanabiliriz. `wodim` programı birkaç tür sunar. En minimal (ve en hızlı) olanı "fast" türüdür.

```
wodim dev=/dev/cdrw blank=fast
```

Görüntü Yazma

Bir görüntü yazmak için yine `wodim komutunu` kullanırız, optik medya yazıcı aygıtının adını ve görüntü dosyasının adını belirtiriz.

```
wodim dev=/dev/cdrw image.iso
```

Aygıt adı ve görüntü dosyasına ek olarak, `wodim` çok sayıda seçeneği destekler. Yaygın olarak kullanılan iki seçenek, ayrıntılı çıktı için "-v" ve diski *tek seferde* yazma modunda yazan "-dao"dur. Bu mod, ticari çoğaltma için bir disk hazırlıyorsak kullanılmalıdır. `wodim` için varsayılan mod, müzik parçalarını kaydetmek için yararlı olan *tek seferde iz mod*udur.

Verileri Doğrulama

ISO dosyaları konusuna değinmişken, indirdiğimiz dosyaların bütünlüğünü nasıl doğrulayacağımıza da bir göz atalım. Genellikle, yeni bir Linux kurulum görüntüsü gibi büyük bir dosya indirdiğimizde, indirdiğimiz görüntü dosyasının eksiksiz ve bozuk olmadığından emin olmak isteriz. Bunun için kullanabileceğimiz araçlardan biri, `md5sum` adlı eski bir programın modern bir versiyonu olan `sha256sum'dur`.

Bunu yapmanın iki yaygın yolu vardır. Bunları deneyelim. Linux Mint 22'nin kurulum görüntüsünü indirmek istediğimizi varsayalım. Linux Mint web sitesine gidip `linuxmint-22-cinnamon-64bit.iso` dosyasını indiriyoruz. Oradayken `sha256sum.txt` adlı bir *sağlama toplamı dosyasını* da indiriyoruz. İçeriğine baktığımızda şunu görüyoruz:

```
[me@linuxbox ~]$ cat sha256sum.txt
7a04b54830004e945c1eda6ed6ec8c57ff4b249de4b331bd021a849694f29b8f *lin
uxmint-22-cinnamon-64bit.iso
```

```
78a2438346cfe69a1779b0ac3fc05499f8dc7202959d597dd724a07475bc6930 *lin
uxmint-22-mate-64bit.iso
55e917b99206187564029476f421b98f5a8a0b6e54c49ff6a4cb39dcfeb4bd80 *lin
uxmint-22-xfce-64bit.iso
```

Uzun onaltılık sayı dizileri içeren üç satırlık veri ve indirdiğimiz ISO dosyası dahil olmak üzere indirilebilir dosyaların adları. Onaltılık sayı dizileri, ISO dosyalarının kesin matematiksel temsilini gösteren *sağlama toplamlarıdır*. Bu sayılar, indirilen ISO dosyası orijinal dosyadan tek bir bit bile değiştirilirse sağlama toplamı önemli ölçüde farklı olacağı için özeldir. Sağlama toplamlarının eşleşip eşleşmediğini görmek için ISO dosyasını test edeceğiz.

```
[me@linuxbox ~]$ sha256sum -b linuxmint-22-cinnamon-64bit.iso
7a04b54830004e945c1eda6ed6ec8c57ff4b249de4b331bd021a849694f29b8f *lin
uxmint-22-cinnamon-64bit.iso
```

`-b` seçeneği, `sha256sum`'a metin yerine ikili bir dosya incelediğimizi bildirir. Gördüğümüz gibi, sağlama toplamı `sha256sum.txt` dosyasında gördüğümüzle eşleşiyor. Ancak sağlama toplamını bu şekilde incelemek biraz zahmetli. Dosyayı kontrol etmenin başka bir yolu da, `sha256sum`'un `sha256sum.txt` dosyasını işleyerek, bu dosyanın sağlama toplamını listedeki dosyalardan oluşturduğu sağlama toplamlarıyla karşılaştırmasını sağlamaktır. Bunu, `sha256sum` programını şu şekilde çalıştırarak yapabiliriz:

```
[me@linuxbox ~]$ sha256sum -c --ignore-missing sha256sum.txt
linuxmint-22-cinnamon-64bit.iso: OK
```

`-c` seçeneği ("check"in kısaltması) bu modu çağırırken, `--ignore-missing` seçeneği `sha256sum`'a indirmediğimiz `lin-uxmint-22-mate-64bit.iso` ve `linuxmint-22-xfce-64bit.iso` dosyalarının mevcut olmadığını bildirmemesini söyler.

Özet

Bu bölümde temel depolama yönetimi görevlerini inceledik. Elbette daha pek çok görev vardır. Linux, çok çeşitli depolama aygıtlarını ve dosya sistemi şemalarını destekler. Ayrıca diğer sistemlerle birlikte çalışabilirlik için birçok özellik sunar.

Daha Fazla Okuma

Ele aldığımız komutların man sayfalarını inceleyin. Bazıları çok sayıda seçenek ve işlemi destekler. Ayrıca, Linux sistemine sabit disk eklemek (çok sayıda vardır) ve optik ortamlarla çalışmak için çevrimiçi öğreticiler arayın.

- Arch Wiki'de bölümleme hakkında iyi bir makale var: https://wiki.archlinux.org/title/Partitioning
- GNU Parted Kullanıcı Kılavuzunun tamamı şu adreste bulunabilir: https://www.gnu.org/software/ parted/manual/parted.html
- Kontrol toplamları hakkında bir Wikipedia makalesi: https://en.wikipedia.org/wiki/Checksum
- Herkesin en sevdiği SCSI hakkında Wikipedia girişi: https://en.wikipedia.org/ wiki/SCSI

16 – Ağ

Ağ iletişimi söz konusu olduğunda, Linux ile yapılamayacak hiçbir şey yoktur. Linux, güvenlik duvarları, yönlendiriciler, ad sunucuları, ağa bağlı depolama (NAS) kutuları ve daha pek çok şey dahil olmak üzere her türlü ağ sistemi ve cihazını oluşturmak için kullanılır.

Ağ iletişimi konusu çok geniş olduğu gibi, onu yapılandırmak ve kontrol etmek için kullanılabilecek komutların sayısı da çok fazladır. Biz sadece en sık kullanılanlardan birkaçına odaklanacağız. İnceleme için seçilen komutlar arasında ağları izlemek için kullanılanlar ve dosya aktarımı için kullanılanlar bulunmaktadır. Ayrıca, uzaktan oturum açmak için kullanılan `ssh` programını da inceleyeceğiz. Bu bölümde aşağıdaki komutlar ele alınacaktır:

- `ping` – Ağ ana bilgisayarlarına ICMP ECHO_REQUEST gönderir
- `traceroute` – Bir ağ ana bilgisayarına giden paketlerin izini yazdırır
- `ip` – Yönlendirme, aygıtlar, ilke yönlendirme ve tünelleri gösterir/değiştirir
- `netstat` – Ağ bağlantılarını, yönlendirme tablolarını, arayüz istatistiklerini, maskeleme bağlantılarını ve çoklu yayın üyeliklerini yazdırır
- `ftp` – İnternet dosya aktarım programı
- `curl - Bir URL'yi aktar`
- `wget` – Etkileşimli olmayan ağ indiricisi
- `ssh` – OpenSSH SSH istemcisi (uzaktan oturum açma programı)

Ağ konusunda biraz bilgi sahibi olduğunuzu varsayacağız. İnternet çağında, bilgisayar kullanan herkesin ağ kavramları hakkında temel bilgilere sahip olması gerekir. Bu bölümü tam olarak anlayabilmek için aşağıdaki terimlere aşina olmalıyız:

- İnternet protokolü (IP) adresi
- Ana bilgisayar ve etki alanı adı
- Tekdüzen kaynak tanımlayıcı (URI)

Bu terimlerle ilgili bazı yararlı makaleler için lütfen aşağıdaki "Daha Fazla Okuma" bölümüne bakın.

Not: Ele alacağımız bazı komutlar (dağıtımınıza bağlı olarak) dağıtımınızın depolarından ek paketlerin yüklenmesini gerektirebilir ve bazıları çalıştırmak için süper kullanıcı ayrıcalıkları gerektirebilir.

Ağı İnceleme ve İzleme

Sistem yöneticisi olmasanız bile, bir ağın performansını ve çalışmasını incelemek genellikle yararlıdır.

ping

En temel ağ komutu `ping'dir. Ping` komutu, ICMP ECHO_REQUEST adı verilen özel bir ağ paketini belirtilen bir ana bilgisayara gönderir. Bu paketi alan çoğu ağ aygıtı buna yanıt verir ve ağ bağlantısının doğrulanmasını sağlar.

Not: Çoğu ağ cihazını (Linux ana bilgisayarlar dahil) bu paketleri yok sayacak şekilde yapılandırmak mümkündür. Bu genellikle güvenlik nedenleriyle, potansiyel bir saldırganın ana bilgisayarı kısmen gizlemek için yapılır. Güvenlik duvarlarının ICMP trafiğini engelleyecek şekilde yapılandırılması da yaygındır.

Örneğin, `linuxcommand.org` (en sevdiğimiz sitelerden biri ;-) adresine ulaşıp ulaşamadığımızı görmek için `ping komutunu` şu şekilde kullanabiliriz:

```
[me@linuxbox ~]$ ping linuxcommand.org
```

Başlatıldığında, `ping` kesintiye uğrayana kadar belirli aralıklarla (varsayılan aralık bir saniyedir) paket göndermeye devam eder.

```
[me@linuxbox ~]$ ping linuxcommand.org
PING linuxcommand.org (66.35.250.210) 56(84) bayt veri.
vhost.sourceforge.net (66.35.250.210) 'dan 64 bayt: icmp_seq=1 ttl=43
süre=107 ms
vhost.sourceforge.net (66.35.250.210) adresinden 64 bayt: icmp_seq=2
ttl=43 süre=108 ms
vhost.sourceforge.net (66.35.250.210) adresinden 64 bayt: icmp_seq=3
ttl=43 süre=106 ms
vhost.sourceforge.net (66.35.250.210) adresinden 64 bayt: icmp_seq=4
ttl=43 süre=106 ms
```

```
vhost.sourceforge.net (66.35.250.210) adresinden 64 bayt: icmp_seq=5
ttl=43 süre=105 ms
vhost.sourceforge.net (66.35.250.210) adresinden 64 bayt: icmp_seq=6
ttl=43 süre=107 ms

--- linuxcommand.org ping istatistikleri ---
6 paket gönderildi, 6 paket alındı, %0 paket kaybı, süre 6010 ms
rtt min/ort/maks/mdev = 105,647/107,052/108,118/0,824 ms
```

`Ctrl-c` tuşuna basılarak kesintiye uğradıktan sonra (bu durumda altıncı paketten sonra), `ping` performans istatistiklerini yazdırır. Düzgün çalışan bir ağda paket kaybı yüzde 0 olur. Başarılı bir "ping" işlemi, ağın bileşenlerinin (arayüz kartları, kablolama, yönlendirme ve ağ geçitleri) genel olarak iyi çalıştığını gösterir.

Dikkat edilmesi gereken bir diğer nokta, ping'e yanıt veren makinenin, bizim istediğimiz makineden farklı olabileceğidir. Bunun nedeni, bir makinenin (bizim durumumuzda `vhost.source-forge.net`) farklı isimlere sahip birçok siteyi barındırabilmesidir.

traceroute

`Traceroute` programı (bazı sistemler bunun yerine benzer `tracepath` programını kullanır), yerel sistemden belirli bir ana bilgisayara ulaşmak için ağ trafiğinin geçtiği tüm "atlamaları" listeler. Örneğin, `slashdot.org'a` ulaşmak için kullanılan yolu görmek için şunu yaparız:

```
[me@linuxbox ~]$ traceroute slashdot.org
```

Çıktı şöyle görünür:

```
slashdot.org'a (216.34.181.45) traceroute, maksimum 30 atlama, 40
bayt
paket
 1  ipcop.localdomain (192.168.1.1) 1,066 ms 1,366 ms 1,720 ms
 2  * * *
 3  ge-4-13-ur01.rockville.md.bad.comcast.net (68.87.130.9)      14,622
ms  14,885 ms 15,169 ms
 4  po-30-ur02.rockville.md.bad.comcast.net (68.87.129.154)      17,634
ms  17,626 ms 17,899 ms
 5  po-60-ur03.rockville.md.bad.comcast.net (68.87.129.158)      15,992
ms  15,983 ms 16,256 ms
 6  po-30-ar01.howardcounty.md.bad.comcast.net (68.87.136.5)      22,835
ms  14,233 ms 14,405 ms
 7  po-10-ar02.whitemarsh.md.bad.comcast.net (68.87.129.34)      16.154
```

```
ms 13,600 ms 18,867 ms
 8 te-0-3-0-1-cr01.philadelphia.pa.ibone.comcast.net (68.86.90.77)
21,951 ms 21,073 ms 21,557 ms
 9 pos-0-8-0-0-cr01.newyork.ny.ibone.comcast.net (68.86.85.10)
22,917 ms 21,884 ms 22,126 ms
10 204.70.144.1 (204.70.144.1) 43,110 ms 21,248 ms 21,264 ms
11 cr1-pos-0-7-3-1.newyork.savvis.net (204.70.195.93) 21,857 ms
cr2-pos-0-0-3-1.newyork.savvis.net (204.70.204.238) 19,556 ms cr1-
pos-0-7-3-1.newyork.savvis.net (204.70.195.93) 19,634 ms
12 cr2-pos-0-7-3-0.chicago.savvis.net (204.70.192.109) 41,586 ms
42,843 ms cr2-tengig-0-0-2-0.chicago.savvis.net (204.70.196.242)
43,115 ms
13  hr2-tengigabitethernet-12-1.elkgrovech3.savvis.net
(204.70.195.122) 44,215 ms 41,833 ms 45,658 ms
14  csr1-ve241.elkgrovech3.savvis.net (216.64.194.42) 46,840 ms
43,372 ms 47,041 ms
15 64.27.160.194 (64.27.160.194) 56,137 ms 55,887 ms 52,810 ms
16 slashdot.org (216.34.181.45) 42,727 ms 42,016 ms 41,437 ms
```

Çıktıda, test sistemimizden `slashdot.org'a` bağlanmak için 16 yönlendiriciden geçilmesi gerektiğini görebiliriz. Tanımlayıcı bilgi sağlayan yönlendiriciler için, ana bilgisayar adlarını, IP adreslerini ve yerel sistemden yönlendiriciye gidiş-dönüş süresinin üç örneğini içeren performans verilerini görürüz. Tanımlayıcı bilgi sağlamayan yönlendiriciler için (yönlendirici yapılandırması, ağ tıkanıklığı, güvenlik duvarları vb. nedenleriyle), 2 numaralı atlama satırında olduğu gibi yıldız işaretleri görürüz. Yönlendirme bilgilerinin engellendiği durumlarda, bazen `traceroute` komutuna `-T` veya `-I` seçeneğini ekleyerek bu sorunu aşabiliriz.

ip

`ip` programı, modern Linux çekirdeklerinde bulunan tüm ağ özelliklerini kullanan çok amaçlı bir ağ yapılandırma aracıdır. Eskiden kullanılan ve artık kullanımdan kaldırılan `ifconfig` programının yerini almıştır. ip programı, çeşitli ağ ayarlarını ve istatistiklerini incelemek için kullanılır. Programın birçok seçeneğini kullanarak, ağ kurulumumuzdaki çeşitli özellikleri inceleyebiliriz. `ip` ile, bir sistemin ağ arayüzlerini ve yönlendirme tablosunu inceleyebiliriz. İlk olarak, arayüzler:

```
[me@linuxbox ~]$ ip address show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group
default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
```

```
    inet 127.0.0.1/8 kapsam ana bilgisayar lo
       valid_lft sonsuza kadar preferred_lft
    sonsuza kadar inet6 ::1/128 kapsam ana
    bilgisayar noprefixroute
       valid_lft sonsuza kadar preferred_lft sonsuza kadar
2: enp1s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq durumu
UP grup varsayılan qlen 1000
    link/ether 00:26:6c:26:67:bf brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.223/24 brd 192.168.50.255 kapsam global dinamik
noprefixroute enp1s0
       valid_lft 82366sec preferred_lft 82366sec
    inet6 fe80::226:6cff:fe26:67bf/64 kapsam bağlantı noprefixroute
       valid_lft sonsuza kadar preferred_lft sonsuza kadar
3: wlp2s0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc mq
durum DOWN grup varsayılan qlen 1000
    link/ether 06:8a:59:f4:f6:d3 brd ff:ff:ff:ff:ff:ff permaddr
24:ec:99:46:aa:1e
```

Yukarıdaki örnekte, test sistemimizin üç ağ arabirimi olduğunu görüyoruz. İlki, `lo` olarak adlandırılan, sistemin "kendisiyle iletişim kurmak" için kullandığı sanal bir arabirim olan *geri döngü arabirimidir*, ikincisi, `enp1s0` olarak adlandırılan, bir Ethernet arabirimidir (`en` = Ethernet) ve üçüncüsü, `wlp2s0` olarak adlandırılan, bir kablosuz arabirimdir (`wl` = kablosuz).

Gündelik ağ tanılama yaparken dikkat edilmesi gereken önemli noktalar, arabirimin ilk satırında etkin olduğunu belirten `state UP` ifadesinin bulunması ve üçüncü satırdaki `inet` alanında geçerli bir IP adresinin bulunmasıdır. Dinamik Ana Bilgisayar Yapılandırma Protokolü (DHCP) kullanan sistemlerde, bu alandaki geçerli bir IP adresi DHCP'nin çalıştığını doğrular.

Sırada, yönlendirme tablosu var:

```
[me@linuxbox ~]$ ip route show
varsayılan yol 192.168.1.1 dev enp1s0 proto dhcp src 192.168.50.223
metrik 100
169.254.0.0/16 dev enp1s0 scope link metric 1000
192.168.1.0/24 dev enp1s0 proto kernel scope link src 192.168.1.223 metric
100
```

Bu basit örnekte, bir güvenlik duvarı/yönlendiricinin arkasındaki yerel alan ağındaki (LAN) bir istemci makine için tipik bir yönlendirme tablosu görüyoruz. Sıfırla biten IP adresleri, tek tek ana bilgisayarlar yerine ağları ifade eder, bu nedenle bu hedef, LAN'daki herhangi bir ana bilgisayara doğrudan ulaşılabileceği anlamına gelir. İki ağ listelenmiştir: `169.254.0.0` ve `192.168.1.0`. `192.168.1.0` ağı bizim LAN ağımızdır, peki `169.254.0.0` nedir? Bu,

Otomatik Özel IP Adresleme (APIPA) adlı bir ağ hilesi. DHCP sunucusu kullanılamadığında otomatik olarak bir IP adresi atamak için kullanılır.

İlk satır hedef `varsayılanını` içerir. Bu, tabloda listelenmeyen bir ağa giden tüm trafiği bu adrese gönderin anlamına gelir. Örneğimizde, varsayılan ağ geçidinin `192.168.1.1` adresine sahip bir yönlendirici olarak tanımlandığını görüyoruz (ev yönlendiricileri için tipik bir adres). Bu yönlendirici, hedef trafiği ne yapacağını muhtemelen bilir.

`ip` komutu, birçok seçenek ve komut içeren karmaşık bir programdır. Komut sözdizimi şunlardan oluşur:

```
ip [-seçenekler] nesne [komut]
```

Önceki örneklerde, `show` komutuyla `address` ve `route` nesnelerini kullandık. Kolaylık olması açısından, `ip` nesne adlarının ve komut adlarının belirsiz olmayan herhangi bir önekle (genellikle tek bir karakter kadar kısa) kısaltılmasına izin verir ve varsayılan komut `show` olduğu için, komutları `ip a` ve `ip r` olarak kısaltabilir ve aynı sonuçları elde edebiliriz.

Ağ Üzerinden Dosya Aktarımı

Ağ üzerinden dosya aktaramazsak ağın ne faydası var? Ağ üzerinden veri aktaran birçok program vardır. Şimdi bunlardan ikisini ele alacağız, diğerlerini ise sonraki bölümlerde ele alacağız.

`ftp`

Gerçek "klasik" programlardan biri olan `ftp`, adını kullandığı protokol olan *Dosya Aktarım Protokolü'*nden alır. FTP, bir zamanlar İnternet üzerinden dosya indirmek için en yaygın kullanılan yöntemdi. Bazı web tarayıcıları hala bunu desteklemektedir ve `ftp://` protokolüyle başlayan URI'lar görebiliriz.

Web tarayıcıları ortaya çıkmadan önce `ftp` programı vardı. `ftp`, ağ üzerinden yüklenip indirilebilen dosyaları içeren makineler olan *FTP sunucuları* ile iletişim kurmak için kullanılır.

FTP (orijinal haliyle) hesap adlarını ve şifreleri *açık metin* olarak gönderdiği için güvenli değildir. Bu, bunların şifrelenmediği ve ağı *dinleyen* herkesin bunları görebileceği anlamına gelir. Bu nedenle, İnternet üzerinden yapılan neredeyse tüm FTP işlemleri *anonim FTP sunucuları* tarafından gerçekleştirilir. Anonim bir sunucu, herkesin "anony-mous" kullanıcı adı ve anlamsız bir şifre kullanarak oturum açmasına izin verir.

Aşağıdaki örnekte, `ftp` programı ile anonim FTP sunucusu `fileserver'ın /pub/cd_images/Ubuntu-24.04` dizininde bulunan bir Ubuntu ISO görüntüsünü indiren hayali bir oturumu gösteriyoruz:

```
[me@linuxbox ~]$ ftp fileserver
fileserver.localdomain.
220 (vsFTPd 2.0.1)
Ad (fileserver:me): anonymous
331 Lütfen şifreyi belirtin. Şifre:
230 Giriş başarılı. Uzak
sistem türü UNIX.
Dosyaları aktarmak için ikili mod
kullanılıyor. ftp> cd
pub/cd_images/Ubuntu-24.04
250 Dizin başarıyla değiştirildi. ftp> ls
200 PORT komutu başarılı. PASV kullanmayı düşünün.
150 Dizin listesi geliyor.
-rw-rw-r--     1 500      500       733079552 25 Nisan 03:53
ubuntu-24.04-desktop-amd64.iso
226 Dizin gönderildi,
tamam. ftp> lcd Masaüstü
Yerel dizin şimdi /home/me/Desktop ftp>
ubuntu-24.04-desktop-amd64.iso al
yerel: ubuntu-24.04-desktop-amd64.iso uzak: ubuntu-24.04-desktop-
amd64.iso
200 PORT komutu başarılı. PASV kullanmayı düşünün.
150 ubuntu-24.04-desktop-amd64.iso (733079552 bayt) için BINARY modu
veri bağlantısı açılıyor.
226 Dosya gönderimi tamamlandı.
68,56 saniyede 733079552 bayt alındı (10441,5 kB/s) ftp> bye
```

Tablo 16-1, bu oturum sırasında girilen komutların açıklamasını içerir.

Tablo 16-1: Etkileşimli `ftp` Komutlarına Örnekler

Komut	Anlam
`ftp fileserver`	`ftp` programını çağırın ve FTP sunucusu `fileserver'a` bağlanmasını sağlayın.
`anonim`	Giriş adı. Giriş isteminden sonra, bir

	şifre istemini görürsünüz. Bazı sunucular boş şifreyi kabul ederken, diğerleri e-posta adresi biçiminde bir şifre ister. Bu durumda,`user@example.com` gibi bir şifre deneyin.
`cd pub/cd_images/Ubuntu-24.04`	İstenen dosyayı içeren uzak istemcisinde istenen dosyayı içeren dizine geçin. Çoğu anonim FTP sunucusunda, genel indirmeye açık dosyalar `pub` dizini altında bir yerde bulunur.
`ls`	Uzak sistemdeki dizini listeleyin sistemdeki dizini listeleyin.
`lcd Masaüstü`	Yerel sistemdeki dizini `~/Desktop` olarak değiştirir. Örnekte, çalışma dizini ~ iken `ftp` programı çağrılmıştır. Bu komut, çalışma dizinini `~/Desktop` olarak değiştirir.
`get ubuntu-24.04-desktop-amd64.iso`	Uzak sisteme `ubuntu-24.04-desktop-amd64.iso` dosyasını yerel sisteme aktarmasını söyleyin. Yerel sistemdeki çalışma dizini `~/Masaüstü`, dosya buraya indirilecektir.
`Hoşçakal`	Uzak sunucudan çıkış yapın ve `ftp` programı oturumunu sonlandırın. `quit` ve `exit komutları` da kullanılabilir.

`ftp>` isteminde `help` yazdığınızda desteklenen komutların listesi görüntülenir. Yeterli izinlerin verildiği bir sunucuda `ftp` kullanarak birçok sıradan dosya yönetimi görevini gerçekleştirmek mümkündür. Beceriksizce bir yöntemdir, ancak işe yarar.

`lftp` – Daha İyi Bir `ftp`

`ftp`, tek komut satırı FTP istemcisi değildir. Aslında, birçok komut satırı FTP istemcisi vardır. Daha iyi

(ve daha popüler) olanlardan biri Alexander Lukyanov tarafından geliştirilen `lftp'dir`. Geleneksel `ftp` programına çok benzer şekilde çalışır, ancak çoklu protokol desteği (HTTP dahil), başarısız indirmelerde otomatik yeniden deneme, arka plan işlemleri, yol adlarının sekme tamamlama ve daha birçok ek kolaylık özelliğine sahiptir.

`curl` - Bir URL'yi aktarın

Bir başka popüler dosya aktarım programı da `curl'dur`. En temel kullanımı şu şekildedir:

```
[me@linuxbox ~]$ curl https://linuxcommand.org
```

Bir URL belirtiyoruz ve `curl`, URL'nin ilk sayfasını indirip standart çıktıya gönderiyor. Birden fazla URL belirtilebilir.

`curl`, HTTP, HTTPS, FTP, IMAP, POP3, SFTP, SMB ve diğerleri dahil olmak üzere çoğu ağ protokolünü destekler. Tablo 16-2, `curl'un` desteklediği *birçok* seçenekten birkaçını listeler.

Tablo 16-2: Yaygın curl seçenekleri

Seçenek	Açıklama
`-o, --output` *dosya*	Çıktıyı standart çıktı yerine belirtilen *dosyaya* gönderir. .
`-O, --remote-name`	`-o` gibi, ancak yerel dosyayı uzak dosyanın adıyla aynı şekilde adlandırır.
`-s, --silent`	İlerleme göstergesini ve hata mesajlarını gizler.
`-u, --proxy-user` *kullanıcı*`:`*şifre*	Bir kullanıcı adı/şifre kombinasyonunu belirtin.
`-v, --verbose`	Yürütme sırasında ayrıntılı mesajları görüntüleyin.

`curl` man sayfası tüm ayrıntıları kapsamaktadır.

`wget` - Etkileşimli olmayan ağ indiricisi

Dosya indirmek için kullanılan bir başka komut satırı programı da `wget'tir`. Bu program, hem web hem de FTP sitelerinden içerik indirmek için kullanışlıdır. Tekli dosyalar, çoklu dosyalar ve hatta tüm siteler indirilebilir. `linuxcommand.org` sitesinin ilk sayfasını indirmek için şunu yapabiliriz

şunu yapabiliriz:

```
[me@linuxbox ~]$ wget http://linuxcommand.org/index.php
--11:02:51-- http://linuxcommand.org/index.php
           => `index.php'
linuxcommand.org çözülüyor... 66.35.250.210
linuxcommand.org|66.35.250.210|:80'e bağlanılıyor... bağlandı. HTTP
isteği gönderildi, yanıt bekleniyor... 200 OK
Uzunluk: belirtilmemiş [text/html]

    [ <=>                                   ] 3.120          --.--K/s

11:02:51 (161,75 MB/s) - `index.php' kaydedildi
[3120]
```

Programın birçok seçeneği, `wget'in` yinelemeli olarak indirme yapmasına, arka planda dosya indirmesine (oturumu kapatıp indirmeye devam etmenize olanak tanır) ve kısmen indirilmiş bir dosyanın indirilmesini tamamlamasına olanak tanır. Bu özellikler, ortalamanın üzerinde bir man sayfasında ayrıntılı olarak açıklanmıştır.

Uzak Ana Bilgisayarlarla Güvenli İletişim

Uzun yıllardır, Unix benzeri işletim sistemleri ağ üzerinden uzaktan yönetilebilme özelliğine sahiptir. İnternetin yaygınlaşmasından önceki ilk zamanlarda, uzak ana bilgisayarlara giriş yapmak için kullanılan birkaç popüler program vardı. Bunlar rlogin ve telnet programlarıydı. Ancak bu programlar, `ftp` programının sahip olduğu aynı ölümcül kusuru taşıyorlardı; tüm iletişimlerini (giriş adları ve şifreler dahil) açık metin olarak iletiyorlardı. Bu da onları İnternet çağında kullanıma tamamen uygunsuz hale getiriyordu.

`ssh`

Bu sorunu çözmek için Secure Shell (SSH) adlı yeni bir protokol geliştirildi. SSH, uzak bir ana bilgisayarla güvenli iletişim kurmanın iki temel sorununu çözer.

1. Uzak ana bilgisayarın söylediği kişi olduğunu doğrular (böylece sözde ortadaki adam saldırılarını önler).
2. Yerel ve uzak ana bilgisayarlar arasındaki tüm iletişimi şifreler.

SSH iki bölümden oluşur. Bir SSH sunucusu uzak ana bilgisayarda çalışır ve varsayılan olarak 22 numaralı bağlantı noktasında gelen bağlantıları dinlerken, bir SSH istemcisi yerel sistemde uzak sunucuyla iletişim kurmak için kullanılır.

Çoğu Linux dağıtımı, OpenBSD projesinden OpenSSH adlı bir SSH uygulamasını içerir.

Bazı dağıtımlar varsayılan olarak hem istemci hem de sunucu paketlerini içerirken, diğerleri sadece istemciyi sağlar. Bir sistemin uzaktan bağlantıları alabilmesi için `OpenSSH-server` paketinin kurulu, yapılandırılmış ve çalışır durumda olması ve (sistem çalışıyorsa veya bir güvenlik duvarının arkasında ise) TCP bağlantı noktası 22'de gelen ağ bağlantılarına izin vermesi gerekir.

İpucu: Bağlanacak bir uzak sisteminiz yoksa, ancak bu örnekleri denemek istiyorsanız, `OpenSSH-server` paketinin sisteminizde kurulu olduğundan emin olun ve uzak ana bilgisayar adı olarak `localhost` kullanın. Bu şekilde, makineniz kendisiyle ağ bağlantıları oluşturacaktır.

Uzak SSH sunucularına bağlanmak için kullanılan SSH istemci programı, uygun bir şekilde `ssh` olarak adlandırılır. `remote-sys` adlı uzak bir ana bilgisayara bağlanmak için `ssh` istemci programını şu şekilde kullanırız:

```
[me@linuxbox ~]$ ssh remote-sys
'remote-sys (192.168.1.4)' ana bilgisayarının gerçekliği doğrulanamıyor.
RSA anahtar parmak izi 41:ed:7a:df:23:19:bf:3c:a5:17:bc:61:b3:7f:d9:bb.
Bağlantıya devam etmek istediğinizden emin misiniz (evet/hayır)?
```

Bağlantı ilk kez denendiğinde, uzak ana bilgisayarın gerçekliği doğrulanamıyor olduğunu belirten bir mesaj görüntülenir. Bunun nedeni, istemci programının bu uzak ana bilgisayarı daha önce hiç görmemiş olmasıdır. Uzak ana bilgisayarın kimlik bilgilerini kabul etmek için, istendiğinde "evet" yazın. Bağlantı kurulduktan sonra, kullanıcıdan bir parola girilmesi istenir:

```
Uyarı: 'remote-sys,192.168.1.4' (RSA) bilinen ana bilgisayarlar listesine
kalıcı olarak eklendi.
me@remote-sys'in şifresi:
```

Şifre başarıyla girildikten sonra, uzak sistemden kabuk istemini alırız.

```
Son oturum açma: 30 Ağustos 2025
Cumartesi 13:00:48 [me@remote-sys ~]$
```

Uzak kabuk oturumu, kullanıcı uzak kabuk isteminde `exit` komutunu girerek uzak bağlantıyı kapatana kadar devam eder. Bu noktada, yerel kabuk oturumu devam eder ve yerel kabuk istemini yeniden görünür.

Farklı bir kullanıcı adı kullanarak uzak sistemlere bağlanmak da mümkündür. Örneğin, yerel kullanıcı "me" uzak bir sistemde "bob" adlı bir hesaba sahipse, kullanıcı `me` uzak sistemdeki `bob` hesabına şu şekilde giriş yapabilir:

```
[me@linuxbox ~]$ ssh bob@remote-sys
bob@remote-sys'in şifresi:
Son oturum açma: 30 Ağustos 2025
Cumartesi 13:03:21 [bob@remote-sys
~]$
```

Daha önce belirtildiği gibi, `ssh` uzak ana bilgisayarın gerçekliğini doğrular. Uzak ana bilgisayar başarılı bir şekilde doğrulanmazsa, aşağıdaki mesaj görüntülenir:

```
[me@linuxbox ~]$ ssh remote-sys
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ @
     UYARI: UZAK ANA BİLGİSAYARIN TANIMLAMASI DEĞİŞTİ!         @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
BİRİSİ KÖTÜ BİR ŞEY YAPMAYA ÇALIŞIYOR OLABİLİR!
Şu anda biri sizi dinliyor olabilir (ortadaki adam saldırısı)!
Ayrıca, ana bilgisayar anahtarı yeni değiştirilmiş olabilir. Uzak
ana bilgisayar tarafından gönderilen ECDSA anahtarının parmak izi
SHA256:9uhOXHzPz0AjIs0ZDanNYfv7ksUO4Mjy5pR4kUTSKkA'dır.
Lütfen sistem yöneticinize başvurun.
Bu mesajdan kurtulmak için /home/me/.ssh/known_hosts dosyasına doğru
anahtar anahtarını ekleyin.
/home/me/.ssh/known_hosts:42 içindeki hatalı ECDSA
  anahtarını şu komutla kaldırın:
  ssh-keygen -f "/home/me/.ssh/known_hosts" -R "remote-sys"
Remote-sys için ana bilgisayar anahtarı değişti ve siz sıkı kontrol talep
ettiniz.
Ana bilgisayar anahtarı doğrulaması başarısız oldu.
```

Bu mesaj, iki olası durumdan birinin sonucu olarak ortaya çıkar. İlk olarak, bir saldırgan ortadaki adam saldırısı girişiminde bulunmuş olabilir. Bu durum nadirdir, çünkü herkes `ssh'nin` kullanıcıyı bu konuda uyardığını bilir. Daha olası olan neden ise, uzak sistemin bir şekilde değiştirilmiş olmasıdır; örneğin, işletim sistemi veya SSH sunucusu yeniden yüklenmiştir. Ancak, güvenlik ve emniyet açısından, ilk olasılık da göz ardı edilmemelidir.

Bu mesaj görüntülendiğinde her zaman uzak sistemin yöneticisine danışın.

Mesajın zararsız bir nedenden kaynaklandığı belirlendikten sonra, sorunu istemci tarafında düzeltmek güvenlidir. Bu, uyarı mesajında verilen öneri kullanılarak yapılır:

```
ssh-keygen -f "/home/me/.ssh/known_hosts" -R "remote-sys"
```

Bu mümkün değilse, bir metin düzenleyici (örneğin `vim`) kullanarak `~/.ssh/known_hosts` dosyasından eski anahtarı kaldırabiliriz. Yukarıdaki örnek mesajda şunu görüyoruz:

```
/home/me/.ssh/known_hosts:42'deki hatalı anahtar
```

Bu, known_hosts dosyasının 42. satırında sorunlu anahtarın bulunduğu anlamına gelir. Bu satırı dosyadan silin, `ssh` programı uzak sistemden yeni kimlik doğrulama bilgilerini kabul edebilecektir.

`ssh`, uzak bir sistemde bir kabuk oturumu açmanın yanı sıra, uzak bir sistemde tek bir komut çalıştırmamızı da sağlar. Örneğin, `remote-sys` adlı uzak bir ana bilgisayarda `free` komutunu çalıştırmak ve sonuçları yerel sistemde görüntülemek için şunu kullanın:

```
[me@linuxbox ~]$ ssh  remote-sys free
me@twin4's password:
              toplam     kullanılmış boş       paylaşıl    tamponlar   önbellek
                                                   an
Mem:          775536    507184    268352           0      110068      154596
-/+                     242520    533016
tamponlar/önbellek:
Takas:      1572856          0    1572856
```

Bu tekniği daha ilginç şekillerde kullanmak mümkündür, örneğin aşağıdaki örnekte olduğu gibi, uzak sistemde ls komutunu çalıştırıp çıktıyı yerel sistemdeki bir dosyaya yönlendiriyoruz:

```
[me@linuxbox ~]$ ssh remote-sys 'ls *' > dirlist.txt
me@twin4'ün şifresi:
[me@linuxbox ~]$
```

Yukarıdaki komutta tek tırnak işaretlerinin kullanıldığına dikkat edin. Bunun nedeni, yol adının yerel makinede genişletilmesini istememiz, bunun yerine

uzak sistemde yapılmasını istiyoruz. Benzer şekilde, çıktının uzak makinedeki bir dosyaya yönlendirilmesini isteseydik, yönlendirme operatörünü ve dosya adını tek tırnak içine alabilirdik.

```
[me@linuxbox ~]$ ssh remote-sys 'ls * > dirlist.txt'
```

SSH ile Tünelleme

SSH aracılığıyla uzak bir ana bilgisayarla bağlantı kurduğunuzda, yerel ve uzak sistemler arasında *şifreli* bir *tünel* oluşturulur. Normalde bu tünel, yerel sistemde yazılan komutların uzak sisteme güvenli bir şekilde iletilmesini ve sonuçların da güvenli bir şekilde geri iletilmesini sağlamak için kullanılır. Bu temel işlevine ek olarak, SSH protokolü çoğu türdeki ağ trafiğinin şifreli tünel üzerinden gönderilmesine izin vererek yerel ve uzak sistemler arasında bir tür sanal özel ağ (VPN) oluşturur.

Bu özelliğin belki de en yaygın kullanımı, X Window sistemi trafiğinin iletilmesine izin vermektir. Hala bir X sunucusu çalıştıran bir sistemde (Wayland kullanan modern bir sistem yerine), uzak bir sistemde bir X istemci programı (grafiksel bir uygulama) başlatıp çalıştırmak ve ekranını yerel sistemde görüntülemek mümkündür. Bunu yapmak kolaydır; işte bir örnek. X sunucusu çalıştıran `linuxbox` adlı bir Linux sisteminde oturuyoruz ve `remote-sys` adlı uzak bir sistemde `xload` programını çalıştırarak programın grafiksel çıktısını yerel sistemimizde görmek istiyoruz. Bunu şu şekilde yapabiliriz:

```
[me@linuxbox ~]$ ssh -X remote-sys
me@remote-sys'in şifresi:
Son giriş: 08 Eylül 2025 Pazartesi
13:23:11 [me@remote-sys ~]$ xload
```

`xload` komutu uzak sistemde çalıştırıldıktan sonra, penceresi yerel sistemde görünür. Bazı sistemlerde, bunu yapmak için "-X" seçeneği yerine "-Y" seçeneğini kullanmanız gerekebilir.

`scp` ve `sftp`

OpenSSH paketi, SSH şifreli tüneli kullanarak ağ üzerinden dosya kopyalamak için kullanılabilen iki program da içerir. İlki, `scp` (güvenli kopyalama), dosya kopyalamak için tanıdık `cp` programına çok benzer şekilde kullanılır. En belirgin farkı, kaynak veya hedef yol adlarının önüne uzak ana bilgisayar adı, ardından iki nokta üst üste işareti eklenebilmesidir.

Örneğin, uzak sistemdeki `remote-sys` adlı ev dizinimizden `document.txt` adlı bir belgeyi yerel sistemimizdeki geçerli çalışma dizinine kopyalamak istersek, bunu şu şekilde yapabiliriz:

```
[me@linuxbox ~]$ scp remote-sys:document.txt .
me@remote-sys'in şifresi:
document.txt                              100% 5581      5,5 KB/s   00:00
[me@linuxbox ~]$
```

`ssh` ile olduğu gibi, istenen uzak ana bilgisayar hesap adı yerel sisteminkiyle eşleşmiyorsa, uzak ana bilgisayar adının başına bir kullanıcı adı ekleyebilirsiniz.

```
[me@linuxbox ~]$ scp bob@remote-sys:document.txt .
```

İkinci SSH dosya kopyalama programı, adından da anlaşılacağı gibi, ftp programının güvenli bir alternatifi olan `sftp'dir`. `sftp`, daha önce kullandığımız orijinal `ftp` programına çok benzer şekilde çalışır; ancak, her şeyi düz metin olarak iletmek yerine, SSH şifreli tünel kullanır. `sftp`, uzak ana bilgisayarda bir FTP sunucusunun çalışmasını gerektirmediği için geleneksel `ftp'ye` göre önemli bir avantaja sahiptir. Yalnızca SSH sunucusu gerektirir. Bu, SSH istemcisine bağlanabilen herhangi bir uzak makinenin FTP benzeri bir sunucu olarak da kullanılabileceği anlamına gelir. İşte bir örnek oturum:

```
[me@linuxbox ~]$ sftp remote-sys
remote-sys'e bağlanılıyor...
me@remote-sys'in şifresi:
sftp> ls
ubuntu-24.04-desktop-amd64.iso
sftp> lcd Masaüstü
sftp> ubuntu-24.04-desktop-amd64.iso dosyasını al
/home/me/ubuntu-24.04-desktop-amd64.iso dosyasını ubuntu-24.04-
desktop-amd64.iso dosyasına indiriliyor
/home/me/ubuntu-24.04-desktop-amd64.iso 100% 699MB       7,4        01:35
sftp> bye                                                MB/s
```

İpucu: SFTP protokolü, Linux dağıtımlarında bulunan birçok grafik dosya yöneticisi tarafından desteklenir. GNOME veya KDE kullanarak, konum çubuğuna `sftp://` ile başlayan bir URI girebilir ve SSH sunucusu çalıştıran bir

SSH sunucusu çalıştıran uzak sistemde bulunan dosyalar üzerinde işlem yapabiliriz.

Windows için bir SSH İstemcisi mi?

Diyelim ki bir Windows makinesinde oturuyorsunuz, ancak Linux sunucunuza giriş yapıp bazı işleri halletmeniz gerekiyor; ne yaparsınız? Elbette Windows makineniz için bir SSH istemci programı edinirsiniz! Bunların birçoğu vardır. En popüler olanı muhtemelen Simon Tatham ve ekibi tarafından geliştirilen PuTTY'dir. PuTTY programı bir terminal penceresi görüntüler ve Windows kullanıcısının uzak bir ana bilgisayarda SSH (veya telnet) oturumu açmasına olanak tanır. Program ayrıca `scp` ve `sftp` programları için benzerler de sağlar.

PuTTY, http://www.chiark.greenend.org.uk/~sgtatham/putty/ adresinden edinilebilir.

Özet

Bu bölümde, çoğu Linux sisteminde bulunan birkaç ağ aracını inceledik. Linux, sunucularda ve ağ cihazlarında çok yaygın olarak kullanıldığından, ek yazılımlar yükleyerek daha birçok araç eklenebilir. Ancak temel araç setiyle bile, ağla ilgili birçok yararlı görevi gerçekleştirmek mümkündür.

Daha Fazla Okuma

- Ağ yönetimine genel (ancak güncel olmayan) bir bakış için, Linux Dokümantasyon Projesi *Linux Ağ Yöneticisi Kılavuzu* sunmaktadır: https://tldp.org/LDP/nag2/index.html
- Wikipedia, ağ yönetimi ile ilgili birçok iyi makale içerir. İşte temel bilgilerden bazıları:
  https://en.wikipedia.org/wiki/Internet_protocol_address
  https://en.wikipedia.org/wiki/Host_name
  https://en.wikipedia.org/wiki/Uniform_Resource_Identifier
- `Curl` ana sayfasında, `curl'ün` yapabileceği birçok örneği gösteren mükemmel bir eğitim bulunmaktadır: https://curl.se/docs/tutorial.html
- Otomatik Özel IP Adresleme (APIPA) özelliğinin nasıl ve neden kullanıldığına dair bir açıklama mı istiyorsunuz? Şunu deneyin: https://www.33rdsquare.com/what-does-169-254-0-0-mean/

17 – Dosya Arama

Linux sistemimizi incelerken bir şey çok net bir şekilde ortaya çıktı: tipik bir Linux sisteminde çok sayıda dosya var! Bu da şu soruyu akla getiriyor: "Dosyaları nasıl bulabiliriz?" Linux dosya sisteminin, bir nesil Unix benzeri sistemden diğerine aktarılan kurallara göre iyi organize edildiğini zaten biliyoruz, ancak dosya sayısının çokluğu zorlu bir sorun oluşturabilir.

Bu bölümde, sistemde dosya bulmak için kullanılan iki aracı inceleyeceğiz.

- `locate` – Dosyaları ada göre bulma
- `find` – Dizin hiyerarşisinde dosya arama

Ayrıca, dosya arama komutlarıyla sıklıkla kullanılan ve sonuçta ortaya çıkan dosya listesini işlemek için kullanılan bir komutu da inceleyeceğiz.

- `xargs` – Standart girişten komut satırları oluşturma ve yürütme

Ayrıca, araştırmalarımızda bize yardımcı olacak birkaç komut tanıtacağız.

- `touch` – Dosya zamanlarını değiştir
- `stat` – Dosya veya dosya sistemi durumunu görüntüleme

`locate` – Dosyaları Kolay Yoldan Bulun

`Locate` programı, yol adlarının hızlı bir şekilde veritabanında aranmasını sağlar ve ardından verilen alt dizeyle eşleşen tüm adları görüntüler. Örneğin, adı `zip` ile başlayan tüm programları bulmak istediğimizi varsayalım. Programları aradığımız için, programları içeren dizinin adının bin/ ile bittiğini varsayabiliriz. Bu nedenle, dosyalarımızı bulmak için `locate komutunu` şu şekilde kullanabiliriz:

```
[me@linuxbox ~]$ locate bin/zip
```

`loc`ate, yol adları veritabanını arar ve
bin`/zip`

```
/usr/bin/zip
/usr/bin/zipcloak
/usr/bin/zipinfo
/usr/bin/zipinfo
/usr/bin/zipnote
/usr/bin/zipsplit
```

Arama gereksinimi çok basit değilse, `locate komutunu grep` gibi diğer araçlarla birleştirerek daha ilginç aramalar tasarlayabiliriz.

```
[me@linuxbox ~]$ locate zip | grep bin
/bin/bunzip2
/bin/bzip2
/bin/bzip2recover
/bin/gunzip
/bin/gzip
/usr/bin/funzip
/usr/bin/gpg-zip
/usr/bin/preunzip
/usr/bin/prezip
/usr/bin/prezip-bin
/usr/bin/unzip
/usr/bin/unzipsfx
/usr/bin/zip
/usr/bin/zipcloak
/usr/bin/zipgrep
/usr/bin/zipinfo
/usr/bin/zipnote
/usr/bin/zipsplit
```

`Locate` programı yıllardır kullanılmaktadır ve yaygın olarak kullanılan birkaç varyantı vardır. Modern Linux dağıtımlarında en yaygın olarak kullanılan ikisi `plocate` ve `mlocate`'dir, ancak genellikle `locate` adlı bir sembolik bağlantı ile erişilirler. `Locate'in` farklı sürümleri, birbiriyle örtüşen seçenekler setlerine sahiptir. Bazı sürümler, düzenli ifade eşleştirme (bunu Bölüm 19'da ele alacağız) ve joker karakter desteği içerir. Hangi `locate` sürümünün kurulu olduğunu belirlemek için `locate` man sayfasını kontrol edin.

`locate` Veritabanı Nereden Geliyor?

Bazı dağıtımlarda, sistem kurulduktan hemen sonra `locate` komutunun çalışmadığını, ancak ertesi gün tekrar denediğinizde sorunsuz çalıştığını fark edebilirsiniz. Neden? `Locate` veritabanı, `updatedb` adlı başka bir program tarafından oluşturulur. Genellikle, *cron işi* olarak, yani cron daemon tarafından düzenli aralıklarla gerçekleştirilen bir görev olarak periyodik olarak çalıştırılır. `Locate` ile donatılmış çoğu sistem, `updatedb'yi` günde bir kez çalıştırır. Veritabanı sürekli güncellenmediğinden, `locate'u` kullanırken çok yeni dosyaların görünmediğini fark edeceksiniz. Bunu aşmak için, süper kullanıcı olarak oturum açıp komut isteminde `updatedb'yi` çalıştırarak `updatedb` programını manuel olarak çalıştırmak mümkündür.

`find` – Dosyaları Zor Yoldan Bulma

`Locate` programı yalnızca adına göre bir dosyayı bulabilirken, `find` programı belirli bir dizini (ve alt dizinlerini) çeşitli özelliklere göre dosyaları arar. `Find` ile çok zaman geçireceğiz çünkü daha sonraki bölümlerde programlama kavramlarını ele almaya başladığımızda tekrar tekrar göreceğimiz birçok ilginç özelliği var.

En basit kullanımında, `find'a` aranacak bir veya daha fazla dizin adı verilir. Örneğin, ana dizininizin bir listesini oluşturmak için şunu kullanabilirsiniz:

```
[me@linuxbox ~]$ find ~
```

Çoğu aktif kullanıcı hesabında, bu komut büyük bir liste oluşturur. Liste standart çıktıya gönderildiğinden, listeyi diğer programlara aktarabiliriz. `wc komutunu` kullanarak dosya sayısını sayalım.

```
[me@linuxbox ~]$ find ~ | wc -l
47068
```

Vay canına, çok meşgulmüşüz! `find` komutunun güzelliği, belirli kriterlere uyan dosyaları tanımlamak için kullanılabilmesidir. Bunu, (biraz garip) *seçenekler*, *testler* ve *eylemler* uygulayarak yapar. Önce testlere bakalım.

Testler

Diyelim ki aramamızdan bir dizin listesi istiyoruz. Bunu yapmak için aşağıdaki testi ekleyebiliriz:

```
[me@linuxbox ~]$ find ~ -type d | wc -l
1695
```

`-type d` testini eklemek, aramayı dizinlerle sınırladı. Tersine, bu testle aramayı normal dosyalarla sınırlayabilirdik:

```
[me@linuxbox ~]$ find ~ -type f | wc -l
38737
```

Tablo 17-1, `find` tarafından desteklenen yaygın dosya türü testlerini listeler.

Tablo 17-1: Dosya Türlerini `Bulma`

Dosya Türü	Açıklama
`b`	Özel aygıt dosyası bloğu
`c`	Karakter özel aygıt dosyası
`d`	Dizin
`f`	Normal dosya
`l`	Sembolik bağlantı

Bazı ek testler ekleyerek dosya boyutuna ve dosya adına göre de arama yapabiliriz. Joker karakter kalıbı `*.JPG` ile eşleşen ve 1 MB'den büyük tüm normal dosyaları arayalım.

```
[me@linuxbox ~]$ find ~ -type f -name "*.JPG" -size +1M | wc -l 840
```

Bu örnekte, `-name` test ifadesini ve ardından joker karakter desenini ekliyoruz. Kabuğun yol adını genişletmesini önlemek için bunu tırnak içine aldığımıza dikkat edin. Ardından, `-size` test ifadesini ve ardından "+1M" dizesini ekliyoruz. Baştaki artı işareti, belirtilen sayıdan daha büyük dosyaları aradığımızı gösterir. Baştaki eksi işareti, dizenin anlamını belirtilen sayıdan daha küçük olarak değiştirir. İşaret kullanmamak, "değeri tam olarak eşleştir" anlamına gelir

tam olarak eşleş" anlamına gelir. Sonundaki `M` harfi, ölçü biriminin megabayt olduğunu belirtir. Tablo 17-2, birimleri belirtmek için kullanılabilecek karakterleri listeler.

Tablo 17-2: `find` Boyut Birimleri

Karakter	Birim
`b`	512 baytlık bloklar. Birim belirtilmezse varsayılan ayardır.
`c`	Bayt.
`w`	2 baytlık kelimeler.
`k`	Kilobayt (1.024 baytlık birimler).
`M`	Megabayt (1.048.576 baytlık birimler).
`G`	Gigabayt (1.073.741.824 baytlık birimler).

find çok sayıda testi destekler. Tablo 17-3 yaygın olanların bir özetini sunar. Sayısal bir argüman gerektiğinde, yukarıda açıklanan aynı `+` ve `−` notasyonu uygulanabilir.

Tablo 17-3: `find` Testleri

Test	Açıklama
`-cmin` *`n`*	İçeriği veya öznitelikleri en son *`n`* dakika önce değiştirilen dosya veya dizinleri eşleştirir. *`n`* dakikadan daha kısa bir süre önceyi belirtmek için *`−n`*, *`n`* dakikadan daha uzun bir süre önceyi belirtmek için *`+n`* kullanın.
`-cnewer` *`dosyası`*	İçeriği veya öznitelikleri dosyadan daha yakın zamanda değiştirilmiş olan dosya veya *`dizinleri`* eşleştirir.
`-ctime` *`n`*	İçeriği veya öznitelikleri (son n*24 saat önce değiştirilen dosya veya dizinleri eşleştirir.
`-empty`	Boş dosya ve dizinleri eşleştirir.
`-group` *`name`*	*`Gruba`* ait dosya veya dizinleri eşleştir. *`grup`* grup adı veya sayısal grup kimliği olarak ifade edilebilir.
`-iname` *`kalıbı`*	`-name` testi gibi, ancak büyük/küçük harf duyarlı değildir.
`-inum` *`n`*	Inode numarası *`n`* olan dosyaları eşleştirir. Bu,

	tüm sabit bağlantıları bulmak için yararlıdır.
`-mmin` *`n`*	İçeriği en son *`n`* dakika önce değiştirilen dosya veya dizinleri eşleştirir.
`-mtime` *`n`*	İçeriği en son n*24 saat önce değiştirilen dosya veya dizinleri eşleştirir.
`-name` *`desen`*	Belirtilen joker karakter desenine sahip dosya ve dizinleri eşleştirir. *`deseniyle eşleşen dosya ve dizinleri eşleştirir`*.
`-daha yeni` *`dosya`*	İçeriği belirtilen dosyadan Bu, dosya yedeklemesi yapan kabuk betikleri yazarken kullanışlıdır. Her yedekleme yaptığınızda, bir dosyayı (günlük dosyası gibi) güncelleyin ve ardından `find komutunu` kullanarak son güncellemeden bu yana hangi dosyaların değiştiğini belirleyin.
`-nouser`	Geçerli bir kullanıcıya ait olmayan dosya ve dizinleri eşleştirir kullanıcıya ait olmayan dosya ve dizinleri eşleştirir. Bu, silinmiş hesaplara ait dosyaları bulmak veya saldırganların faaliyetlerini tespit etmek için kullanılabilir.
`-nogroup`	Geçerli bir gruba ait olmayan dosya ve dizinleri eşleştirir gruba ait olmayan dosya ve dizinleri eşleştirir.
`-perm` *`modu`*	İzinleri belirtilen *`mod`*. *`mod`*, sekizlik veya sembolik gösterimle ifade edilebilir.
`-samefile` *`adı`*	`-inum` testine benzer. Aynı inode numarasını paylaşan dosyaları *`eşleştirir`*.
`-size` *`n`*	Boyutu *`n`* olan dosyaları eşleştirir.
`-type` *`c`*	*`c`* türündeki dosyaları eşleştirir.
`-kullanıcı` *`adı`*	Kullanıcı *`adına`* ait dosya veya dizinleri eşleştirir. Kullanıcı kullanıcı adı veya sayısal kullanıcı kimliği ile ifade edilebilir.

Bu liste eksiksiz değildir. `find` man sayfasında tüm ayrıntılar bulunmaktadır.

Operatörler

`find komutunun` sağladığı tüm testlere rağmen, testler arasındaki *mantıksal ilişkileri* tanımlamak için daha iyi bir yönteme ihtiyaç duyabiliriz. Örneğin, bir dizindeki tüm dosyaların ve alt dizinlerin güvenli izinlere sahip olup olmadığını belirlememiz gerekirse ne yapmalıyız?

0600 olmayan izinlere sahip tüm dosyaları ve 0700 olmayan izinlere sahip dizinleri arardık. Neyse ki, `find` *mantıksal operatörleri* kullanarak testleri birleştirip daha karmaşık mantıksal ilişkiler oluşturmanın bir yolunu sunar. Yukarıda bahsedilen testi ifade etmek için şunu yapabiliriz:

```
[me@linuxbox ~]$ find ~ \( -type f -not -perm 0600 \) -or \( -type d
-not -perm 0700 \)
```

Vay canına! Bu gerçekten garip görünüyor. Bunlar neyin nesi? Aslında, operatörleri bir kez öğrendikten sonra o kadar da karmaşık değiller. Tablo 17-4, `find` ile kullanılan mantıksal operatörleri açıklamaktadır.

Tablo 17-4: `find` Mantıksal Operatörleri

Operatör	Açıklama
`-ve`	Operatörün her iki tarafındaki testler doğruysa eşleşir. Bu, `-a` olarak kısaltılabilir. Operatör bulunmadığında, varsayılan olarak `-and` kullanıldığına dikkat edin.
`-or`	Operatörün her iki tarafındaki testlerden biri doğruysa eşleşir. Bu, `-o` olarak kısaltılabilir.
`-not`	Operatörün ardından gelen test yanlışsa eşleşir. Bu, ünlem işareti (`!`) ile kısaltılabilir.
`( )`	Testleri ve operatörleri bir araya getirerek daha büyük ifadeler oluşturur. Bu, mantıksal değerlendirmelerin önceliğini kontrol etmek için kullanılır. Varsayılan olarak, `find` soldan sağa doğru değerlendirir. İstenen sonucu elde etmek için genellikle varsayılan değerlendirme sırasını geçersiz kılmak gerekir. Gerekli olmasa bile, komutun okunabilirliğini artırmak için bazen gruplama karakterlerini eklemek yararlı olabilir. Parantezlerin kabuk için özel bir anlamı olduğundan, komut satırında kullanıldıklarında `find'a` argüman olarak aktarılabilmeleri için tırnak içine alınmaları gerekir. Genellikle ters eğik çizgi karakteri bunları kaçırmak için kullanılır. `(` ve `)` karakterlerinin komuttaki diğer kelimelerden ayırmak için boşluklarla çevrilmesi de önemlidir. Örneğin, `find ~ \( -type f \)`

Elimizdeki bu operatör listesiyle, `find` komutumuzu parçalara ayıralım. En üst düzeyden bakıldığında

en üst düzeyden bakıldığında, testlerimizin `-or` operatörü ile ayrılmış iki grup halinde düzenlendiğini görürüz.

```
( ifade 1 ) -or ( ifade 2 )
```

Bu mantıklıdır, çünkü belirli bir izin kümesine sahip dosyaları ve farklı bir izin kümesine sahip dizinleri arıyoruz. Hem dosyaları hem de dizinleri arıyorsak, neden `-and` yerine `-or` kullanıyoruz? `Find`, dosyaları ve dizinleri tararken, her biri belirtilen testlerle eşleşip eşleşmediğini görmek için değerlendirilir. Bunun, izinleri hatalı bir dosya *mı yoksa* izinleri hatalı bir dizin *mi* olduğunu bilmek istiyoruz. İkisi aynı anda olamaz. Dolayısıyla, gruplandırılmış ifadeleri genişletirsek, bunu şu şekilde görebiliriz:

```
(hatalı izinlere sahip dosya) -veya (hatalı izinlere sahip dizin)
```

Bir sonraki zorluğumuz, "hatalı izinleri" nasıl test edeceğimizdir. Bunu nasıl yapacağız? Aslında, yapmayacağız. Test edeceğimiz şey, "iyi izinler"in ne olduğunu bildiğimiz için "iyi olmayan izinler"dir. Dosyalar için iyi olanı 0600 olarak, dizinler için ise 0700 olarak tanımlıyoruz. Dosyaları "iyi olmayan" izinler açısından test edecek ifade şöyledir:

```
-type f -and -not -perm 0600
```

Dizinler için ise şu şekildedir:

```
-type d -and -not -perm 0700
```

Yukarıdaki Tablo 17-4'te belirtildiği gibi, `-and` operatörü varsayılan olarak ima edildiğinden güvenle kaldırılabilir. Tüm bunları bir araya getirirsek, nihai komutumuzu elde ederiz.

```
find ~ ( -type f -not -perm 0600 ) -or ( -type d -not -perm 0700 )
```

Ancak, parantezlerin kabuk için özel bir anlamı olduğundan, kabuğun bunları yorumlamasını önlemek için bunları kaçırmamız gerekir. Her birinin önüne ters eğik çizgi karakteri koymak bu işi görür.

Mantıksal operatörlerin anlaşılması gereken bir başka özelliği daha vardır. Mantıksal bir operatörle ayrılmış iki ifademiz olduğunu varsayalım.

```
expr1 -operator expr2
```

Her durumda, *`expr1`* her zaman gerçekleştirilir; ancak, *`expr2'nin`* gerçekleştirilip gerçekleştirilmeyeceği operatör tarafından belirlenir. Tablo 17-5 bunun nasıl çalıştığını özetlemektedir.

Tablo 17-5: AND/OR Mantığı `Bulma`

***`expr1`*'in sonuçları**	**Operatör**	***`expr2`* şudur...**
Doğru	`-ve`	Her zaman gerçekleştirilir
Yanlış	`-ve`	Asla gerçekleştirilmez
Doğru	`-veya`	Hiç gerçekleştirilmedi

Yanlış	`-veya`	Her zaman gerçekleştirildi

Bu neden olur? Performansı artırmak için yapılır. Örneğin `-and'ı` ele alalım. *`Expr1`* ifadesinin sonucu yanlışsa, *`expr1 -and expr2`* ifadesinin doğru olamayacağını biliyoruz, bu nedenle *`expr2 ifadesini`* gerçekleştirmekte bir anlam yoktur. Benzer şekilde, *`expr1 -or expr2`* ifadesine sahipsek ve *`expr1`* ifadesinin sonucu doğruysa, *`expr1 -or expr2`* ifadesinin doğru olduğunu zaten bildiğimiz için *`expr2`* ifadesini gerçekleştirmekte bir anlam yoktur.

Tamam, bu daha hızlı çalışmasına yardımcı oluyor. Bu neden önemli? Bu davranışa güvenerek eylemlerin nasıl gerçekleştirileceğini kontrol edebileceğimiz için önemlidir, bunu birazdan göreceğiz.

Önceden Tanımlanmış Eylemler

Hadi biraz iş yapalım! Find komutumuzun sonuçlarının bir listesine sahip olmak yararlıdır, ancak asıl yapmak istediğimiz şey listedeki öğeler üzerinde işlem yapmaktır. Neyse ki, `find komutu` arama sonuçlarına göre işlemlerin gerçekleştirilmesine izin verir. Önceden tanımlanmış bir dizi işlem ve kullanıcı tanımlı işlemleri uygulamak için birkaç yol vardır. Öncelikle, Tablo 17-6'da listelenen önceden tanımlanmış işlemlerden birkaçına bakalım.

Tablo 17-6: Önceden tanımlanmış `find` eylemleri

Eylem	Açıklama
`-delete`	Şu anda eşleşen dosyayı siler.
`-ls`	Eşleşen dosya üzerinde `ls -dils` komutuna eşdeğer bir işlem gerçekleştirir. Çıktı standart çıktıya gönderilir.
`-print`	Eşleşen dosyanın tam yol adını standart çıktıya yazdırır. Başka bir eylem belirtilmediğinde varsayılan eylem budur.
`-quit`	Eşleşme bulunduğunda çık.

Testlerde olduğu gibi, daha birçok eylem vardır. Ayrıntılı bilgi için `find` man sayfasına bakın. İlk örnekte şunu yaptık:

```
find ~
```

Bu, ana dizinde bulunan tüm dosya ve alt dizinlerin bir listesini oluşturdu.

Başka bir eylem belirtilmediğinde `-print` eylemi ima edildiği için bir liste oluşturuldu. Dolayısıyla, komutumuz şu şekilde de ifade edilebilir:

```
find ~ -print
```

`find komutunu` kullanarak belirli kriterlere uyan dosyaları silebiliriz. Örneğin, `.bak` uzantılı dosyaları (genellikle yedekleme dosyalarını belirtmek için kullanılır) silmek için şu komutu kullanabiliriz:

```
find ~ -type f -name '*.bak' -delete
```

Bu örnekte, kullanıcının ana dizinindeki (ve alt dizinlerindeki) tüm dosyalar `.bak` ile biten dosya adları için aranır. Bulunduklarında silinirler.

Uyarı: `-delete` eylemini kullanırken ***son derece dikkatli olmanız*** gerektiğini belirtmeye gerek yoktur. Her zaman önce `-delete` yerine `-print` eylemini kullanarak komutu test edin ve arama sonuçlarını doğrulayın.

Devam etmeden önce, mantıksal operatörlerin eylemleri nasıl etkilediğine bir kez daha bakalım. Aşağıdaki komutu ele alalım:

```
~ -type f -name '*.bak' -print
```

Gördüğümüz gibi, bu komut adı `.bak` (`-name '*.bak'`) ile biten her normal dosyayı (`-type f`) arar ve eşleşen her dosyanın göreceli yol adını standart çıktıya (`-print`) gönderir. Ancak, komutun bu şekilde çalışmasının nedeni, her bir test ve eylem arasındaki mantıksal ilişkiler tarafından belirlenir. Unutmayın, varsayılan olarak, her test ve eylem arasında örtük bir `-and` ilişkisi vardır. Mantıksal ilişkileri daha kolay görebilmek için komutu şu şekilde de ifade edebiliriz:

```
find ~ -type f -and -name '*.bak' -and -print
```

Komutumuzu tam olarak ifade ettikten sonra, mantıksal operatörlerin komutun yürütülmesini nasıl etkilediğine bakalım:

Test veya eylem	Yalnızca şu durumda gerçekleştirilir...
`-print`	`-type f` ve `-name '*.bak'` doğruysa
`-name '*.bak'`	`-type f` doğruysa
`-type f`	`-` `ve` ilişkisinde ilk test/eylem olduğu için her zaman gerçekleştirilir.

Testler ve eylemler arasındaki mantıksal ilişki hangilerinin gerçekleştirileceğini belirlediğinden, testlerin ve eylemlerin sırasının önemli olduğunu görebiliriz. Örneğin, `-print` eyleminin ilk sırada olması için testleri ve eylemleri yeniden sıralarsak, komut çok farklı bir şekilde çalışacaktır.

```
find ~ -print -and -type f -and -name '*.bak'
```

Bu komut sürümü, her dosyayı yazdırır (`-print` eylemi her zaman doğru olarak değerlendirilir) ve ardından dosya türünü ve belirtilen dosya uzantısını test eder.

Kullanıcı Tanımlı Eylemler

Önceden tanımlanmış eylemlerin yanı sıra, rastgele komutları da çağırabiliriz. Bunu yapmanın geleneksel yolu `-exec` eylemidir. Bu eylem şu şekilde çalışır:

`-exec` *komut* `{} ;`

Burada *komut*, bir komutun adıdır, {} geçerli yol adının sembolik temsilidir ve noktalı virgül, komutun sonunu belirten gerekli bir sınırlayıcıdır. Aşağıda, `-exec'i` daha önce açıklanan `-delete` eylemi gibi kullanmanın bir örneği verilmiştir:

```
-exec rm '{}' ';'
```

Yine, köşeli parantez ve noktalı virgül karakterleri kabuk için özel bir anlama sahip olduğundan, tırnak içine alınmalı veya kaçış karakteri ile korunmalıdır.

Kullanıcı tanımlı bir eylemi etkileşimli olarak yürütmek de mümkündür. `-exec` yerine `-ok` eylemini kullanarak, her bir belirtilen komutun yürütülmesinden önce kullanıcıya sorulur.

```
find ~ -type f -name 'foo*' -ok ls -l '{}' ';'
< ls ... /home/me/bin/foo > ? y
-rwxr-xr-x 1 me    me 224 2016-10-29 18:44 /home/me/bin/foo
```

```
< ls ... /home/me/foo.txt > ? y
-rw-r--r-- 1 ben   me    0 2025-09-19 12:53 /home/me/foo.txt
```

Bu örnekte, adı `foo` dizesiyle başlayan dosyaları arıyoruz ve her bir dosya bulunduğunda `ls -l` komutunu çalıştırıyoruz. `-ok` eylemi kullanıldığında, `ls` komutu çalıştırılmadan önce kullanıcıya bir uyarı gösterilir.

Verimliliği Artırma

`-exec` eylemi kullanıldığında, eşleşen bir dosya her bulunduğunda belirtilen komutun yeni bir örneği başlatılır. Bazen tüm arama sonuçlarını birleştirip komutun tek bir örneğini başlatmayı tercih edebiliriz. Örneğin, şu komutları çalıştırmak yerine:

`ls -l` *`file1`*

`ls -l` *`file2`*

şeklinde çalıştırmak yerine, şu şekilde çalıştırmayı tercih edebiliriz:

`ls -l` *`dosya1 dosya2`*

Bu, komutun birden çok kez yerine yalnızca bir kez yürütülmesini sağlar. Bunu iki şekilde yapabiliriz: geleneksel yöntem olan `xargs` harici komutunu kullanarak ve alternatif yöntem olan `find` komutunun yeni bir özelliğini kullanarak. Önce alternatif yöntemden bahsedelim.

Sonundaki noktalı virgül karakterini artı işaretine değiştirerek, `find`'ın arama sonuçlarını istenen komutun tek bir yürütülmesi için bir argüman listesinde birleştirme özelliğini etkinleştiririz. Örneğimize geri dönersek, bu, eşleşen bir dosya her bulunduğunda `ls` komutunu çalıştıracaktır:

```
find ~ -type f -name 'foo*' -exec ls -l '{}' ';'
-rwxr-xr-x 1 me    me 224 2016-10-29 18:44 /home/me/bin/foo
-rw-r--r-- 1 ben   me    0 2025-09-19 12:53 /home/me/foo.txt
```

Komutu aşağıdaki şekilde değiştirerek:

```
find ~ -type f -name 'foo*' -exec ls -l '{}' +
-rwxr-xr-x 1 ben   me 224 2016-10-29 18:44 /home/me/bin/foo
-rw-r--r-- 1 ben   be    0 2025-09-19 12:53 /home/me/foo.txt
                   n
```

Aynı sonuçları elde ediyoruz, ancak sistem `ls` komutunu yalnızca bir kez çalıştırmak zorunda.

xargs

`xargs` komutu ilginç bir işlev gerçekleştirir. Standart girişten gelen girdiyi kabul eder ve bunu belirtilen komut için bir argüman listesine dönüştürür. Örneğimizde, bunu şu şekilde kullanırız:

```
find ~ -type f -name 'foo*' -print | xargs ls -l
-rwxr-xr-x 1 me    me 224 2016-10-29 18:44 /home/me/bin/foo
-rw-r--r-- 1 ben   me   0 2025-09-19 12:53 /home/me/foo.txt
```

Burada, `xargs'a` aktarılan `find` komutunun çıktısını görüyoruz. xargs, `ls` komutu için bir argüman listesi oluşturur ve ardından bunu yürütür.

Not: Komut satırına yerleştirilebilecek argüman sayısı oldukça fazla olsa da, sınırsız değildir. Kabuk tarafından kabul edilemeyecek kadar uzun komutlar oluşturmak mümkündür. Bir komut satırı sistem tarafından desteklenen maksimum uzunluğu aştığında, `xargs` belirtilen komutu mümkün olan maksimum argüman sayısı ile yürütür ve standart giriş tükenene kadar bu işlemi tekrarlar. Komut satırının maksimum boyutunu görmek için, `xargs komutunu --show-limits` seçeneği ile çalıştırın.

Komik Dosya Adlarıyla Başa Çıkmak

Unix benzeri sistemler, dosya adlarında gömülü boşluklara (ve hatta satır sonlarına!) izin verir. Bu, diğer programlar için argüman listeleri oluşturan `xargs` gibi programlar için sorunlara neden olur. Gömülü bir boşluk sınırlayıcı olarak değerlendirilir ve sonuçta ortaya çıkan komut, boşlukla ayrılmış her kelimeyi ayrı bir argüman olarak yorumlar. Bunu aşmak için, `find` ve `xargs`, argüman ayırıcı olarak isteğe bağlı olarak *null karakterinin* kullanımına izin verir. Null karakteri, ASCII'de sıfır rakamıyla temsil edilen karakter olarak tanımlanır (örneğin, ASCII'de 32 rakamıyla temsil edilen boşluk karakterinin aksine). `find` komutu, boşlukla ayrılmış çıktı üreten `-print0` eylemini sağlar ve `xargs` komutu, boşlukla ayrılmış girişi kabul eden `--null` (veya `-0`) seçeneğine sahiptir. İşte bir örnek:

```
find ~ -iname '*.jpg' -print0 | xargs --null ls -l
```

Bu tekniği kullanarak, adlarında gömülü boşluklar bulunan dosyalar da dahil olmak üzere tüm dosyaların doğru şekilde işlenmesini sağlayabiliriz.

Oyun Alanına Dönüş

`Find komutunu` (neredeyse) pratik olarak kullanma zamanı geldi. Bir oyun alanı oluşturacak ve öğrendiklerimizin bir kısmını deneyeceğiz.

İlk olarak, çok sayıda alt dizin ve dosya içeren bir oyun alanı oluşturalım.

```
[me@linuxbox ~]$ mkdir -p playground/dir-{001..100} [me@linuxbox
~]$ touch playground/dir-{001..100}/file-{A..Z}
```

Komut satırının gücüne hayran kalın! Bu iki satırla, her biri 26 boş dosya içeren 100 alt dizin içeren bir oyun alanı dizini oluşturduk. Bunu GUI ile deneyin!

Bu sihirli işlemi gerçekleştirmek için kullandığımız yöntem, tanıdık bir komut (`mkdir`), egzotik bir kabuk genişletme (köşeli parantezler) ve yeni bir komut olan `touch'ı` içeriyordu. `mkdir komutunu -p` seçeneği (`mkdir'in` belirtilen yolların üst dizinlerini oluşturmasını sağlar) ile köşeli parantez genişletme ile birleştirerek 100 alt dizin oluşturabildik.

`touch` komutu genellikle dosyaların erişim, değişiklik ve düzenleme zamanlarını ayarlamak veya güncellemek için kullanılır. Ancak, dosya adı argümanı var olmayan bir dosyanın adıysa, boş bir dosya oluşturulur.

Oyun alanımızda, `file-A` adlı bir dosyanın 100 örneğini oluşturduk. Bunları bulalım.

```
[me@linuxbox ~]$ find playground -type f -name 'file-A'
```

`ls komutundan` farklı olarak, `find` komutu sonuçları sıralı olarak göstermez. Sıralama, depolama aygıtının düzenine göre belirlenir. Bu şekilde, dosyanın 100 örneği olduğunu doğrulayabiliriz.

```
[me@linuxbox ~]$ find playground -type f -name 'file-A' | wc -l 100
```

Şimdi, dosyaları değiştirilme zamanlarına göre bulmaya bakalım. Bu, yedeklemeler oluştururken veya dosyaları kronolojik sırayla düzenlerken yararlı olacaktır. Bunu yapmak için, önce değiştirilme zamanını karşılaştıracağımız bir referans dosyası oluşturacağız.

```
[me@linuxbox ~]$ touch playground/timestamp
```

Bu, `timestamp` adlı boş bir dosya oluşturur ve değiştirilme zamanını geçerli zamana ayarlar. Bunu, `ls` komutunun geliştirilmiş bir versiyonu olan `stat` komutunu kullanarak doğrulayabiliriz. `stat` komutu, sistemin bir dosya ve öznitelikleri hakkında bildiği her şeyi gösterir.

```
[me@linuxbox ~]$ stat playground/timestamp
  Dosya: `playground/timestamp'
  Boyut: 0          Bloklar: 0       IO Bloğu: 4096 normal boş dosya
Aygıt: 803h/2051d Inode: 14265061 Bağlantılar: 1
 Erişim:  2025-10-08   15:15:39.00000000   -0400   Gid: ( 1001/ me)
Değiştirm 08.10.2025    15:15:39.000000    -0400
   e:
Değişikli 08.10.2025       15:15:39       -0400
   k:
```

`Touch komutunu` tekrar kullanıp dosyayı `stat` komutuyla incelersek, dosyanın zamanlarının güncellendiğini görürüz.

```
[me@linuxbox ~]$ touch playground/timestamp
[me@linuxbox ~]$ stat playground/timestamp
  Dosya: `playground/timestamp'
  Boyut: 0          Bloklar: 0       IO Bloğu: 4096 normal boş dosya
Aygıt: 803h/2051d Inode: 14265061 Bağlantılar: 1
 Erişim:  2025-10-08   15:23:33.0000000   -0400   Gid: ( 1001/ me)
Değiştirm 08.10.2025       15:23:33       -0400
   e:
Değişikli 2025-10-08   15:23:33.00000000   -0400
   k:
```

Şimdi, `find komutunu` kullanarak oyun alanı dosyalarımızın bazılarını güncelleyelim.

```
[me@linuxbox ~]$ find playground -type f -name 'file-B' -exec touch
'{}' ';'
```

Bu komut, playground içindeki `file-B` adlı tüm dosyaları günceller. Ardından, `find` komutunu kullanarak tüm dosyaları referans dosyanın `zaman damgasıyla` karşılaştırarak güncellenen dosyaları belirleyeceğiz.

```
[me@linuxbox ~]$ find playground -type f -newer playground/timestamp
```

Sonuçlar, `file-B adlı dosyanın` 100 örneğinin tümünü içerir. `Zaman damgasını` güncelledikten sonra playground içindeki `file-B` adlı tüm dosyalara `touch` komutunu uyguladığımız için, bu dosyalar artık `zaman damgasından` "daha yeni"dir ve bu nedenle `-newer` testi ile tanımlanabilir.

Son olarak, daha önce yaptığımız hatalı izinler testine geri dönelim ve bunu `playground'a` uygulayalım.

```
[me@linuxbox ~]$ find playground \( -type f -not -perm 0600 \) -or \(
-type d -not -perm 0700 \)
```

Bu komut, `playground` içindeki 100 dizin ve 2.600 dosyayı (`zaman damgası` ve `playground'un` kendisiyle birlikte toplam 2.702) listeler, çünkü bunların hiçbiri "iyi izinler" tanımımıza uymaz. Operatörler ve eylemler hakkındaki bilgilerimizle, bu komuta eylemler ekleyerek playground'umuzdaki dosya ve dizinlere yeni izinler uygulayabiliriz.

```
[me@linuxbox ~]$ find playground \( -type f -not -perm 0600 -exec
chmod 0600 '{}' ';' \) -or \( -type d -not -perm 0700 -exec chmod
0700 '{}' ';' \)
```

Günlük kullanımda, bu büyük bileşik komut yerine, biri dizinler ve diğeri dosyalar için olmak üzere iki komut vermek daha kolay olabilir, ancak bu şekilde de yapabileceğimizi bilmek güzel. Burada önemli olan nokta, operatörlerin ve eylemlerin yararlı görevleri gerçekleştirmek için nasıl birlikte kullanılabileceğini anlamaktır.

Seçenekler

Son olarak, seçenekler var. Seçenekler, `find` aramasının kapsamını kontrol etmek için kullanılır. `Find` ifadeleri oluşturulurken diğer testler ve eylemlerle birlikte kullanılabilirler. Tablo 17-7, en sık kullanılan `find` seçeneklerini listeler.

Tablo 17-7: `find` Seçenekleri

Seçenek	Açıklama

`-derinlik`	Dizinin kendisini işlemden önce dizindeki dosyaları işlemek için doğrudan `arama`. Bu seçenek, `-delete` eylemi belirtildiğinde otomatik olarak uygulanır.
`-maxdepth` *düzeyleri*	Testler ve eylemler gerçekleştirilirken `find'ın` bir dizin ağacında inebileceği maksimum seviye sayısını ayarlayın.
`-mindepth` *seviyeleri*	Testleri ve eylemleri uygulamadan önce `find'ın` bir dizin ağacına ineceği minimum seviye sayısını ayarlayın.
`-mount`	Diğer dosya sistemlerine bağlanmış dizinleri taramamasını `find'a` belirtin.
`-noleaf`	`Find'ın,` Unix benzeri bir dosya sisteminde arama yaptığı varsayımıyla aramayı optimize etmemesini belirtir. Bu, DOS/Windows dosya sistemlerini ve CD-ROM'ları tararken gereklidir.

Özet

`Locate'in` `find` kadar karmaşık olmadığı kolayca anlaşılabilir. Her ikisinin de kendine özgü kullanım alanları vardır. Zaman ayırıp `find'ın` birçok özelliğini keşfedin. Düzenli olarak kullanıldığında, Linux dosya sistemi işlemlerine ilişkin anlayışınızı geliştirebilir.

Daha Fazla Okuma

- `loc`ate, `updatedb`, `find` ve `xargs` programları GNU Projesi'nin *findutils* paketinin bir parçasıdır. GNU Projesi, kapsamlı çevrimiçi belgeler içeren bir web sitesi sunmaktadır. Bu belgeler oldukça iyidir ve bu programları yüksek güvenlikli ortamlarda kullanıyorsanız okunması gerekir. Ancak, çoğu modern Linux dağıtımının `locate` ve `updatedb` programlarını findutils paketinden kaldırdığını ve bunları genellikle ayrı bir pakette bulunan `plocate` gibi daha yeni ve daha iyi performans gösteren sürümlerle değiştirdiğini unutmayın: https://www.gnu.org/software/findutils/

18 – Arşivleme ve Yedekleme

Sistem yöneticisinin temel görevlerinden biri, sistem verilerinin güvenliğini sağlamaktır. Bunun bir yolu, sistem dosyalarının zamanında yedeklenmesidir. Sistem yöneticisi olmasak bile, genellikle bir şeylerin kopyalarını almak ve büyük dosya koleksiyonlarını bir yerden başka bir yere, bir cihazdan başka bir cihaza taşımak yararlıdır.

Bu bölümde, dosya koleksiyonlarını yönetmek için kullanılan birkaç yaygın programa bakacağız. Bunlar dosya sıkıştırma programlarıdır:

- `gzip` – Dosyaları sıkıştırır veya genişletir
- `bzip2` – Blok sıralama dosya sıkıştırıcı

Bunlar arşivleme programlarıdır:

- `tar` – Teyp arşivleme yardımcı programı
- `zip` – Dosyaları paketler ve

sıkıştırır Bu, dosya senkronizasyon

programıdır:

- `rsync` – Uzak dosya ve dizin senkronizasyonu

Dosyaları Sıkıştırma

Bilgisayar tarihinin başlangıcından beri, bellek, depolama aygıtları veya ağ bant genişliği gibi mevcut en küçük alana en fazla veriyi sığdırmak için çaba gösterilmiştir. Günümüzde cep telefonu hizmeti, yüksek çözünürlüklü televizyon veya geniş bant İnternet gibi birçok veri hizmeti, etkili *veri sıkıştırma* teknikleri sayesinde var olabilmektedir.

Veri sıkıştırma, verilerden *gereksiz bilgileri* kaldırma işlemidir. Hayali bir örnek ele alalım. 100 piksel x 100 piksel boyutlarında tamamen siyah bir resim dosyamız olduğunu varsayalım. Veri depolama açısından (piksel başına 24 bit veya 3 bayt olduğunu varsayarsak), bu resim 30.000 bayt depolama alanı kaplayacaktır.

100 * 100 * 3 = 30.000

Tek renkten oluşan bir görüntü tamamen gereksiz veriler içerir. Akıllı davranırsak, verileri, 10.000 siyah pikselden oluşan bir blok

Bu nedenle, 30.000 sıfır içeren bir veri bloğunu depolamak yerine (siyah genellikle görüntü dosyalarında sıfır olarak temsil edilir), verileri 10.000 sayısına sıkıştırabilir ve ardından verilerimizi temsil etmek için bir sıfır ekleyebiliriz. Bu tür bir veri sıkıştırma şeması, *çalıştırma uzunluğu kodlaması olarak* adlandırılır ve en temel sıkıştırma tekniklerinden biridir. Günümüzün teknikleri çok daha gelişmiş ve karmaşıktır, ancak temel amaç aynıdır: gereksiz verilerden kurtulmak.

Sıkıştırma algoritmaları (sıkıştırmayı gerçekleştirmek için kullanılan matematiksel teknikler) genel olarak iki kategoriye ayrılır.

- *Kayıpsız*: Kayıpsız sıkıştırma, orijinalde bulunan tüm verileri korur. Bu, bir dosya sıkıştırılmış bir sürümden geri yüklendiğinde, geri yüklenen dosyanın orijinal, sıkıştırılmamış sürümle aynı olduğu anlamına gelir.
- *Kayıplı*: Kayıplı sıkıştırma ise, daha fazla sıkıştırma uygulanabilmesi için sıkıştırma işlemi sırasında verileri kaldırır. Kayıplı bir dosya geri yüklendiğinde, orijinal sürümle aynı olmaz; daha çok, orijinaline yakın bir yaklaşımdır. Kayıplı sıkıştırma örnekleri arasında JPEG (görüntüler için) ve MP3 (müzik için) bulunur.

Tartışmamızda, bilgisayarlardaki çoğu veri herhangi bir veri kaybını tolere edemeyeceğinden, yalnızca kayıpsız sıkıştırmaya odaklanacağız.

gzip

`gzip` programı bir veya daha fazla dosyayı sıkıştırmak için kullanılır. Çalıştırıldığında, orijinal dosyayı sıkıştırılmış bir versiyonuyla değiştirir. Karşılık gelen `gunzip` programı, sıkıştırılmış dosyaları orijinal, sıkıştırılmamış hallerine geri döndürmek için kullanılır. İşte bir örnek:

```
[me@linuxbox ~]$ ls -l /etc > foo.txt
[me@linuxbox ~]$ ls -l foo.*
-rw-r--r-- 1 me     me    15738 2025-10-14 07:15 foo.txt
[me@linuxbox ~]$ gzip foo.txt
[me@linuxbox ~]$ ls -l foo.*
-rw-r--r-- 1 ben    ben    3230 2025-10-14 07:15 foo.txt.gz
[me@linuxbox ~]$ gunzip foo.txt
[me@linuxbox ~]$ ls -l foo.*
-rw-r--r-- 1 ben    ben    15738 2025-10-14 07:15 foo.txt
```

Bu örnekte, bir dizin listesinden `foo.txt` adlı bir metin dosyası oluşturuyoruz. Ardından, `gzip` komutunu çalıştırarak orijinal dosyayı `foo.tx-t.gz` adlı sıkıştırılmış bir sürümle değiştiriyoruz. `foo.*` dizin listesinde, orijinal dosyanın sıkıştırılmış sürümle değiştirildiğini ve sıkıştırılmış sürümün orijinal dosyanın yaklaşık beşte biri kadar olduğunu görüyoruz. Ayrıca, sıkıştırılmış dosyanın orijinal dosya ile aynı izinlere ve zaman damgasına sahip olduğunu da görebiliriz.

aynı izinlere ve zaman damgasına sahip olduğunu da görebiliriz.

Ardından, dosyayı sıkıştırmayı kaldırmak için `gunzip` programını çalıştırıyoruz. Sonrasında, dosyanın sıkıştırılmış versiyonunun orijinal ile değiştirildiğini ve yine izinlerin ve zaman damgasının korunduğunu görebiliriz.

`gzip'in` Tablo 18-1'de açıklanan birçok seçeneği vardır.

Tablo 18-1: `gzip` Seçenekleri

Seçenek	Uzun Seçenek	Açıklama
`-c`	`--stdout` `--to-stdout`	Çıktıyı standart çıktıya yazın ve orijinal dosyaları koruyun.
`-d`	`--decompress` `--uncompress`	Sıkıştırmayı açar. Bu, `gzip`'in `gunzip` gibi davranmasına neden olur.
`-f`	`--force`	Orijinal dosyanın sıkıştırılmış bir sürümü zaten mevcut olsa bile sıkıştırmayı zorla. sürümü mevcut olsa bile sıkıştırmayı zorlar.
`-h`	`--help`	Kullanım bilgilerini gösterir.
`-l`	`--list`	Her bir dosya için sıkıştırma istatistiklerini listeler. .
`-r`	`--recursive`	Komut satırındaki bir veya daha fazla argüman bir dizin ise, içerdikleri dosyaları içerdiği dosyaları yinelemeli olarak sıkıştırır.
`-t`	`--test`	Sıkıştırılmış bir dosyanın bütünlüğünü test eder.
`-v`	`--verbose`	Sıkıştırma sırasında ayrıntılı mesajları görüntüler.
`-sayı`		Sıkıştırma miktarını ayarla. *number*, 1 (en hızlı, en az sıkıştırma) ile 9 (en yavaş, en sıkıştırma) aralığında bir tamsayıdır. 1 ve 9 değerleri `--fast` ve `--best` olarak da ifade edilebilir ifade edilebilir. Varsayılan değer 6'dır.

Önceki örneğimize geri dönelim.

```
[me@linuxbox ~]$ gzip foo.txt
[me@linuxbox ~]$ gzip -tv foo.txt.gz
foo.txt.gz:    OK
```

`bzip2, .bz2` uzantısıyla gösterilir.

```
[me@linuxbox ~]$ ls -l /etc > foo.txt
[me@linuxbox ~]$ ls -l foo.txt
-rw-r--r-- 1 me     me    15738 2025-10-17 13:51 foo.txt
[me@linuxbox ~]$ bzip2 foo.txt
[me@linuxbox ~]$ ls -l foo.txt.bz2
-rw-r--r-- 1 ben    ben     2792 2025-10-17 13:51 foo.txt.bz2
[me@linuxbox ~]$ bunzip2 foo.txt.bz2
```

Gördüğümüz gibi, `bzip2 gzip` ile aynı şekilde kullanılabilir. `Gzip` için ele aldığımız tüm seçenekler (`- r` hariç) `bzip2'de` de desteklenmektedir. Ancak, sıkıştırma düzeyi seçeneğinin (*`-number`*) `bzip2` için biraz farklı bir anlamı olduğunu unutmayın. `bzip2,` dosyaları açmak için `bunzip2` ve `bzcat` ile birlikte gelir.

`bzip2` ayrıca, hasarlı `.bz2` dosyalarını kurtarmaya çalışan `bzip2recover` programı ile birlikte gelir.

Sıkıştırma Takıntısı Yapmayın

Bazen, etkili bir sıkıştırma algoritmasıyla zaten sıkıştırılmış bir dosyayı şu şekilde sıkıştırmaya çalışan insanlar görüyorum:

```
$ gzip picture.jpg
```

Bunu yapmayın. Muhtemelen sadece zaman ve alan israf ediyorsunuz! Zaten sıkıştırılmış bir dosyayı sıkıştırırsanız, genellikle daha büyük bir dosya elde edersiniz. Bunun nedeni, tüm sıkıştırma tekniklerinin, sıkıştırmayı tanımlamak için dosyaya eklenen bir miktar ek yük içermesidir. Zaten gereksiz bilgi içermeyen bir dosyayı sıkıştırmaya çalışırsanız, sıkıştırma çoğu zaman ek yükü telafi edecek herhangi bir tasarruf sağlamaz.

Dosyaları Arşivleme

Sıkıştırma ile birlikte sıklıkla kullanılan yaygın bir dosya yönetimi görevi *arşivlemedir*. Arşivleme, birçok dosyayı toplayıp tek bir büyük dosyada bir araya getirme işlemidir. Arşivleme genellikle sistem yedeklemelerinin bir parçası olarak yapılır. Ayrıca eski veriler bir sistemden bir tür uzun vadeli depolama alanına taşındığında da kullanılır.

tar

Unix benzeri yazılım dünyasında, `tar` programı dosyaları arşivlemek için kullanılan klasik bir araçtır. Adı, *tape archive* (*bant arşivi*) kelimesinin kısaltması olan tar, yedekleme bantları oluşturmak için kullanılan bir araç olarak kökenini ortaya koymaktadır. Hala bu geleneksel görev için kullanılsa da, diğer depolama aygıtlarında da aynı derecede yeteneklidir. Sık sık `.tar` veya `.tgz` uzantısıyla biten dosya adları görürüz; bunlar sırasıyla "düz" bir tar arşivi ve gzip ile sıkıştırılmış bir arşivi belirtir. Bir tar arşivi, bir grup ayrı dosyadan, bir veya daha fazla dizin hiyerarşisinden veya her ikisinin karışımından oluşabilir. Komut sözdizimi şu şekilde çalışır:

`tar` *modu*`[`*seçenekler*`]` *yoladı*`...`

Burada *mod*, Tablo 18-2'de listelenen aşağıdaki çalışma modlarından biridir (burada sadece kısmi bir liste gösterilmektedir; tam liste için `tar` man sayfasına bakın).

Tablo 18-2: `tar` Modları

Mod	Açıklama
`c`	Dosya ve/veya dizin listesinden bir arşiv oluşturur.
`x`	Arşivi çıkarın.
`r`	Belirtilen yol adlarını arşivin sonuna ekler. Arşiv mevcut değilse, oluşturulur.
`t`	Arşivin içeriğini listeler.

`tar`, seçenekleri ifade etmek için biraz garip bir yol kullanır, bu yüzden nasıl çalıştığını göstermek için bazı örneklere ihtiyacımız olacak. Öncelikle, önceki bölümdeki oyun alanımızı yeniden oluşturalım.

```
[me@linuxbox ~]$ mkdir -p playground/dir-{001..100} [me@linuxbox
~]$ touch playground/dir-{001..100}/file-{A..Z}
```

Şimdi, tüm oyun alanının bir tar arşivi oluşturalım.

```
[me@linuxbox ~]$ tar cf playground.tar playground
```

Bu komut, `playground.tar` adlı ve tüm playground dizin hiyerarşisini içeren bir tar arşivi oluşturur. Tar arşivinin adını belirtmek için kullanılan mod ve `f` seçeneğinin birleştirilebileceğini ve başlangıçta tire işareti gerekmediğini görebiliriz. Ancak, modun her zaman seçeneklerden önce belirtilmesi gerektiğini unutmayın.

Arşivin içeriğini listelemek için şunu yapabiliriz:

```
[me@linuxbox ~]$ tar tf playground.tar
```

Daha ayrıntılı bir liste için `v` (verbose) seçeneğini ekleyebiliriz.

```
[me@linuxbox ~]$ tar tvf playground.tar
```

Şimdi, playground'u yeni bir konuma çıkaralım. Bunu, `foo` adında yeni bir dizin oluşturarak, dizini değiştirerek ve tar arşivini çıkararak yapacağız.

```
[me@linuxbox ~]$ mkdir foo
[me@linuxbox ~]$ cd foo
[me@linuxbox foo]$ tar xf ../playground.tar
[me@linuxbox foo]$ ls
playground
```

`~/foo/playground` içeriğini incelersek, arşivin başarıyla kurulduğunu ve orijinal dosyaların tam bir kopyasının oluşturulduğunu görürüz. Ancak bir uyarı var. Süper kullanıcı olarak çalışmıyorsak, arşivlerden çıkarılan dosyalar ve dizinler orijinal sahibinin değil, geri yüklemeyi gerçekleştiren kullanıcının mülkiyetine geçer.

`tar'ın` bir başka ilginç özelliği de arşivlerdeki yol adlarını işleme şeklidir. Yol adları varsayılan olarak mutlak değil, görecelidir. `tar` bunu, arşivi oluştururken yol adından baştaki eğik çizgiyi kaldırarak yapar. Bunu göstermek için arşivimizi yeniden oluşturacağız, ancak bu sefer mutlak bir yol adı belirteceğiz.

```
[me@linuxbox foo]$ cd
[me@linuxbox ~]$ tar cf playground2.tar ~/playground
```

Unutmayın, `ENTER` tuşuna bastığımızda `~/playground`, `/home/me/playground` olarak genişleyecektir, böylece demomuz için mutlak bir yol adı elde edeceğiz. Ardından, daha önce olduğu gibi arşivi çıkaracağız ve ne olacağını izleyeceğiz.

```
[me@linuxbox ~]$ cd foo
[me@linuxbox foo]$ tar xf ../playground2.tar [me@linuxbox
foo]$ ls
home   playground
[me@linuxbox foo]$ ls home
```

```
me
[me@linuxbox foo]$ ls home/me oyun alanı
```

Burada, ikinci arşivimizi çıkardığımızda, mutlak yol adında olduğu gibi kök dizine göre değil, mevcut çalışma dizini ~/foo'ya göre home/me/playground dizinini yeniden oluşturduğunu görebiliriz. Bu, garip bir çalışma şekli gibi görünebilir, ancak aslında bu şekilde daha kullanışlıdır, çünkü arşivleri orijinal konumlarına çıkarmak zorunda kalmadan herhangi bir konuma çıkarabilmemizi sağlar. Verbose seçeneğini (v) ekleyerek alıştırmayı tekrarlamak, neler olup bittiğini daha net bir şekilde gösterecektir.

Tar'ın kullanımına ilişkin varsayımsal, ancak pratik bir örnek ele alalım. Bir sistemden diğerine ana dizini ve içeriğini kopyalamak istediğimizi ve aktarım için kullanabileceğimiz büyük bir USB sabit sürücümüz olduğunu varsayalım. Modern Linux sistemimizde, sürücü "otomatik olarak" /media dizinine bağlanır. Ayrıca, diski bağladığımızda diskin birim adının BigDisk olduğunu varsayalım. Tar arşivini oluşturmak için aşağıdakileri yapabiliriz:

```
[me@linuxbox ~]$ sudo tar cf /media/BigDisk/home.tar /home
```

Tar dosyası yazıldıktan sonra, sürücüyü söküp ikinci bilgisayara takıyoruz. Yine, /media/BigDisk dizinine takılıyor. Arşivi çıkarmak için şunu yapıyoruz:

```
[me@linuxbox2 ~]$ cd /
[me@linuxbox2 /]$ sudo tar xf /media/BigDisk/home.tar
```

Burada dikkat edilmesi gereken önemli nokta, arşivdeki tüm yol adları göreli olduğu için, çıkarmanın kök dizine göreli olması için önce dizini / olarak değiştirmemiz gerektiğidir.

Bir arşivi açarken, arşivden çıkarılanları sınırlamak mümkündür. Örneğin, bir arşivden tek bir dosyayı çıkarmak istersek, bunu şu şekilde yapabiliriz:

```
tar xf archive.tar yol adı
```

Komuta sonuna *yol adını* ekleyerek, tar yalnızca belirtilen dosyayı geri yükler. Birden fazla yol adı belirtilebilir. Yol adının, arşivde depolanan tam ve kesin göreceli yol adı olması gerektiğine dikkat edin. Yol adlarını belirtirken, joker karakterler normalde desteklenmez; ancak, GNU sürümü tar (Linux dağıtımlarında en sık bulunan sürüm

burada önceki `playground.tar` dosyamızı kullanan bir örnek verilmiştir:

```
[me@linuxbox ~]$ cd foo
[me@linuxbox foo]$ tar xf ../playground2.tar --wildcards 'home/me/pla
yground/dir-*/file-A'
```

Bu komut, `dir-*` joker karakterini içeren belirtilen yol adıyla eşleşen dosyaları çıkaracaktır.

`tar` genellikle `find` ile birlikte arşiv oluşturmak için kullanılır. Bu örnekte, `find komutunu` kullanarak arşive eklenecek bir dizi dosya oluşturacağız.

```
[me@linuxbox ~]$ find playground -name 'file-A' -exec tar rf
playground.tar '{}' '+'
```

Burada `find komutunu` kullanarak `playground` içinde `file-A` adlı tüm dosyaları eşleştiriyoruz ve ardından `-exec` eylemini kullanarak `tar komutunu` ekleme modunda (`r`) çalıştırarak eşleşen dosyaları `playground.tar` arşivine ekliyoruz.

`Tar komutunu find` ile birlikte kullanmak, bir dizin ağacının veya tüm sistemin *artımlı yedeklemelerini* oluşturmak için iyi bir yoldur. `Find komutunu` kullanarak zaman damgası dosyasından daha yeni dosyaları eşleştirerek, her yedekleme oluşturulduktan hemen sonra zaman damgası dosyasının güncellendiğini varsayarak, yalnızca son yedeklemeden daha yeni dosyaları içeren bir arşiv oluşturabiliriz.

tar, hem standart girişi hem de çıkışı kullanabilir. İşte kapsamlı bir örnek:

```
[me@linuxbox foo]$ cd
[me@linuxbox ~]$ find playground -name 'file-A' | tar cf - --files-
from=- | gzip > playground.tgz
```

Bu örnekte, eşleşen dosyaların bir listesini oluşturmak için `find` programını kullandık ve bunları `tar'a` aktardık. `-` dosya adı belirtilirse, gerektiğinde standart giriş veya çıkış anlamına gelir. (Bu arada, standart giriş/çıkışı temsil etmek için `-` kullanma kuralı, diğer birçok programda da kullanılır). `--files-from` seçeneği (aynı zamanda `-T` olarak da belirtilebilir), `tar`'ın yol adları listesini komut satırından değil bir dosyadan okumasını sağlar. Son olarak, `tar` tarafından oluşturulan arşiv `gzip'e` aktarılır ve sıkıştırılmış `playground.tgz` arşivi oluşturulur. `.tgz` uzantısı, gzip ile sıkıştırılmış tar dosyalarına verilen geleneksel uzantıdır. `.tar.gz` uzantısı da bazen kullanılır.

.

Yukarıdaki örnekte sıkıştırılmış arşivimizi oluşturmak için gzip programını harici olarak kullanmış olsak da, GNU tar'ın modern sürümleri sırasıyla z ve j seçeneklerini kullanarak hem gzip hem de bzip2 sıkıştırmasını doğrudan destekler. Önceki örneğimizi temel alarak, bunu şu şekilde basitleştirebiliriz:

```
[me@linuxbox ~]$ find playground -name 'file-A' | tar czf
playground.tgz -T -
```

Bunun yerine bzip2 ile sıkıştırılmış bir arşiv oluşturmak isteseydik, şunu yapabilirdik:

```
[me@linuxbox ~]$ find playground -name 'file-A' | tar cjf
playground.tbz -T -
```

Sıkıştırma seçeneğini z'den j'ye değiştirerek (ve çıktı dosyasının uzantısını bzip2 ile sıkıştırılmış tar dosyasını belirtmek için .tbz olarak değiştirerek) bzip2 sıkıştırmasını etkinleştirdik.

Tar komutuyla standart giriş ve çıkışın bir başka ilginç kullanımı, ağ üzerinden sistemler arasında dosya aktarımıdır. Tar ve ssh ile donatılmış Unix benzeri bir sistem çalıştıran iki makinemiz olduğunu düşünün. Böyle bir senaryoda, uzak bir sistemden (bu örnekte remote-sys olarak adlandırılmıştır) yerel sistemimize bir dizin aktarabiliriz.

```
[me@linuxbox ~]$ mkdir remote-stuff
[me@linuxbox ~]$ cd remote-stuff
[me@linuxbox remote-stuff]$ ssh remote-sys 'tar cf - Documents' | tar xf
-
me@remote-sys'in şifresi:
[me@linuxbox remote-stuff]$ ls
Documents
```

Burada, uzak sistem re-mote-sys'teki Documents adlı bir dizini yerel sistemdeki remote-stuff adlı dizinin içine kopyalayabildik. Bunu nasıl yaptık? İlk olarak, ssh kullanarak uzak sistemde tar programını başlattık. Unutmayın, ssh, ağa bağlı bir bilgisayarda bir programı uzaktan çalıştırmamızı ve sonuçları yerel sistemde "görmemizi" sağlar — uzak sistemde üretilen standart çıktı, görüntülenmek üzere yerel sisteme gönderilir. Bundan yararlanarak

tar'ın bir arşiv oluşturmasını (c modu) ve bunu bir dosyaya (tire argümanlı f seçeneği) değil, standart çıktıya göndermesini sağlayarak, arşivi ssh tarafından sağlanan şifreli tünel üzerinden yerel sisteme aktarabiliriz. Yerel sistemde, tar'ı çalıştırır ve standart girişten (yine, tire argümanlı f seçeneği) sağlanan bir arşivi (x modu) genişletiriz.

zip

Zip programı hem bir sıkıştırma aracı hem de bir arşivleyicidir. Programın kullandığı dosya biçimi, .zip dosyalarını okuyup yazdığı için Windows kullanıcılarına aşinadır. Ancak Linux'ta, gzip en yaygın kullanılan sıkıştırma programıdır ve bzip2 ikinci sıradadır.

En temel kullanımında, zip şu şekilde çağrılır:

zip *seçenekleri zip dosyası dosya...*

Örneğin, oyun alanımızın zip arşivini oluşturmak için şunu yaparız:

```
[me@linuxbox ~]$ zip -r playground.zip playground
```

Yineleme için -r seçeneğini eklemezsek, yalnızca oyun alanı dizini (içeriği hariç) depolanır. .zip uzantısının eklenmesi otomatik olsa da, netlik için dosya uzantısını ekleyeceğiz.

Zip arşivi oluşturulurken, zip normalde aşağıdaki gibi bir dizi mesaj görüntüler:

```
ekliyor: playground/dir-020/file-Z (saklanan
%0) ekliyor: playground/dir-020/file-Y
(saklanan %0) ekliyor: playground/dir-020/file-
X (saklanan %0) ekliyor: playground/dir-087/
(saklanan %0) ekliyor: playground/dir-087/file-
S (%0 depolandı)
```

Bu mesajlar, arşive eklenen her dosyanın durumunu gösterir. zip, iki depolama yönteminden birini kullanarak dosyaları arşive ekler: ya burada gösterildiği gibi dosyayı sıkıştırmadan "depolar" ya da dosyayı "sıkıştırarak" sıkıştırır. Depolama yönteminden sonra görüntülenen sayısal değer, elde edilen sıkıştırma miktarını gösterir. Oyun alanımız yalnızca boş dosyalar içerdiğinden, içeriği üzerinde sıkıştırma işlemi gerçekleştirilmez.

Unzip programı kullanıldığında zip dosyasının içeriğini çıkarmak çok kolaydır.

```
[me@linuxbox ~]$ cd foo
[me@linuxbox foo]$ unzip ../playground.zip
```

Zip hakkında dikkat edilmesi gereken bir nokta (tar'ın aksine), mevcut bir arşiv belirtildiğinde, bu arşivin değiştirilmeyip güncellenmesidir. Bu, mevcut arşivin korunacağı, ancak yeni dosyaların ekleneceği ve eşleşen dosyaların değiştirileceği anlamına gelir.

Dosyalar, zip arşivinden seçici olarak listelenebilir ve çıkarılabilir. Bunun için dosyaları unzip komutuyla seçilerek seçici olarak listelenebilir ve çıkarılabilir.

```
[me@linuxbox ~]$ unzip -l playground.zip playground/dir-087/file-Z
Arşiv: ../playground.zip
  Uzunluk     Tarih   Saat     Ad
 ---------    -----   -----    -----
        0 10-05-25 09:25     oyun alanı/dir-087/dosya-Z
 ---------                    --------
        0                    1
 [me@linuxbox ~]$ cd foo      dosya
[me@linuxbox foo]$ unzip ../playground.zip playground/dir-087/file-Z
Arşiv: ../playground.zip
playground/dir-087/file-Z dosyasını değiştirmek ister misiniz? [y]evet,
[n]hayır, [A]tümünü, [N]hiçbiri, [r]yeniden adlandır: y
 çıkarılıyor: playground/dir-087/file-Z
```

-l seçeneği kullanıldığında, unzip dosyayı açmadan yalnızca arşivin içeriğini listeler. Hiçbir dosya belirtilmezse, unzip arşivdeki tüm dosyaları listeler. -v seçeneği eklenerek listenin ayrıntı düzeyi artırılabilir. Arşivin açılması mevcut bir dosyayla çakıştığında, dosya değiştirilmeden önce kullanıcıya sorulur.

Varsayılan olarak, unzip dosyayı geçerli dizine çıkarır. Çıkarma için alternatif bir dizin belirtmek için, -d seçeneğini ve ardından istenen dizinin adını kullanın.

tar gibi, zip de standart giriş ve çıkışı kullanabilir, ancak uygulaması biraz daha az kullanışlıdır. -@ seçeneği ile bir dosya adı listesini zip'e aktarmak mümkündür.

```
[me@linuxbox foo]$ cd
[me@linuxbox ~]$ find playground -name "file-A" | zip -@ file-A.zip
```

Burada find komutunu kullanarak test -name "file-A" ile eşleşen dosyaların bir listesini oluşturuyoruz ve ardından bu listeyi zip komutuna aktarıyoruz. Bu komut, seçilen dosyaları içeren file-A.zip adlı bir arşiv dosyası oluşturur.

zip, çıktısını standart çıktıya yazmayı da destekler, ancak bu çıktıdan yararlanabilen program sayısı az olduğu için kullanımı sınırlıdır. Ne yazık ki, unzip programı standart girişi kabul etmez. Bu durum, zip ve unzip'in tar gibi ağ dosya kopyalama işlemlerinde birlikte kullanılamasına engel olur.

Ancak zip, standart girişi kabul edebilir, bu nedenle diğer programların çıktısını sıkıştırmak için kullanılabilir.

```
[me@linuxbox ~]$ ls -l /etc/ | zip ls-etc.zip -
 ekleme: - (%80 oranında küçültülmüş)
```

Bu örnekte, ls komutunun çıktısını zip komutuna aktarıyoruz. Tar komutu gibi, zip komutu da sondaki tireyi "dosya listesi için standart girişi kullan" olarak yorumlar. Zip komutu, sıkıştırılmış dosyayı (zip arşivi içindeki dosya) "-" dosyası olarak kaydedecektir. Bu, bir dosya için çok kötü bir isimdir; birçok komut satırı programı bu isimden hoşnut olmayacaktır.

Bu sorunu çözmek için, -p (boru için) seçeneği belirtildiğinde çıktısının standart çıktıya gönderilmesine izin veren unzip programını kullanın.

```
[me@linuxbox ~]$ unzip -p ls-etc.zip | less
```

Zip/unzip programlarının yapabileceği bazı temel görevlere değindik. Her ikisi de esnekliğini artıran birçok seçeneğe sahiptir, ancak bazıları diğer sistemlere özeldir. Zip ve unzip programlarının man sayfaları oldukça iyidir ve yararlı örnekler içerir. Ancak, bu programların ana kullanımı, tar ve gzip'in çok tercih edildiği Linux'ta sıkıştırma ve arşivleme yapmak değil, Windows sistemleriyle dosya alışverişi yapmaktır.

Dosya ve Dizinleri Senkronize Etme

Bir sistemin yedek kopyasını korumak için yaygın olarak kullanılan bir strateji, bir veya daha fazla dizini yerel sistemde (genellikle bir tür çıkarılabilir depolama aygıtı) veya uzak bir sistemde bulunan başka bir dizinle (veya dizinlerle) senkronize tutmaktır. Örneğin, geliştirilmekte olan bir web sitesinin yerel bir kopyasına sahip olabilir ve bunu zaman zaman uzak bir web sunucusundaki "canlı" kopya ile senkronize edebiliriz.

Unix benzeri dünyada, bu görev için tercih edilen araç rsync'tir. Bu program, *rsync uzak güncelleme protokolünü* kullanarak hem yerel hem de uzak dizinleri senkronize edebilir. Bu protokol, rsync'in iki dizin arasındaki farkları hızlı bir şekilde algılamasını ve bunları senkronize etmek için gereken minimum miktarda kopyalama işlemini gerçekleştirmesini sağlar. Bu, rsync'i diğer kopyalama programlarına kıyasla çok hızlı ve ekonomik hale getirir.

`rsync` şu şekilde çağrılır:

`rsync` *`seçenekleri kaynak hedef`*

burada *kaynak* ve *hedef* aşağıdakilerden biridir:

- Yerel bir dosya veya dizin
- *[kullanıcı@]host:yol* biçiminde bir uzak dosya veya dizin
- *rsync://[kullanıcı@]host[:port]/yol* URI'si ile belirtilen uzak bir rsync sunucusu

Kaynak veya hedefin yerel bir dosya olması gerektiğine dikkat edin. Uzak-uzak kopyalama desteklenmez.

Bazı yerel dosyalarda `rsync`'i deneyelim. Öncelikle, `foo` dizinimizi temizleyelim.

```
[me@linuxbox ~]$ rm -rf foo/*
```

Ardından, `playground` dizinini `foo` dizinindeki karşılık gelen kopyayla senkronize edeceğiz.

```
[me@linuxbox ~]$ rsync -av playground foo
```

`Foo` içindeki `playground` dizinini yansıtmak için hem `-a` seçeneğini (arşivleme için — yineleme ve dosya özniteliklerinin korunmasına neden olur) hem de `-v` seçeneğini (ayrıntılı çıktı) ekledik. Komut çalışırken, kopyalanan dosya ve dizinlerin bir listesini göreceğiz. Sonunda, gerçekleştirilen kopyalama miktarını gösteren aşağıdaki gibi bir özet mesajı göreceğiz:

```
135759 bayt gönderildi 57870 bayt alındı 387258,00 bayt/sn toplam
boyut 3230 hız artışı 0,02
```

Komutu tekrar çalıştırırsak, farklı bir sonuç göreceğiz.

```
[me@linuxbox ~]$ rsync -av playground foo dosya
listesi oluşturuluyor ... tamam

22635 bayt gönderildi 20 bayt alındı 45310,00 bayt/sn
toplam boyut 3230 hız artışı 0,14
```

Dosya listesi olmadığını fark edin. Bunun nedeni, `rsync`'in ~/playground ile ~/foo/playground arasında

fark olmadığını algıladığı ve bu nedenle hiçbir şey kopyalamasına gerek olmadığı içindir. Playground içindeki bir dosyayı değiştirip rsync'i tekrar çalıştırırsak:

```
[me@linuxbox ~]$ touch playground/dir-099/file-Z
[me@linuxbox ~]$ rsync -av playground foo dosya
listesi oluşturuluyor ... tamam
playground/dir-099/file-Z
22685 bayt gönderildi 42 bayt alındı 45454.00 bayt/sn
toplam boyut 3230 hız artışı 0.14
```

rsync'in değişikliği algıladığını ve yalnızca güncellenen dosyayı kopyaladığını görüyoruz.

rsync kaynağını belirlerken kullanabileceğimiz ince ama kullanışlı bir özellik vardır. Şu iki dizini ele alalım.

```
[me@linuxbox ~]$ ls source
                 destination
```

source dizini file1 adlı bir dosya içerir ve destination dizini boştur. source'u destination'a şu şekilde kopyalarsak:

```
[me@linuxbox ~]$ rsync kaynak hedef
```

rsync, source dizinini destination dizinine kopyalar.

```
[me@linuxbox ~]$ ls hedef kaynak
```

Ancak, kaynak dizin adının sonuna / eklediğimizde, rsync sadece kaynak dizinin içeriğini kopyalar, dizinin kendisini kopyalamaz.

```
[me@linuxbox ~]$ rsync kaynak/ hedef
[me@linuxbox ~]$ ls hedef
file1
```

Bu, hedef içinde başka bir dizin düzeyi oluşturmadan yalnızca bir dizinin içeriğini kopyalamak istediğimizde kullanışlıdır. Sonucunu source/* gibi düşünebiliriz

sonuç olarak düşünebiliriz, ancak bu yöntem gizli dosyalar da dahil olmak üzere kaynak dizinin tüm içeriğini kopyalar.

Pratik bir örnek olarak, daha önce `tar` ile kullandığımız hayali harici sabit sürücüyü ele alalım. Sürücüyü sistemimize bağlarsak ve yine `/media/BigDisk` konumuna takılırsa, önce harici sürücüde `/backup` adlı bir dizin oluşturup ardından `rsync` kullanarak sistemimizdeki en önemli dosyaları harici sürücüye kopyalayarak kullanışlı bir sistem yedeklemesi gerçekleştirebiliriz.

```
[me@linuxbox ~]$ mkdir /media/BigDisk/backup
[me@linuxbox ~]$ sudo rsync -av --delete /etc /home /usr/local
/media/BigDisk/backup
```

Bu örnekte, sistemimizden `/etc`, `/home` ve `/usr/local` dizinlerini hayali depolama aygıtımıza kopyaladık. Kaynak aygıtta artık bulunmayan, yedekleme aygıtında bulunabilecek dosyaları silmek için `--delete` seçeneğini ekledik (bu, ilk yedekleme yaparken önemsizdir, ancak sonraki kopyalamalarda yararlı olacaktır). Harici sürücüyü takma ve bu `rsync` komutunu çalıştırma prosedürünü tekrarlamak, küçük bir sistemi yedeklemek için yararlı (ancak ideal olmayan) bir yol olacaktır. Elbette, burada bir takma ad da yardımcı olabilir. Bu özelliği sağlamak için bir takma ad oluşturabilir ve bunu `.bashrc` dosyasına ekleyerek bu özelliği sağlayabiliriz.

```
alias backup='sudo rsync -av --delete /etc /home /usr/local
/media/BigDisk/backup'
```

Şimdi tek yapmamız gereken harici sürücümüzü takmak ve `yedekleme` komutunu çalıştırmak.

Ağ Üzerinden `rsync` Kullanımı

`rsync'in` en güzel özelliklerinden biri, ağ üzerinden dosya kopyalamak için kullanılabilmesidir. Sonuçta, `rsync'deki r harfi` "remote" (uzak) anlamına gelir. Uzaktan kopyalama iki yoldan biriyle yapılabilir. İlk yol, `rsync'in` yüklü olduğu başka bir sistem ve `ssh` gibi bir uzak kabuk programı kullanmaktır. Yerel ağımızda çok fazla kullanılabilir sabit disk alanı olan başka bir sistem olduğunu ve yedekleme işlemini harici sürücü yerine uzak sistemi kullanarak gerçekleştirmek istediğimizi varsayalım. Dosyalarımızı gönderebileceğimiz `/backup` adlı bir dizin olduğunu varsayarsak, bunu şu şekilde yapabiliriz:

```
[me@linuxbox ~]$ sudo rsync -av --delete --rsh=ssh /etc /home
```

```
/usr/local remote-sys:/backup
```

Ağ kopyalamayı kolaylaştırmak için komutumuzda iki değişiklik yaptık. İlk olarak, `--rsh=ssh` seçeneğini ekledik. Bu seçenek, `rsync'e ssh` programını uzak kabuk olarak kullanmasını söyler. Bu şekilde, ssh ile şifrelenmiş bir tünel kullanarak verileri yerel sistemden uzak ana bilgisayara güvenli bir şekilde aktarabildik. Modern sistemlerde `ssh` varsayılan olarak kullanılır, bu nedenle bu seçeneğe nadiren ihtiyaç duyulur. İkinci olarak, hedef yol adına uzaktaki ana bilgisayarın adını (bu durumda uzaktaki ana bilgisayarın adı `remote-sys`) ekleyerek uzaktaki ana bilgisayarı belirttik.

`rsync'in` bir ağ üzerinden dosyaları senkronize etmek için kullanılabileceği ikinci yol, bir *rsync sunucusu* kullanmaktır. `rsync,` bir arka plan programı olarak çalışacak ve gelen senkronizasyon isteklerini dinleyecek şekilde yapılandırılabilir. Bu genellikle uzak bir sistemin yansıtılmasına izin vermek için yapılır. Örneğin, Red Hat Software, Fedora dağıtımı için geliştirilmekte olan yazılım paketlerinin büyük bir deposunu tutmaktadır. Yazılım testçileri için, dağıtım sürüm döngüsünün test aşamasında bu koleksiyonu yansıtmak yararlıdır. Depodaki dosyalar sık sık (genellikle günde birden fazla kez) değiştiğinden, depoyu toplu olarak kopyalamak yerine, periyodik senkronizasyonla yerel bir yansıtma tutmak tercih edilir. Bu depolardan biri Duke Üniversitesi'nde tutulmaktadır; yerel `rsync` kopyamızı ve onların `rsync` sunucusunu kullanarak bunu şu şekilde yansıtabiliriz:

```
[me@linuxbox ~]$ mkdir fedora-devel
[me@linuxbox ~]$ rsync -av –delete rsync://archive.linux.duke.edu/
fedora/linux/development/rawhide/Everything/x86_64/os/ fedora-devel
```

Bu örnekte, uzak `rsync` sunucusunun URI'sini kullanıyoruz. Bu URI, bir protokol (`rsync://`), ardından uzak ana bilgisayar adı (`archive.linux.duke.edu`) ve ardından deponun yol adından oluşur.

Özet

Linux ve diğer Unix benzeri işletim sistemlerinde kullanılan yaygın sıkıştırma ve arşivleme programlarını inceledik. Dosyaları arşivlemek için, `tar/gzip` kombinasyonu Unix benzeri sistemlerde tercih edilen yöntemdir, `zip/unzip` ise Windows sistemleriyle birlikte çalışabilirlik için kullanılır. Son olarak, sistemler arasında dosya ve dizinlerin verimli bir şekilde senkronize edilmesi için çok kullanışlı olan `rsync` programını (kişisel favorim) inceledik.

Daha Fazla Okuma

- Burada ele alınan tüm komutların man sayfaları oldukça açıktır ve yararlı örnekler içerir. Ayrıca, GNU Projesi'nin kendi tar sürümü için iyi bir çevrimiçi kılavuzu vardır.

burada bulabilirsiniz: https://www.gnu.org/software/tar/manual/index.html

19 – Düzenli İfadeler

Önümüzdeki birkaç bölümde, metinleri işlemek için kullanılan araçlara bakacağız. Gördüğümüz gibi, metin verileri Linux gibi tüm Unix benzeri sistemlerde önemli bir rol oynar. Ancak bu araçların sunduğu tüm özellikleri tam olarak anlayabilmemiz için, önce bu araçların en sofistike kullanımlarıyla sıklıkla ilişkilendirilen bir teknolojiyi incelememiz gerekir: *düzenli ifadeler*.

Komut satırının sunduğu birçok özelliği ve olanakları incelerken, vi editöründen bahsetmeye gerek bile yok, kabuk genişletme ve alıntı yapma, klavye kısayolları ve komut geçmişi gibi gerçekten gizemli kabuk özellikleri ve komutlarıyla karşılaştık. Düzenli ifadeler bu "geleneği" sürdürür ve (tartışmalı olarak) bunların en gizemli özelliği olabilir. Bu, onları öğrenmek için harcanan zamanın çabaya değmeyeceği anlamına gelmez. Tam tersine. İyi bir anlayış, tam değerleri hemen anlaşılmasa da, şaşırtıcı başarılar elde etmemizi sağlayacaktır.

Düzenli İfadeler Nedir?

Basitçe ifade etmek gerekirse, düzenli ifadeler metindeki kalıpları tanımlamak için kullanılan sembolik notasyonlardır. Bazı yönlerden, kabuğun dosya ve yol adlarını eşleştirme joker karakter yöntemine benzerler, ancak çok daha büyük ölçekte. Düzenli ifadeler, metin işleme sorunlarının çözümünü kolaylaştırmak için birçok komut satırı aracı ve çoğu programlama dili tarafından desteklenir. Ancak, işleri daha da karmaşık hale getirmek için, tüm düzenli ifadeler aynı değildir; araçtan araca ve programlama dilinden dile biraz farklılık gösterirler. Tartışmamızda, biraz daha büyük ve zengin notasyon kümeleri kullanan birçok programlama dili (en önemlisi *Perl*) yerine, POSIX standardında açıklanan düzenli ifadelere (komut satırı araçlarının çoğunu kapsar) odaklanacağız.

grep

Düzenli ifadelerle çalışmak için kullanacağımız ana program, eski dostumuz `grep'tir`. "Grep" adı aslında "global regular expression print" (küresel düzenli ifade yazdırma) ifadesinden türemiştir, bu nedenle `grep'in` düzenli ifadelerle bir ilgisi olduğunu görebiliriz. Esasen, `grep` metin dosyalarında belirtilen düzenli ifadeyle eşleşen metinleri arar ve

eşleşen satırları standart çıktıya gönderir. Şimdiye kadar
`grep'i` sabit dizelerle şu şekilde kullandık:

```
[me@linuxbox ~]$ ls /usr/bin | grep zip
```

Bu, `/usr/bin` dizininde adı
`zip alt dizesini içeren` tüm dosyaları listeler.

`grep` programı seçenekleri ve argümanları şu şekilde kabul eder, burada *regex* bir düzenli ifadedir:

`grep [seçenekler] regex [dosya...]`

Tablo 19-1, sık kullanılan `grep` seçeneklerini açıklamaktadır.

Tablo 19-1: `grep` Seçenekleri

Seçenek	Uzun Seçenek	Açıklama
`-i`	`--ignore-case`	Büyük/küçük harfleri yok say. Büyük/küçük harfleri ayırt etme büyük ve küçük harf ayrım yapma.
`-v`	`--invert-match`	Eşleşmeyi ters çevir. Normalde, `grep` eşleşen satırları yazdırır. Bu seçenek, `grep`'in her satırı yazdırmasını sağlar. çalışır.
`-c`	`--count`	Eşleşenlerin sayısını yazdırır (veya eşleşmeyenlerin sayısını yazdırır (`-v` seçeneği ayrıca belirtildiği gibi) yerine kendileri yerine.
`-l`	`--files-with-matches`	Eşleşen her dosyanın adını yazdırır satırların kendileri yerine satırların kendisini yazdırır.
`-L`	`--files-without-match`	`-l` seçeneği gibi, ancak yalnızca eşleşmeyen dosyaların adlarını içeren dosyaların adlarını yazdırır.
`-n`	`--satır-numarası`	Eşleşen her satırın başına satır numarasını ekler.
`-h`	`--no-filename`	Birden fazla dosya içeren aramalarda,

		dosya adlarının çıktısını bastırın.
-q kullanışlıdır	--quiet, --silent	Tüm çıktıyı bastırın. Bu, aşağıdakiler için kabuk betiklerinde, bir eşleşme bulunup bulunmadığını test etmek istediğimizde kullanılır. Bir komutun *çıkış durumunu* test etmeyi 27. bölümde ele alacağız.

Grep'i daha ayrıntılı olarak incelemek için, arama yapmak üzere bazı metin dosyaları oluşturalım.

```
[me@linuxbox ~]$ ls /bin > dirlist-bin.txt [me@linuxbox
~]$ ls /usr/bin > dirlist-usr-bin.txt [me@linuxbox ~]$
ls /sbin > dirlist-sbin.txt [me@linuxbox ~]$ ls
/usr/sbin > dirlist-usr-sbin.txt [me@linuxbox ~]$ ls
dirlist*.txt
dirlist-bin.txt    dirlist-sbin.txt      dirlist-usr-sbin.txt
dirlist-usr-bin.txt
```

Dosya listemizde şu şekilde basit bir arama yapabiliriz:

```
[me@linuxbox ~]$ grep bzip dirlist*.txt
dirlist-bin.txt:bzip2
dirlist-bin.txt:bzip2recover
```

Bu örnekte, grep listelenen tüm dosyalarda bzip dizesini arar ve her ikisi de dirlist-bin.txt dosyasında bulunan iki eşleşme bulur. Eşleşmelerin kendisinden ziyade eşleşmeleri içeren dosyaların listesiyle ilgileniyorsak, -l seçeneğini belirtebiliriz.

```
[me@linuxbox ~]$ grep -l bzip dirlist*.txt
dirlist-bin.txt
```

Tersine, yalnızca eşleşmeyen dosyaların listesini görmek istersek, şunu yapabiliriz:

```
[me@linuxbox ~]$ grep -L bzip dirlist*.txt
dirlist-sbin.txt
```

```
dirlist-usr-bin.txt
dirlist-usr-sbin.txt
```

Meta karakterler ve Literaler

Belirgin görünmese de, `grep` aramalarımız başından beri düzenli ifadeler kullanmaktadır, ancak bunlar çok basittir. Düzenli ifade `bzip`, dosyadaki satır en az dört karakter içeriyorsa ve satırın herhangi bir yerinde `b`, `z`, `i` ve `p` karakterleri bu sırayla, aralarında başka karakter olmadan bulunuyorsa eşleşme olacağı anlamına gelir. `bzip` dizesindeki karakterlerin tümü, kendileriyle eşleşen *gerçek karakterlerdir*. Gerçek karakterlere ek olarak, düzenli ifadeler daha karmaşık eşleşmeleri belirtmek için kullanılan meta karakterleri de içerebilir. Düzenli ifade meta karakterleri şunlardan oluşur:

```
^ $ . [ ] { } - ? * + ( ) | \
```

Diğer tüm karakterler literal olarak kabul edilir, ancak ters eğik çizgi karakteri bazı durumlarda *meta diziler* oluşturmak için kullanılır ve meta karakterlerin kaçış karakteri olarak kullanılmasına ve meta karakter olarak yorumlanmak yerine literal olarak ele alınmasına olanak tanır.

Not: Görüldüğü gibi, düzenli ifade meta karakterlerinin çoğu, genişletme gerçekleştirildiğinde kabuk için anlamlı karakterlerdir. Komut satırında meta karakterler içeren düzenli ifadeleri geçtiğimizde, kabuğun bunları genişletmeye çalışmasını önlemek için bunların tırnak içine alınması çok önemlidir.

Herhangi Bir Karakter

İlk olarak inceleyeceğimiz meta karakter, herhangi bir karakterle eşleşmek için kullanılan nokta veya nokta karakteridir. Bunu düzenli ifadeye dahil edersek, o karakter konumundaki herhangi bir karakterle eşleşir. İşte bir örnek:

```
[me@linuxbox ~]$ grep -h '.zip' dirlist*.txt
bunzip2
bzip2
bzip2recover
gunzip
gzip
funzip
gpg-zip
```

```
preunzip
prezip
prezip-bin
unzip
unzipsfx
```

Dosyalarımızda ".zip" düzenli ifadesiyle eşleşen herhangi bir satır aradık. Sonuçlarla ilgili dikkat edilmesi gereken birkaç ilginç nokta var. Zip programının bulunamadığına dikkat edin. Bunun nedeni, düzenli ifademize nokta meta karakterinin eklenmesiyle gerekli eşleşmenin uzunluğunun dört karaktere çıkmasıdır; bu karakterlerden biri "z"den önce gelmelidir. Ayrıca, listelerimizdeki herhangi bir dosya `.zip` dosya uzantısını içeriyor olsaydı, bunlar da eşleşirdi, çünkü dosya uzantısındaki nokta karakteri de "herhangi bir karakter" ile eşleşirdi.

Bağlantı noktaları

Caret (`^`) ve dolar işareti (`$`) karakterleri, düzenli ifadelerde *çapa* olarak kabul edilir. Bu, düzenli ifadenin satırın başında (`^`) veya satırın sonunda (`$`) bulunması durumunda eşleşmenin gerçekleşmesi anlamına gelir.

```
[me@linuxbox ~]$ grep -h '^zip' dirlist*.txt zip
zipcloak
zipgrep
zipinfo
zipnote
zipsplit
[me@linuxbox ~]$ grep -h 'zip$' dirlist*.txt
gunzip
gzip
funzip
gpg-zip
preunzip
prezip
unzip
zip
[me@linuxbox ~]$ grep -h '^zip$' dirlist*.txt
zip
```

Burada, dosya listesinde satırın başında, satırın sonunda ve hem satırın başında hem de sonunda bulunan `zip` dizesini aradık

(yani satırda tek başına). `^$` düzenli ifadesinin (başlangıcı ve sonu arasında hiçbir şey olmayan) boş satırlarla eşleşeceğini unutmayın.

Çapraz Bulmaca Yardımcısı

Şu anda düzenli ifadeler hakkındaki bilgimiz sınırlı olsa da, yararlı bir şey yapabiliriz.

Eşim bulmaca yapmayı çok sever ve bazen bana belirli bir soru için yardım ister. Mesela, "Üçüncü harfi 'j' ve son harfi 'r' olan ve anlamı ... olan beş harfli kelime nedir?" gibi. Bu tür sorular beni düşündürdü.

Linux sisteminizde bir sözlük olduğunu biliyor muydunuz? Evet, var. /usr/share/dict dizinine bakın, bir veya birkaç tane bulabilirsiniz. Burada bulunan sözlük dosyaları, alfabetik sırayla düzenlenmiş, her satırda bir kelime bulunan uzun kelime listeleridir. Benim sistemimde, `kelime` dosyası 98.500'den fazla kelime içeriyor. Eşimin bulmaca sorusuna olası cevapları bulmak için şunu yapabiliriz:

```
[me@linuxbox ~]$ grep -i '^..j.r$' /usr/share/dict/words
Major
major
```

Bu düzenli ifadeyi kullanarak, sözlük dosyamızda beş harf uzunluğunda, üçüncü konumda `j` ve son konumda `r` harfi bulunan tüm kelimeleri bulabiliriz.

Köşeli Parantez İfadeleri ve Karakter Sınıfları

Düzenli ifademizde belirli bir konumdaki herhangi bir karakterle eşleşmenin yanı sıra, *köşeli parantez ifadeleri* kullanarak belirli bir karakter kümesinden tek bir karakterle de eşleşebiliriz. Köşeli parantez ifadeleriyle, eşleşecek bir karakter kümesi (aksi takdirde meta karakterler olarak yorumlanacak karakterler dahil) belirtebiliriz. Bu örnekte, iki karakterlik bir küme kullanarak, `bzip` veya `gzip` dizesini içeren herhangi bir satırla eşleşiriz:

```
[me@linuxbox ~]$ grep -h '[bg]zip' dirlist*.txt
bzip2
bzip2recover
gzip
```

Bir küme herhangi bir sayıda karakter içerebilir ve meta karakterler köşeli parantez içine alındığında özel anlamlarını yitirirler. Ancak, meta karakterlerin köşeli parantez ifadeleri içinde kullanıldığı ve farklı anlamlara sahip olduğu iki durum vardır. Birincisi, olumsuzluğu belirtmek için kullanılan caret

(^) olup, olumsuzluğu belirtmek için kullanılır; ikincisi ise karakter aralığını belirtmek için kullanılan tire (-) karakteridir.

Olumsuzluk

Köşeli parantez ifadesindeki ilk karakter caret (^) ise, kalan karakterler verilen karakter konumunda bulunmaması gereken bir karakter kümesi olarak kabul edilir. Bunu, önceki örneğimizi aşağıdaki gibi değiştirerek yapıyoruz:

```
[me@linuxbox ~]$ grep -h '[^bg]zip' dirlist*.txt
bunzip2
gunzip
funzip
gpg-zip
preunzip
prezip
prezip-bin
unzip
unzipsfx
```

Olumsuzlama etkinleştirildiğinde, `b` veya `g` harfleri dışında herhangi bir karakterin önünde `zip` dizesi bulunan dosyaların listesini alırız. `zip` dosyasının bulunamadığına dikkat edin. Olumsuzlanmış bir karakter kümesi, yine de verilen konumda bir karakter gerektirir, ancak bu karakter olumsuzlanmış kümenin bir üyesi olmamalıdır.

Caret karakteri, yalnızca köşeli parantez içindeki ifadenin ilk karakteri ise olumsuzlamayı çağırır; aksi takdirde, özel anlamını yitirir ve kümedeki sıradan bir karakter haline gelir.

Geleneksel Karakter Aralıkları

Listelerimizdeki büyük harfle başlayan tüm dosyaları bulacak bir düzenli ifade oluşturmak istersek, bunu şu şekilde yapabiliriz:

```
[me@linuxbox ~]$ grep -h '^[ABCDEFGHIJKLMNOPQRSTUVWXYZ]' dirlist*.txt
```

Bu, 26 büyük harfin tümünü bir parantez ifadesine koymakla ilgili bir meseledir. Ancak tüm bunları yazmak çok zahmetli bir iştir, bu yüzden başka bir yol daha vardır.

```
[me@linuxbox ~]$ grep -h '^[A-Z]' dirlist*.txt
```

```
MAKEDEV
Kontrol Paneli
GET
HEAD
POST
X
X11
Xorg
MAKEFLOPPIES
Ağ Yöneticisi Ağ Yöneticisi
Dağıtıcı
```

Üç karakterlik bir aralık kullanarak 26 harfi kısaltabiliriz. Herhangi bir karakter aralığı bu şekilde ifade edilebilir, harf ve rakamlarla başlayan tüm dosya adlarını eşleştiren bu ifade gibi birden fazla aralık da dahil olmak üzere:

```
[me@linuxbox ~]$ grep -h '^[A-Za-z0-9]' dirlist*.txt
```

Karakter aralıklarında, tire karakterinin özel olarak ele alındığını görüyoruz, peki tire karakterini köşeli parantez ifadesine nasıl dahil edebiliriz? İfadenin ilk karakteri yaparak. Şu iki örneği ele alalım:

```
[me@linuxbox ~]$ grep -h '[A-Z]' dirlist*.txt
```

Bu, büyük harf içeren tüm dosya adlarıyla eşleşecektir. Aşağıdaki komut, tire, büyük harf `A` veya büyük harf `Z` içeren tüm dosya adlarıyla eşleşecektir:

```
[me@linuxbox ~]$ grep -h '[-AZ]' dirlist*.txt
```

POSIX Karakter Sınıfları

Geleneksel karakter aralıkları, karakter kümelerini hızlı bir şekilde belirtme sorununu ele almak için kolay anlaşılır ve etkili bir yoldur. Ne yazık ki, bunlar her zaman işe yaramaz. Şimdiye kadar `grep` kullanımında herhangi bir sorunla karşılaşmadık, ancak diğer programları kullanırken sorunlarla karşılaşabiliriz.

Bölüm 4'te, yol adı genişletme işlemini gerçekleştirmek için joker karakterlerin nasıl kullanıldığını inceledik. Bu tartışmada, karakter aralıklarının düzenli ifadelerde kullanıldıkları şekilde neredeyse aynı şekilde kullanılabileceğini söylemiştik, ancak sorun şu:

```
[me@linuxbox ~]$ ls /usr/sbin/[ABCDEFGHIJKLMNOPQRSTUVWXYZ]*
/usr/sbin/MAKEFLOPPIES
/usr/sbin/NetworkManagerDispatcher
/usr/sbin/NetworkManager
```

(Linux dağıtımına bağlı olarak, farklı bir dosya listesi, muhtemelen boş bir liste elde ederiz. Bu örnek Ubuntu'dan alınmıştır). Bu komut beklenen sonucu verir — adları büyük harfle başlayan dosyaların listesi, ancak bu komutla tamamen farklı bir sonuç elde ederiz (sonuçların sadece bir kısmı gösterilmektedir):

```
[me@linuxbox ~]$ ls /usr/sbin/[A-Z]*
/usr/sbin/biosdecode
/usr/sbin/chat
/usr/sbin/chgpasswd
/usr/sbin/chpasswd
/usr/sbin/chroot
/usr/sbin/cleanup-info
/usr/sbin/complain
/usr/sbin/console-kit-daemon
```

Neden böyle? Uzun bir hikaye, ama kısaca şöyle:

Unix ilk geliştirildiğinde, sadece ASCII karakterlerini tanıyordu ve bu özellik bu gerçeği yansıtıyor. ASCII'da, ilk 32 karakter (0-31 arası sayılar) kontrol kodlarıdır (sekme, geri alma ve satır başı gibi). Sonraki 32 karakter (32-63) yazdırılabilir karakterleri içerir; bunlar arasında çoğu noktalama işareti ve 0-9 arası sayılar bulunur. Sonraki 32 karakter (64-95 arası sayılar) büyük harfleri ve birkaç noktalama işaretini içerir. Son 31 karakter (96-126 arası sayılar) küçük harfleri ve daha fazla noktalama işaretini içerir. Bu düzenlemeye göre, ASCII kullanan sistemler şu şekilde bir *sıralama düzeni* kullanır:

`ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz` Bu, şu

şekilde olan doğru sözlük sırasından farklıdır:

`aAbBcCdDeEfFgGhHiIjJkKlLmMnNoOpPqQrRsStTuUvVwWxXyYzZ`

Unix'in popülaritesi Amerika Birleşik Devletleri'nin ötesine yayıldıkça, ABD İngilizcesinde bulunmayan karakterleri destekleme ihtiyacı da arttı. ASCII tablosu, sekiz bitin tamamını kullanacak şekilde genişletildi ve 128-255 karakterleri eklendi, böylece çok daha fazla dil desteklenebilir hale geldi. Bu özelliği desteklemek için POSIX standartları, belirli bir konum için gerekli karakter kümesini seçmek üzere ayarlanabilen "*yerel ayar*" adlı bir kavramı getirdi. Bu komutu kullanarak sistemimizin dil ayarını görebiliriz:

```
[me@linuxbox ~]$ echo $LANG
en_US.UTF-8
```

Bu ayar ile POSIX uyumlu uygulamalar ASCII sırası yerine sözlük sıralama sırasını kullanacaktır. Bu, önceki komutların davranışını açıklar. Sözlük sırasına göre yorumlandığında `[A-Z]` karakter aralığı, küçük harf `a` hariç tüm alfabetik karakterleri içerir, bu nedenle sonuçlarımız böyledir.

Bu sorunu kısmen çözmek için POSIX standardı, Tablo 19-2'de açıklanan yararlı karakter aralıkları sağlayan bir dizi karakter sınıfı içerir.

Tablo 19-2: POSIX Karakter Sınıfları

Karakter Sınıfı	Açıklama
`[:alnum:]`	Alfasayısal karakterler. ASCII'de şuna eşdeğerdir: `[A-Za-z0-9]`
`[:word:]`	[:alnum:] ile aynıdır, ancak alt çizgi (_) karakteri eklenmiştir.
`[:alpha:]`	Alfabetik karakterler. ASCII'de şuna eşdeğerdir: `[A-Za-z]`
`[:boşluk:]`	Boşluk ve sekme karakterlerini içerir.
`[:cntrl:]`	ASCII kontrol kodları. ASCII karakterleri 0 ile 31 ve 127'yi içerir.
`[:digit:]`	0 ile 9 arasındaki rakamlar.
`[:graph:]`	Görünür karakterler. ASCII'da 33 ile 126 arasındaki karakterleri içerir.
`[:lower:]`	Küçük harfler.
`[:punct:]`	Noktalama karakterleri. ASCII'de şunlara eşdeğerdir: `[-!"#$%&'()*+,./:;<=>?@[\\\]_`{|}~]`
`[:print:]`	Yazdırılabilir karakterler. `[:graph:]` içindeki tüm karakterler artı boşluk karakteri.
`[:boşluk:]`	Boşluk, sekme, satır başı, satır sonu, dikey sekme ve form besleme dahil olmak üzere boşluk karakterleri. ASCII'de şunlara eşdeğerdir: `[ \t\r\n\v\f]`
`[:upper:]`	Büyük harfler.

`[:xdigit:]`	Onaltılık sayıları ifade etmek için kullanılan karakterler. ASCII'da şunlara eşdeğerdir: `[0-9A-Fa-f]`

Karakter sınıfları olsa bile, [A-M] gibi kısmi aralıkları ifade etmenin hala uygun bir yolu yoktur.

Karakter sınıflarını kullanarak, dizin listemizi tekrarlayabilir ve daha iyi bir sonuç elde edebiliriz.

```
[me@linuxbox ~]$ ls /usr/sbin/[[:upper:]]*
/usr/sbin/MAKEFLOPPIES
/usr/sbin/NetworkManagerDispatcher
/usr/sbin/NetworkManager
```

Ancak bunun bir düzenli ifade örneği olmadığını, bunun yerine kabuğun yol adı genişletmesini gerçekleştirdiğini unutmayın. POSIX karakter sınıfları her ikisi için de kullanılabileceğinden bunu burada gösteriyoruz.

Geleneksel Karşılaştırma Sırasına Geri Dönme

`LANG` ortam değişkeninin değerini değiştirerek sisteminizin geleneksel (ASCII) sıralama düzenini kullanmasını sağlayabilirsiniz. Daha önce gördüğümüz gibi, `LANG` değişkeni yerel ayarlarınızda kullanılan dil ve karakter kümesinin adını içerir. Bu değer, Linux sürümünüz yüklendiğinde kurulum dilini seçtiğinizde belirlenmiştir.

Yerel ayarları görmek için `locale` komutunu kullanın.

```
[me@linuxbox ~]$ locale
LANG=en_US.UTF-8
LC_CTYPE="en_US.UTF-8"
LC_NUMERIC="en_US.UTF-8"
LC_TIME="en_US.UTF-8"
LC_COLLATE="en_US.UTF-8"
LC_MONETARY="en_US.UTF-8"
LC_MESSAGES="en_US.UTF-8"
LC_PAPER="en_US.UTF-8"
LC_NAME="en_US.UTF-8"
LC_ADDRESS="en_US.UTF-8"
```

```
LC_TELEPHONE="en_US.UTF-8"
LC_MEASUREMENT="en_US.UTF-8"
LC_IDENTIFICATION="en_US.UTF-8"
LC_ALL=
```

Yerel ayarları geleneksel Unix davranışlarını kullanacak şekilde değiştirmek için `LANG` değişkenini `POSIX` olarak ayarlayın.

```
[me@linuxbox ~]$ export LANG=POSIX
```

Bu değişiklik, sistemin karakter seti için ABD İngilizcesini (daha spesifik olarak ASCII) kullanmasına neden olur, bu nedenle gerçekten istediğinizin bu olduğundan emin olun.

Bu değişikliği kalıcı hale getirmek için `.bashrc` dosyanıza şu satırı ekleyin.

```
export LANG=POSIX
```

POSIX Temel ve Genişletilmiş Düzenli İfadeler

Bu konunun daha fazla kafa karıştırıcı olamayacağını düşündüğümüz anda, POSIX'in düzenli ifade uygulamalarını da iki türe ayırdığını keşfettik: *temel düzenli ifadeler (BRE)* ve *genişletilmiş düzenli ifadeler (ERE)*. Şimdiye kadar ele aldığımız özellikler, POSIX uyumlu ve BRE uygulayan tüm uygulamalar tarafından desteklenmektedir. `Grep` programımız da bu tür programlardan biridir.

BRE ve ERE arasındaki fark nedir? Bu, meta karakterlerle ilgilidir. BRE ile aşağıdaki meta karakterler tanınır:

```
^ $ . [ ] *
```

Diğer tüm karakterler, sabit karakterler olarak kabul edilir. ERE ile, aşağıdaki meta karakterler (ve bunlarla ilişkili işlevler) eklenir:

```
( ) { } ? + |
```

Ancak (ve bu da işin eğlenceli kısmı), `(`, `)`, `{` ve `}` karakterleri, ters eğik çizgi ile kaçırılırsa BRE'de meta karakter olarak değerlendirilirken, ERE'de herhangi bir meta karakterin önüne ters eğik çizgi konulursa, bu karakter literal olarak değerlendirilir. Karşılaşılan tüm tuhaflıklar, sonraki tartışmalarda ele alınacaktır.

Bundan sonra ele alacağımız özellikler ERE'nin bir parçası olduğu için, farklı bir `grep` kullanmamız gerekecek. Geleneksel olarak bu işlem `egrep` programı ile gerçekleştirilir, ancak GNU sürümündeki `grep`, `-E` seçeneği kullanıldığında genişletilmiş düzenli ifadeleri de destekler.

POSIX

1980'lerde Unix çok popüler bir ticari işletim sistemi haline geldi, ancak 1988'de Unix dünyası kargaşa içindeydi. Birçok bilgisayar üreticisi, Unix kaynak kodunu yaratıcısı AT&T'den lisanslamış ve sistemleriyle birlikte işletim sisteminin çeşitli sürümlerini tedarik ediyordu. Ancak, ürün farklılaştırma çabaları kapsamında, her üretici kendi özel değişikliklerini ve uzantılarını ekledi. Bu durum, yazılımın uyumluluğunu sınırlamaya başladı. Her zaman olduğu gibi, tescilli satıcılar, müşterileriyle "kilitleme" oyununu kazanmaya çalışıyordu. Unix tarihindeki bu karanlık dönem, bugün "*Balkanizasyon*" olarak bilinir.

Elektrik ve Elektronik Mühendisleri Enstitüsü (IEEE) devreye girdi. 1980'lerin ortalarında, IEEE, Unix (ve Unix benzeri) sistemlerin nasıl çalışacağını tanımlayacak bir dizi standart geliştirmeye başladı. Resmi olarak IEEE 1003 olarak bilinen bu standartlar, standart bir Unix benzeri sistemde bulunacak *uygulama programlama arayüzlerini* (API'ler), kabuğu ve yardımcı programları tanımlar. *Portable Operating System Interface* (Taşınabilir İşletim Sistemi Arayüzü) anlamına gelen POSIX adı (sonuna ekstra canlılık katmak için X eklenmiştir), Richard Stallman (evet, *o* Richard Stallman) tarafından önerilmiş ve IEEE tarafından kabul edilmiştir.

Alternatif

Tartışacağımız genişletilmiş düzenli ifade özelliklerinin ilki, bir dizi ifade arasından eşleşme yapılmasına olanak tanıyan *alternatif* olarak adlandırılır. Köşeli parantez ifadesi, belirli bir karakter kümesinden tek bir karakterin eşleşmesine izin verdiği gibi, alternatif de bir dizi dize veya diğer düzenli ifadelerden eşleşmelere izin verir.

Bunu göstermek için `grep'i echo` ile birlikte kullanacağız. Öncelikle, eski bir dize eşleşmesini deneyelim.

```
[me@linuxbox ~]$ echo "AAA" | grep AAA
AAA
[me@linuxbox ~]$ echo "BBB" | grep AAA
[me@linuxbox ~]$
```

Bu oldukça basit bir örnektir. Burada `echo` komutunun çıktısını `grep` komutuna aktarıp sonuçları görüyoruz. Eşleşme olduğunda, sonuçlar ekrana yazdırılır; eşleşme olmadığında ise sonuç görülmez.

Şimdi, dikey çubuk meta karakteriyle gösterilen alternatif ekleyeceğiz.

```
[me@linuxbox ~]$ echo "AAA" | grep -E 'AAA|BBB' AAA
[me@linuxbox ~]$ echo "BBB" | grep -E 'AAA|BBB' BBB
[me@linuxbox ~]$ echo "CCC" | grep -E 'AAA|BBB'
[me@linuxbox ~]$
```

Burada, "AAA veya BBB dizesiyle eşleş" anlamına gelen 'AAA|BBB' düzenli ifadesini görüyoruz. Bu, genişletilmiş bir özellik olduğu için grep komutuna -E seçeneğini eklediğimizi (bunun yerine egrep programını da kullanabilirdik) ve kabuğun dikey çizgi meta karakterini boru operatörü olarak yorumlamasını önlemek için düzenli ifadeyi tırnak içine aldığımızı unutmayın. Alternatifler iki seçenekle sınırlı değildir.

```
[me@linuxbox ~]$ echo "AAA" | grep -E 'AAA|BBB|CCC' AAA
```

Alternatifleri diğer düzenli ifade öğeleriyle birleştirmek için, () kullanarak alternatifleri ayırabiliriz.

```
[me@linuxbox ~]$ grep -Eh '^(bz|gz|zip)' dirlist*.txt
```

Bu ifade, listelerimizdeki bz, gz veya zip ile başlayan dosya adlarıyla eşleşecektir. Parantezleri çıkarsaydık, bu düzenli ifadenin anlamı bz ile başlayan *veya* gz *veya* zip *içeren* herhangi bir dosya adıyla eşleşecek şekilde değişirdi:

```
[me@linuxbox ~]$ grep -Eh '^bz|gz|zip' dirlist*.txt
```

Nicelik belirteçleri

Genişletilmiş düzenli ifadeler, aşağıdaki bölümlerde açıklandığı gibi, bir öğenin eşleşeceği sayıyı belirtmek için çeşitli yöntemler destekler.

? - Bir öğeyi sıfır veya bir kez eşleştir

Bu nicelik belirteci, aslında "Önceki öğeyi isteğe bağlı hale getir" anlamına gelir. Bir telefon numarasının geçerliliğini kontrol etmek istediğimizi ve *n* bir sayı olmak üzere, bu iki formdan herhangi biriyle eşleşen telefon numaralarını geçerli kabul ettiğimizi varsayalım:

`(nnn) nnn-nnnn`

`nnn nnn-nnnn`

Böyle bir düzenli ifade oluşturabiliriz:

```
^\(?[0-9][0-9][0-9]\)? [0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]$
```

Bu ifadede, parantez karakterlerinin ardından soru işaretleri ekleyerek bunların sıfır veya bir kez eşleşmesi gerektiğini belirtiriz. Yine, parantezler normalde (ERE'de) meta karakterler olduğundan, bunların yerine gerçek karakterler olarak değerlendirilmeleri için önlerine ters eğik çizgi ekleriz.

Deneyelim.

```
[me@linuxbox ~]$ echo "(555) 123-4567" | grep -E '^\(?[0-9][0-9][0-9]
\)? [0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]$'
(555) 123-4567
[me@linuxbox ~]$ echo "555 123-4567" | grep -E '^\(?[0-9][0-9][0-9]\)
? [0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]$'
555 123-4567
[me@linuxbox ~]$ echo "AAA 123-4567" | grep -E '^\(?[0-9][0-9][0-9]\)
? [0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]$'
[me@linuxbox ~]$
```

Burada, ifadenin telefon numarasının her iki biçimiyle de eşleştiğini, ancak sayısal olmayan karakterler içeren biçimle eşleşmediğini görüyoruz. Bu ifade, alan kodunun etrafındaki eşleşmeyen parantezlere hala izin verdiği için mükemmel değildir, ancak doğrulamanın ilk aşamasını gerçekleştirecektir.

* - Bir öğeyi sıfır veya daha fazla kez eşleştir

`?` meta karakteri gibi, `*` de isteğe bağlı bir öğeyi belirtmek için kullanılır; ancak `?` 'den farklı olarak, öğe sadece bir kez değil, herhangi bir sayıda tekrar edilebilir. Bir dizenin cümle olup olmadığını görmek istediğimizi varsayalım; yani, büyük harfle başlıyor, ardından herhangi bir sayıda büyük ve küçük harf ve boşluk içeriyor ve nokta ile bitiyor. Bu (kaba) cümle tanımına uymak için, aşağıdaki gibi bir düzenli ifade kullanabiliriz:

```
^[[:upper:]][[:upper:][:lower:] ]*\.
```

İfade üç öğeden oluşur: `[:upper:]` karakter sınıfını içeren bir köşeli parantez ifadesi, `[:upper:]` ve `[:lower:]` karakter sınıflarını ve bir boşluğu içeren bir köşeli parantez ifadesi ve ters eğik çizgi ile kaçış karakteri eklenmiş bir nokta. İkinci öğenin arkasında `*` meta karakteri bulunur, böylece cümlemizin başındaki büyük harfin ardından herhangi bir sayıda büyük ve küçük harf ile boşluk gelebilir ve yine de eşleşebilir.

```
[me@linuxbox ~]$ echo "Bu çalışıyor." | grep -E '^[[:upper:]][[:upper:]
[:lower:] ]*\.'
Bu çalışıyor.
[me@linuxbox ~]$ echo "Bu çalışıyor." | grep -E '^[[:upper:]][[:upper:]
[:lower:] ]*\.'
Bu çalışıyor.
[me@linuxbox ~]$ echo "bu çalışmıyor" | grep -E '^[[:upper:]][[:upp
er:][:lower:] ]*\.'
[me@linuxbox ~]$
```

İfade ilk iki testi eşleştirir, ancak üçüncü testi eşleştirmez, çünkü gerekli olan baştaki büyük harf ve sondaki nokta eksiktir.

+ - Bir öğeyi bir veya daha fazla kez eşleştir

`+` meta karakteri, `*` karakteri ile çok benzer şekilde çalışır, ancak eşleşmesi için önündeki öğenin en az bir kez geçmesi gerekir. Aşağıda, tek boşluklarla ayrılmış bir veya daha fazla alfabetik karakterden oluşan satırlarla eşleşecek bir düzenli ifade verilmiştir:

```
^([[:alpha:]]+ ?)+$
```

```
[me@linuxbox ~]$ echo "This that" | grep -E '^([[:alpha:]]+ ?)+$'
This that
[me@linuxbox ~]$ echo "a b c" | grep -E '^([[:alpha:]]+ ?)+$' a
b c
[me@linuxbox ~]$ echo "a b 9" | grep -E '^([[:alpha:]]+ ?)+$'
[me@linuxbox ~]$ echo "abc d" | grep -E '^([[:alpha:]]+ ?)+$'
[me@linuxbox ~]$
```

Bu ifadenin, alfabetik olmayan bir karakter içerdiği için `a b 9` satırıyla eşleşmediğini görüyoruz; ayrıca, `c` ve `d` karakterlerini ayıran birden fazla boşluk karakteri olduğu için `abc d` ile de eşleşmiyor.

{ } - Bir öğeyi belirli sayıda eşleştirme

`{` ve `}` meta karakterleri, gerekli eşleşmelerin minimum ve maksimum sayısını ifade etmek için kullanılır. Tablo 19-3'te özetlendiği gibi dört farklı şekilde belirtilebilirler.

Tablo 19-3: Eşleşmelerin Sayısını Belirtme

Belirleyici	Anlam

	Nicelik belirteçleri
{*n*}	Önceki öğe tam olarak *n* kez tekrar ediyorsa eşleştirin.
{*n*,*m*}	Önceki öğe en az *n* kez ama en fazla *m* kez tekrar ediyorsa eşleştir.
{*n*,}	Önceki öğe *n* veya daha fazla kez tekrar ediyorsa eşleştir.
{,*m*}	Önceki öğe *m* defadan fazla tekrarlamıyorsa eşleşir.

Telefon numaralarıyla ilgili önceki örneğimize geri dönersek, tekrarları belirtme yöntemini kullanarak aşağıdaki orijinal düzenli ifadeyi basitleştirebiliriz:

```
^\(?[0-9][0-9][0-9]\)? [0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]$
```

şeklinde basitleştirebiliriz:

```
^\(?[0-9]{3}\)? [0-9]{3}-[0-9]{4}$
```

Deneyelim.

```
[me@linuxbox ~]$ echo "(555) 123-4567" | grep -E '^\(?[0-9]{3}\)? [0-
9]{3}-[0-9]{4}$'
(555) 123-4567
[me@linuxbox ~]$ echo "555 123-4567" | grep -E '^\(?[0-9]{3}\)? [0-9]
{3}-[0-9]{4}$'
555 123-4567
[me@linuxbox ~]$ echo "5555 123-4567" | grep -E '^\(?[0-9]{3}\)? [0-9
]{3}-[0-9]{4}$'
[me@linuxbox ~]$
```

Gördüğümüz gibi, revize ettiğimiz ifade, parantezli ve parantezsiz sayıları başarıyla doğrularken, doğru biçimde biçimlendirilmemiş sayıları reddediyor.

Düzenli İfadeleri Kullanmaya Başlayalım

Zaten bildiğimiz bazı komutlara bakalım ve bunların düzenli ifadelerle nasıl kullanılabileceğini görelim.

`grep` ile Telefon Listesi Doğrulama

Önceki örneğimizde, tekil telefon numaralarını inceledik ve doğru biçimlendirilip biçimlendirilmediklerini kontrol ettik. Daha gerçekçi bir senaryo, bir numaralar listesini kontrol etmek olacaktır, bu yüzden bir liste oluşturalım. Bunu, komut satırına sihirli bir büyü okuma ile yapacağız. Sihirli olacak çünkü ilgili komutların çoğunu henüz ele almadık, ancak endişelenmeyin. Gelecek bölümlerde bu konuyu ele alacağız. İşte büyü:

```
[me@linuxbox ~]$ for i in {1..10}; do echo "(${RANDOM:0:3}) ${RANDO
M:0:3}-${RANDOM:0:4}" >> phonelist.txt; done
```

Bu komut, 10 telefon numarası içeren `phonelist.txt` adlı bir dosya oluşturacaktır. Komut her tekrarlandığında, listeye 10 numara daha eklenir. Komutun başındaki `10` değerini değiştirerek daha fazla veya daha az telefon numarası oluşturabiliriz. Ancak dosyanın içeriğini incelediğimizde bir sorun olduğunu görürüz.

```
[me@linuxbox ~]$ cat phonelist.txt
(232)  298-2265
(624)  381-1078
(540)  126-1980
(874)  163-2885
(286)  254-2860
(292)  108-518
(129)  44-1379
(458)  273-1642
(686)  299-8268
(198)  307-2440
```

Bazı numaralar hatalı biçimlendirilmiştir, bu da bizim amacımıza mükemmel bir şekilde uygundur, çünkü `grep` kullanacağız.

Doğrulama için yararlı bir yöntem, dosyayı geçersiz numaralar için taramak ve sonuç listesini görüntülemek olabilir.

```
[me@linuxbox ~]$ grep -Ev '^\([0-9]{3}\) [0-9]{3}-[0-9]{4}$' phonelist.txt
(292) 108-518
(129) 44-1379
[me@linuxbox ~]$
```

Burada, `-v` seçeneğini kullanarak ters eşleşme oluşturuyoruz, böylece listede belirtilen ifadeyle eşleşmeyen satırları çıktı olarak alıyoruz. İfadenin her iki ucunda, sayının her iki ucunda da fazladan karakter bulunmadığından emin olmak için bağlantı meta karakterleri bulunuyor. Bu ifade, önceki telefon numarası örneğimizden farklı olarak, parantezlerin geçerli bir sayı içinde bulunmasını da gerektiriyor.

find ile Çirkin Dosya Adlarını Bulma

find komutu, düzenli ifadeye dayalı bir testi destekler. find ve grep komutlarında düzenli ifade kullanırken akılda tutulması gereken önemli bir husus vardır. grep, bir *satırda* ifadeyle eşleşen bir dize *varsa* o satırı yazdırırken, find komutu yol adının düzenli ifadeyle *tam olarak eşleşmesini* gerektirir. Aşağıdaki örnekte, düzenli ifadeyle find komutunu kullanarak aşağıdaki kümeye dahil olmayan herhangi bir karakter içeren tüm yol adlarını bulacağız:

```
[-_./0-9a-zA-Z]
```

Böyle bir tarama, gömülü boşluklar ve diğer potansiyel olarak rahatsız edici karakterler içeren yol adlarını ortaya çıkaracaktır.

```
[me@linuxbox ~]$ find . -regex '.*[^-_./0-9a-zA-Z].*'
```

Yol adının tamamının tam olarak eşleşmesi gerektiğinden, ifadenin her iki ucunda .* kullanarak sıfır veya daha fazla karakter örneğini eşleştiriyoruz. İfadenin ortasında, kabul edilebilir yol adı karakterleri kümesini içeren ters köşeli parantez ifadesi kullanıyoruz.

Locate ile Dosya Arama

Locate programı hem temel (--regexp seçeneği) hem de genişletilmiş (--regex seçeneği) düzenli ifadeleri destekler. Bununla, daha önce dirlist dosyalarımızla gerçekleştirdiğimiz birçok işlemi gerçekleştirebiliriz.

```
[me@linuxbox ~]$ locate --regex 'bin/(bz|gz|zip)'
/bin/bzcat
/bin/bzcmp
/bin/bzdiff
/bin/bzegrep
/bin/bzexe
/bin/bzfgrep
/bin/bzgrep
/bin/bzip2
/bin/bzip2recover
/bin/bzless
/bin/bzmore
/bin/gzexe
/bin/gzip
/usr/bin/zip
```

```
/usr/bin/zipcloak
/usr/bin/zipgrep
/usr/bin/zipinfo
/usr/bin/zipnote
/usr/bin/zipsplit
```

Alternatif kullanarak, `bin/bz`, `bin/gz` veya `/bin/zip` içeren yol adlarını ararız.

`less` ve `vim` ile Metin Arama

`less` ve `vim`, metin arama konusunda aynı yöntemi kullanır. `/` tuşuna basıp ardından bir düzenli ifade girerek arama yapabilirsiniz. `phonelist.txt` dosyasını `less` ile görüntülemek için şu komutu kullanabiliriz:

```
[me@linuxbox ~]$ less phonelist.txt
```

ve doğrulama ifademizi şu şekilde ararsak:

```
(232) 298-2265
(624) 381-1078
(540) 126-1980
(874) 163-2885
(286) 254-2860
(292) 108-518
(129) 44-1379
(458) 273-1642
(686) 299-8268
(198) 307-2440
~
~
~
/^\([0-9]{3}\) [0-9]{3}-[0-9]{4}$
```

daha sonra `less`, eşleşen dizeleri vurgular ve geçersiz olanları kolayca tespit edilebilir hale getirir.

```
(232) 298-2265
(624) 381-1078
(540) 126-1980
```

```
(874) 163-2885
(286) 254-2860
(292) 108-518
(129) 44-1379
(458) 273-1642
(686) 299-8268
(198) 307-2440
~
~
~
(SON)
```

`vim` ise temel düzenli ifadeleri destekler, bu nedenle arama ifademiz şöyle görünür:

```
/([0-9]\{3\}) [0-9]\{3\}-[0-9]\{4\}
```

İfadenin çoğunlukla aynı olduğunu görebiliriz; ancak, genişletilmiş ifadelerde meta karakter olarak kabul edilen birçok karakter, temel ifadelerde literal olarak kabul edilir. Yalnızca ters eğik çizgi ile kaçırıldıklarında meta karakter olarak kabul edilirler. Sistemimizdeki `vim'in` özel yapılandırmasına bağlı olarak, eşleşenler vurgulanacaktır. Vurgulanmazsa, vurgulamayı etkinleştirmek için şu komut modu komutunu deneyin:

```
:hlsearch
```

Not: Dağıtımınıza bağlı olarak, `vim` metin arama vurgulamayı destekleyebilir veya desteklemeyebilir. Özellikle Ubuntu, varsayılan olarak `vim'in` basitleştirilmiş bir sürümünü sağlar. Bu tür sistemlerde, paket yöneticinizi kullanarak `vim'in` daha eksiksiz bir sürümünü yüklemek isteyebilirsiniz.

Özet

Bu bölümde, düzenli ifadelerin birçok kullanımından birkaçını gördük. Düzenli ifadeleri, bunları kullanan ek uygulamaları aramak için kullanırsak daha fazlasını bulabiliriz. Bunu man sayfalarını arayarak yapabiliriz.

```
[me@linuxbox ~]$ cd /usr/share/man/man1
[me@linuxbox man1]$ zgrep -El 'regex|regular expression' *.gz
```

`zgrep` programı, `grep` için bir ön uç sağlar ve sıkıştırılmış dosyaları okumasına olanak tanır. Örneğimizde, sıkıştırılmış bölüm 1 man sayfası dosyalarını normal konumlarında arıyoruz

Bu komutun sonucu, `regex` dizisini veya `regular expression` dizisini içeren dosyaların bir listesidir. Gördüğümüz gibi, düzenli ifadeler birçok programda karşımıza çıkar.

Temel düzenli ifadelerde ele almadığımız bir özellik vardır. *Geriye dönük referanslar* olarak adlandırılan bu özellik, bir sonraki bölümde ele alınacaktır.

Daha Fazla Okuma

Düzenli ifadeleri öğrenmek için çeşitli öğreticiler ve hile sayfaları dahil olmak üzere birçok çevrimiçi kaynak bulunmaktadır.

Ayrıca, Wikipedia'da aşağıdaki arka plan konularıyla ilgili iyi makaleler bulunmaktadır:

- POSIX: https://en.wikipedia.org/wiki/Posix
- ASCII: https://en.wikipedia.org/wiki/Ascii

20 – Metin İşleme

Tüm Unix benzeri işletim sistemleri, veri depolama için büyük ölçüde metin dosyalarına dayanır. Bu nedenle, metinleri işlemek için birçok araç olması mantıklıdır. Bu bölümde, metinleri "dilimlemek ve parçalamak" için kullanılan programlara bakacağız. Bir sonraki bölümde, yazdırma ve diğer insan tüketimi için metinleri biçimlendirmek için kullanılan programlara odaklanarak, metin işlemeye daha ayrıntılı olarak bakacağız.

Bu bölümde, bazı eski dostları yeniden ziyaret edecek ve bazı yenilerini tanıyacağız:

- `cat` – Dosyaları birleştirir ve standart çıktıya yazdırır
- `sort` – Metin dosyalarının satırlarını sıralar
- `uniq` – Tekrarlanan satırları raporlar veya çıkarır
- `cut` – Dosyaların her satırından bölümleri kaldırır
- `yapıştır` – Dosyaların satırlarını birleştirir
- `join` – İki dosyanın satırlarını ortak bir alanda birleştirir
- `tac` – Dosyaları ters sırada birleştir ve yazdır
- `rev` – Satırları karakter bazında tersine çevir
- `comm` – İki sıralı dosyayı satır satır karşılaştır
- `diff` – Dosyaları satır satır karşılaştır
- `patch` – Bir diff dosyasını orijinal dosyaya uygular
- `tr` – Karakterleri çevir veya sil
- `sed` – Metni filtrelemek ve dönüştürmek için akış düzenleyici
- `aspell` – Etkileşimli yazım denetleyicisi

Metin Uygulamaları

Şimdiye kadar birkaç metin editörü (`nano` ve `vim`) öğrendik, bir dizi yapılandırma dosyasına baktık ve düzinelerce komutun çıktısını gördük, hepsi metin biçiminde. Peki

metin başka ne için kullanılır? Aslında birçok şey için kullanılır.

Belgeler

Birçok kişi, düz metin biçimlerini kullanarak belge yazar. Küçük bir metin dosyasının basit notlar tutmak için ne kadar yararlı olabileceğini görmek kolay olsa da, büyük belgeleri de metin biçiminde yazmak mümkündür. Popüler bir yaklaşım, büyük bir belgeyi metin biçiminde yazıp, bitmiş belgenin biçimlendirmesini tanımlamak için bir *işaretleme dili* eklemektir. Unix tabanlı metin işleme sistemleri, teknik disiplinlerdeki yazarların ihtiyaç duyduğu gelişmiş tipografik düzeni destekleyen ilk sistemler arasında olduğundan, birçok bilimsel makale bu yöntem kullanılarak yazılır. Son yıllarda, Markdown, belge biçimlendirme için düz metin kullanan popüler bir biçimlendirme dili haline gelmiştir.

Web Sayfaları

Dünyanın en popüler elektronik belge türü muhtemelen web sayfasıdır. Web sayfaları, belgenin görsel formatını tanımlamak için *Hypertext Markup Language* (*HTML*) veya *Extensible Markup Language* (*XML*) işaretleme dillerini kullanan metin belgeleridir.

E-posta

E-posta, özünde metin tabanlı bir ortamdır. Metin olmayan ekler bile iletim için metin temsiline dönüştürülür. Bunu, bir e-posta mesajını indirip `less` ile görüntüleyerek kendimiz görebiliriz. Mesajın, mesajın kaynağını ve yolculuğu sırasında aldığı işlemleri açıklayan bir başlıkla başladığını, ardından mesajın içeriğini içeren gövdesinin geldiğini göreceğiz.

Yazıcı Çıktısı

Unix benzeri sistemlerde, yazıcıya gönderilecek çıktı düz metin olarak gönderilir veya sayfa grafik içeriyorsa, genellikle *PostScript* olarak bilinen bir metin formatı *sayfa tanımlama diline* dönüştürülür ve ardından yazdırılacak grafik noktalarını oluşturan bir programa gönderilir.

Program Kaynak Kodu

Unix benzeri sistemlerde bulunan komut satırı programlarının çoğu, sistem yönetimi ve yazılım geliştirmeyi desteklemek için oluşturulmuştur ve metin işleme programları da buna bir istisna değildir. Bunların çoğu, yazılım geliştirme sorunlarını çözmek için tasarlanmıştır. Metin işlemenin yazılım geliştiricileri için önemli olmasının nedeni, tüm yazılımların metin olarak başlamasıdır. Programcının yazdığı programın bir parçası olan *kaynak kodu*, her zaman metin biçimindedir.

Eski Dostları Yeniden Ziyaret Etmek

6. bölümde, komut satırı argümanlarına ek olarak standart girişi kabul edebilen bazı komutları öğrendik. O zaman bunlara sadece kısaca değinmiştik, ancak şimdi bunların metin işleme için nasıl kullanılabileceğini daha yakından inceleyeceğiz.

cat

`cat` programı bir dizi ilginç seçeneğe sahiptir. Bunların çoğu, metin içeriğini daha iyi görselleştirmek için kullanılır. Bir örnek, metinde yazdırılmayan karakterleri görüntülemek için kullanılan -A seçeneğidir. Bazen, görünür metinlerimize kontrol karakterlerinin gömülü olup olmadığını bilmek isteriz. Bunların en yaygın olanları, MS-DOS tarzı metin dosyalarında genellikle satır sonu karakterleri olarak bulunan sekme karakterleri (boşlukların aksine) ve satır başı karakterleridir. Bir başka yaygın durum ise, satır sonlarında boşluklar bulunan metin satırları içeren dosyalardır.

`cat'i` basit bir kelime işlemci olarak kullanarak bir test dosyası oluşturalım. Bunu yapmak için, `cat` komutunu girip (yönlendirilmiş çıktı için bir dosya belirleyerek) metnimizi yazıp, satırı düzgün bir şekilde bitirmek için `Enter tuşuna` basıp, ardından `Ctrl-d` tuşuna basarak `cat'e` dosya sonuna ulaştığımızı belirtiriz. Bu örnekte, başa sekme karakteri girip satırı birkaç boşlukla bitiriyoruz:

```
[me@linuxbox ~]$ cat > foo.txt
     Hızlı kahverengi tilki tembel köpeğin üzerinden atlar.
[me@linuxbox ~]$
```

Ardından, metni görüntülemek için `cat komutunu -A` seçeneği ile kullanacağız:

```
[me@linuxbox ~]$ cat -A foo.txt
^IÇabuk kahverengi tilki tembel köpeğin üzerinden atlar.          $
[me@linuxbox ~]$
```

Sonuçlarda görebileceğimiz gibi, metnimizdeki sekme karakteri `^I` ile temsil edilmektedir. Bu, `Ctrl-i` anlamına gelen yaygın bir notasyon olup, sekme karakteriyle aynıdır. Ayrıca, satırın gerçek sonunda bir `$ işar`eti görünüyor ve bu da metnimizin sonda boşluklar içerdiğini gösteriyor.

MS-DOS Metni ve Unix Metni

Metinde yazdırılmayan karakterleri aramak için `cat komutunu` kullanmak isteyebileceğiniz nedenlerden biri, gizli satır başı karakterlerini bulmaktır. Gizli satır başı karakterleri nereden gelir? DOS ve Windows'tan! Unix ve DOS, metin dosyalarında satır sonunu aynı şekilde tanımlamaz. Unix, satırı satır besleme karakteri (ASCII 10) ile sonlandırırken, MS-DOS ve türevleri her metin satırını sonlandırmak için satır başı dönüşü (ASCII 13) ve satır besleme dizisini kullanır.

DOS formatındaki dosyaları Unix formatına dönüştürmenin birkaç yolu vardır. Birçok Linux sisteminde, metin dosyalarını DOS formatına ve DOS formatından dönüştürebilen `dos2unix` ve `unix2dos` adlı programlar vardır. Ancak, sisteminizde `dos2u-nix` yoksa endişelenmeyin. Metni DOS formatından Unix formatına dönüştürme işlemi basittir; sorun yaratan satır başı dönüşlerinin kaldırılmasını içerir. Bu, bu bölümün ilerleyen kısımlarında ele alınacak birkaç programla kolayca gerçekleştirilebilir.

`cat` ayrıca metni değiştirmek için kullanılan seçeneklere de sahiptir. En öne çıkan ikisi, satırları numaralandıran `-n` ve birden fazla boş satırın çıktısını bastıran `-s` seçenekleridir. Bunu şu şekilde gösterebiliriz:

```
[me@linuxbox ~]$ cat > foo.txt
Hızlı kahverengi tilki



tembel köpeğin üzerinden atlar.
[me@linuxbox ~]$ cat -ns foo.txt
     1  Hızlı kahverengi tilki
     2
     3  tembel köpeğin üzerinden atlar.
[me@linuxbox ~]$
```

Bu örnekte, iki boş satırla ayrılmış iki satırlık metin içeren `foo.txt` test dosyamızın yeni bir sürümünü oluşturuyoruz. -ns seçenekleriyle `cat` tarafından işlendikten sonra, fazladan boş satır kaldırılır ve kalan satırlar numaralandırılır. Bu, metin üzerinde gerçekleştirilecek çok büyük bir işlem olmasa da, yine de bir işlemdir.

sort

`sort` programı, standart girdinin içeriğini veya komut satırında belirtilen bir veya daha fazla dosyayı sıralar ve sonuçları standart çıktıya gönderir. `cat` ile kullandığımız aynı tekniği kullanarak, standart girdinin doğrudan klavyeden işlenmesini aşağıdaki gibi gösterebiliriz:

```
[me@linuxbox ~]$ sort > foo.txt c
b
a
[me@linuxbox ~]$ cat foo.txt a
b
c
```

Komutu girdikten sonra, `c`, `b` ve `a` harflerini yazıyoruz ve ardından `Ctrl-d tuşlarına` basarak dosyanın sonuna geldiğimizi belirtiyoruz. Ardından, ortaya çıkan dosyayı görüntüleyerek satırların artık sıralı olarak göründüğünü görüyoruz.

`sort` komutu, komut satırında birden fazla dosyayı argüman olarak kabul edebildiğinden, birden fazla dosyayı tek bir sıralı bütün halinde *birleştirmek* mümkündür. Örneğin, üç metin dosyamız varsa ve bunları tek bir sıralı dosyada birleştirmek istiyorsak, şöyle bir şey yapabiliriz:

```
sort file1.txt file2.txt file3.txt > final_sorted_list.txt
```

`sort komutunun` birkaç ilginç seçeneği vardır. Tablo 20-1 bunların bir kısmını içermektedir:

Tablo 20-1: Yaygın `sort` Seçenekleri

Seçenek	Uzun Seçenek	Açıklama
-b	--ignore-leading-blanks	Varsayılan olarak, sıralama satırın ilk karakterinden başlayarak tüm satır üzerinde gerçekleştirilir. Bu seçenek, `sort komutunun` satırların başındaki boşlukları yok saymasını ve satırdaki ilk boşluk olmayan karaktere göre sıralama hesaplamasını sağlar.
-f	--ignore-case	Sıralamayı büyük/küçük harf duyarlı hale getirir.
-n	--numeric-sort	Bir dizgenin sayısal değerlendirmesine göre sıralama yapar. Bu seçenek kullanıldığında, alfabetik değerler yerine sayısal değerler üzerinde sıralama yapılır.

`-r`	`--reverse`	Ters sırada sıralama yapar. Sonuçlar artan sırada değil, azalan sırada görüntülenir.
`-k`	`--key=alan1[,alan2]`	Tüm satır yerine *field1* ile *field2* arasında bulunan bir anahtar alana göre sıralama yapar. Aşağıdaki açıklamaya bakın.
`-m`	`--merge`	Her argümanı önceden sıralanmış bir dosyanın adı olarak ele alır. Ek bir sıralama yapmadan birden fazla dosyayı tek bir sıralı sonuçta birleştirir.
`-o`	`--output=dosya`	Sıralanmış çıktıyı standart çıktı yerine *dosyaya* gönderir.
`-t`	`--field-separator=karakter`	Alan ayırıcı karakteri tanımlayın. Varsayılan olarak alanlar boşluklar veya sekmelerle ayrılır.

Bu seçeneklerin çoğu oldukça açıklayıcı olsa da, bazıları öyle değildir. Öncelikle, sayısal sıralama için kullanılan -n seçeneğine bakalım. Bu seçenekle, değerleri sayısal değerlere göre sıralamak mümkündür. Bunu, disk alanını en fazla kullananları belirlemek için du komutunun sonuçlarını sıralayarak gösterebiliriz. Normalde, `du` komutu özet sonuçlarını yol adı sırasına göre listeler.

```
[me@linuxbox ~]$ du -s /usr/share/* | head
252         /usr/share/aclocal
96          /usr/share/acpi-support
8           /usr/share/adduser
196         /usr/share/alacarte
344         /usr/share/alsa
8           /usr/share/alsa-base
12488       /usr/share/anthy
8           /usr/share/apmd
```

```
21440      /usr/share/app-install
48         /usr/share/application-registry
```

Bu örnekte, sonuçları ilk 10 satırla sınırlamak için `head` komutuna aktarıyoruz. Bu şekilde, en fazla yer kaplayan 10 öğeyi gösteren sayısal olarak sıralanmış bir liste oluşturabiliriz.

```
[me@linuxbox ~]$ du -s /usr/share/* | sort -nr | head
 509940     /usr/share/locale-langpack
 242660     /usr/share/doc
 197560     /usr/share/fonts
 179144     /usr/share/gnome
 146764     /usr/share/myspell
 144304     /usr/share/gimp
 135880     /usr/share/dict
 76508      /usr/share/icons
 68072      /usr/paylaşım/uygulamalar
 62844      /usr/share/foomatic
```

`n` ve `r` seçeneklerini kullanarak, en büyük değerlerin sonuçlarda ilk sırada görünmesi için tersine sayısal sıralama yapıyoruz. Bu sıralama, sayısal değerlerin her satırın başında yer alması nedeniyle işe yarıyor. Peki, satır içinde bulunan bir değere göre listeyi sıralamak istersek ne yapmalıyız? Örneğin, `ls -l` komutunun sonuçları şöyle:

```
[me@linuxbox ~]$ ls -l /usr/bin | head toplam 152948

  -rwxr-xr-x    1  root      root           34824   2016-04-04    02:42  [
  -rwxr-xr-x    1  root      root          101556   2007-11-27    06:08  a2p
  -rwxr-xr-x    1  root      root           13036   2016-02-27    08:22  aconnect
  -rwxr-xr-x    1  root      root           10552   2007-08-15    10:34  acpi
  -rwxr-xr-x    1  kök       root            3800   2016-04-14    03:51  acpi_fakekey
  -rwxr-xr-x    1  root      root            7536   2016-04-19    00:19  acpi_listen
  -rwxr-xr-x    1  root      root            3576   2016-04-29    07:57  addpart
  -rwxr-xr-x    1  root      root           20808   2016-01-03    18:02  addr2line
  -rwxr-xr-x    1  root      root          489704   2016-10-09    17:02  adept_batch
```

Şu anda `ls` komutunun sonuçları boyuta göre sıralayabildiğini göz ardı edersek, `sort komutunu` kullanarak bu listeyi dosya boyutuna göre de sıralayabiliriz.

```
[me@linuxbox ~]$ ls -l /usr/bin | sort -nrk 5 | head
```

```
-rwxr-xr-x    1  root       root         8234216   2016-04-07    17:42  inkscape
-rwxr-xr-x    1  root       root         8222692   2016-04-07    17:42  inkview
-rwxr-xr-x    1  kök        root         3746508   2016-03-07    23:45  gimp-2.4
-rwxr-xr-x    1  root       root         3654020   2016-08-26    16:16  quanta
-rwxr-xr-x    1  root       root         2928760   2016-09-10    14:31  gdbtui
-rwxr-xr-x    1  root       root         2928756   2016-09-10    14:31  gdb
-rwxr-xr-x    1  root       root         2602236   2016-10-10    12:56  net
-rwxr-xr-x    1  kök        root         2304684   2016-10-10    12:56  rpcclient
-rwxr-xr-x    1  root       root         2241832   2016-04-04    05:56  aptitude
-rwxr-xr-x    1  root       root         2202476   2016-10-10    12:56  smbcacls
```

`sort komut`unun birçok kullanımı, önceki `ls` komutunun sonuçları gibi *tablo verilerinin* işlenmesini içerir. Önceki tabloya veritabanı terminolojisini uygularsak, her satırın bir *kayıt* olduğunu ve her kaydın dosya öznitelikleri, bağlantı sayısı, dosya adı, dosya boyutu vb. gibi birden çok *alandan* oluştuğunu söyleyebiliriz. `sort komutu`, tek tek alanları işleyebilir. Veritabanı terimleriyle, *sıralama anahtarı* olarak kullanılacak bir veya daha fazla *anahtar alanı* belirleyebiliriz. Önceki örnekte, tersine sayısal sıralama yapmak için `n` ve `r` seçeneklerini belirledik ve sıralamanın beşinci alanı anahtar olarak kullanması için `-k 5 seçeneğini` belirledik.

`k` seçeneği ilginçtir ve birçok özelliğe sahiptir, ancak önce `sort'un` alanları nasıl tanımladığını konuşmamız gerekir. Yazarın adını içeren tek bir satırdan oluşan aşağıdaki basit metin dosyasını ele alalım:

```
William Shotts
```

Varsayılan olarak, `sort` bu satırı iki alan içeren bir satır olarak görür. İlk alan şu karakterleri içerir:

`"William"`

İkinci alan şu karakterleri içerir:

`"Shotts"`

Bu, boşluk karakterlerinin (boşluklar ve sekmeler) alanlar arasında sınırlayıcı olarak kullanıldığı ve sıralama yapıldığında sınırlayıcıların alana dahil edildiği anlamına gelir.

Aşağıdaki gibi `ls` çıktımızdan bir satırı tekrar incelediğimizde, bir satırın sekiz alan içerdiğini ve beşinci alanın dosya boyutu olduğunu görebiliriz:

```
-rwxr-xr-x 1 kök           root       8234216 2016-04-07 17:42 inkscape
```

Bir sonraki deney serimiz için, 2006 ile 2008 yılları arasında piyasaya sürülen üç popüler Linux dağıtımının geçmişini içeren aşağıdaki dosyayı ele alalım. Dosyadaki her satır üç alandan oluşur: dağıtım adı, sürüm numarası ve MM/GG/YYYY biçiminde piyasaya sürülme tarihi.

```
SUSE        10.2            12/07/2006
Fedora      10              25/11/2008
SUSE        11.0            19.06.2008
Ubuntu      8.04            24.04.2008
Fedora      8               08.11.2007
SUSE        10.3            10/04/2007
Ubuntu      6.10            26/10/2006
Fedora      7               31.05.2007
Ubuntu      7.10            18/10/2007
Ubuntu      7.04            19.04.2007
SUSE        10.1            11.05.2006
Fedora      6               24/10/2006
Fedora      9               13.05.2008
Ubuntu      6.06            01.06.2006
Ubuntu      8.10            30/10/2008
Fedora      5               20/03/2006
```

Bir metin düzenleyici (örneğin `vim`) kullanarak bu verileri gireceğiz ve ortaya çıkan dosyaya `dis-tros.txt` adını vereceğiz.

Ardından, dosyayı sıralamayı deneyeceğiz ve şu sonuçları gözlemleyeceğiz:

```
[me@linuxbox      ~]$   sort distros.txt
Fedora    10              25/11/2008
Fedora    5               20/03/2006
Fedora    6               24.10.2006
Fedora    7               31.05.2007
Fedora    8               08.11.2007
Fedora    9               13.05.2008
SUSE         10.1         05/11/2006
SUSE         10.2         12/07/2006
SUSE         10.3         10/04/2007
SUSE        11.0          19/06/2008
Ubuntu    6.06            01/06/2006
Ubuntu    6.10            26/10/2006
Ubuntu    7.04            19/04/2007
Ubuntu    7.10            18.10.2007
```

```
Ubuntu    8.04            24.04.2008
Ubuntu    8.10            30/10/2008
```

Çoğunlukla işe yaradı. Sorun, Fedora sürüm numaralarının sıralanmasında ortaya çıkıyor. Karakter setinde 1, 5'ten önce geldiği için, sürüm 10 en üste, sürüm 9 ise en alta düşüyor.

Bu sorunu çözmek için, birden fazla anahtara göre sıralama yapmamız gerekecek. İlk alanda alfabetik sıralama, ikinci alanda ise sayısal sıralama yapmak istiyoruz. `sort,` birden fazla sıralama anahtarı belirtilebilmesi için -k seçeneğinin birden fazla kullanılmasına izin verir. Aslında, bir anahtar bir dizi alanı içerebilir. Herhangi bir aralık belirtilmezse (önceki örneklerimizde olduğu gibi), `sort` belirtilen alanla başlayan ve satırın sonuna kadar uzanan bir anahtar kullanır. İşte çoklu anahtar sıralamamızın sözdizimi:

```
[me@linuxbox      ~]$  sort --key=1,1 --key=2n distros.txt
Fedora    5               03/20/2006
Fedora    6               24/10/2006
Fedora    7               31.05.2007
Fedora    8               08.11.2007
Fedora    9               13.05.2008
Fedora    10              25/11/2008
SUSE         10.1         11.05.2006
SUSE         10.2         12/07/2006
SUSE         10.3         10/04/2007
SUSE        11.0          19/06/2008
Ubuntu    6.06            01/06/2006
Ubuntu    6.10            26/10/2006
Ubuntu    7.04            19.04.2007
Ubuntu    7.10            18.10.2007
Ubuntu    8.04            24.04.2008
Ubuntu    8.10            30.10.2008
```

Netlik için seçeneğin uzun biçimini kullandık, ancak `-k 1,1 -k 2n` de tam olarak aynı anlama gelir. Anahtar seçeneğinin ilk örneğinde, ilk anahtara dahil edilecek alan aralığını belirttik. Sıralamayı sadece ilk alanla sınırlamak istediğimizden, "1. alandan başlayıp 1. alanda bitir" anlamına gelen `1,1` değerini belirttik. İkinci örnekte, `2n` belirttik, bu da 2. alanın sıralama anahtarı olduğu ve sıralamanın sayısal olması gerektiği anlamına gelir. Sıralama anahtarının sonuna, gerçekleştirilecek sıralama türünü belirtmek için bir seçenek harfi eklenebilir. Bu seçenek harfleri, `sıralama` programı için genel seçeneklerle aynıdır: `b` (baştaki boşlukları yok say), `n` (sayısal sıralama), `r` (ters sıralama) vb.

Listemizdeki üçüncü alan, sıralama için uygun olmayan bir biçimde bir tarih içerir. Bilgisayarlarda

bilgisayarlarda tarihler genellikle kronolojik sıralamayı kolaylaştırmak için YYYY-MM-DD biçiminde biçimlendirilir, ancak bizim tarihlerimiz Amerikan biçimi olan MM/DD/YYYY biçimindedir. Bu listeyi kronolojik sırayla nasıl sıralayabiliriz?

Neyse ki, `sort` bir yol sunar. Anahtar seçeneği, alanlar içindeki *ofsetlerin* belirtilmesine izin verir, böylece alanlar içinde anahtarlar tanımlayabiliriz.

```
[me@linuxbox ~]$ sort -k 3.7nbr -k 3.1nbr -k 3.4nbr distros.txt
Fedora    10          25/11/2008
Ubuntu    8.10        30.10.2008
SUSE      11.0        19.06.2008
Fedora    9           13.05.2008
Ubuntu    8.04        24.04.2008
Fedora    8           08.11.2007
Ubuntu    7.10        18.10.2007
SUSE      10.3        04.10.2007
Fedora    7           31.05.2007
Ubuntu    7.04        19/04/2007
SUSE      10.2        07.12.2006
Ubuntu    6.10        26.10.2006
Fedora    6           24/10/2006
Ubuntu    6.06        01.06.2006
SUSE      10.1        11.05.2006
Fedora    5           20/03/2006
```

`-k 3.7` belirtilerek, `sıralama` işleminin üçüncü alandaki yedinci karakterden başlayan ve yılın başlangıcına karşılık gelen bir sıralama anahtarı kullanması talimatı verilir. Benzer şekilde, tarihin ay ve gün kısımlarını ayırmak için `-k 3.1` ve `-k 3.4` belirtilir. Tersine sayısal sıralama elde etmek için `n` ve `r` seçeneklerini de ekliyoruz. `b` seçeneği, tarih alanında bulunan baştaki boşlukları (sayıları satırdan satıra değişir ve bu da sıralama sonucunu etkiler) bastırmak için eklenmiştir.

Bazı dosyalar alan sınırlayıcı olarak sekme ve boşluk kullanmaz; örneğin, `/etc/ passwd` dosyası:

```
[me@linuxbox ~]$ head /etc/passwd root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
```

```
man:x:6:12:man:/var/cache/man:/bin/sh
lp:x:7:7:lp:/var/spool/lpd:/bin/sh mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
```

Bu dosyadaki alanlar iki nokta üst üste (`:`) ile ayrılmıştır, peki bu dosyayı bir anahtar alanı kullanarak nasıl sıralayabiliriz? `sort,` alan ayırıcı karakteri tanımlamak için `-t` seçeneği sunar. `Passwd` dosyasını yedinci alanda (hesabın varsayılan kabuğu) sıralamak için şunu yapabiliriz:

```
[me@linuxbox ~]$ sort -t ':' -k 7 /etc/passwd | head
me:x:1001:1001:Myself,,,:/home/me:/bin/bash
root:x:0:0:root:/root:/bin/bash dhcp:x:101:102::/nonexistent:/bin/false
gdm:x:106:114:Gnome Display Manager:/var/lib/gdm:/bin/false hplip:x:104:7:HPLIP system
user,,,:/var/run/hplip:/bin/false klog:x:103:104::/home/klog:/bin/false
messagebus:x:108:119::/var/run/dbus:/bin/false
polkituser:x:110:122:PolicyKit,,,:/var/run/PolicyKit:/bin/false pulse:x:107:116:PulseAudio
daemon,,,:/var/run/pulse:/bin/false
```

Alan ayırıcı olarak iki nokta üst üste karakterini belirleyerek, yedinci alana göre sıralama yapabiliriz. `Sıralama işleminin` varsayılan olarak birden fazla boşluk karakteri dizisini (sekme ve boşluklar) tek bir ayırıcı olarak ele alırken, diğer ayırıcıları ayrı ayrı ele aldığını unutmayın.

uniq

`Sort` ile karşılaştırıldığında, `uniq` programı hafiftir. `uniq,` görünüşte önemsiz bir görevi yerine getirir. Sıralanmış bir dosya (veya standart giriş) verildiğinde, yinelenen satırları kaldırır ve sonuçları standart çıktıya gönderir. Yinelenenleri temizlemek için genellikle `sort` ile birlikte kullanılır.

İpucu: `uniq`, `sort` ile birlikte sıklıkla kullanılan geleneksel bir Unix aracı olsa da, GNU sürümündeki `sort`, sıralanmış çıktılardan yinelenenleri kaldıran `-u` seçeneğini destekler.

Bunu denemek için burada gösterildiği gibi bir metin dosyası oluşturalım:

```
[me@linuxbox ~]$ cat > foo.txt a
```

```
b
c
a
b
c
```

Standart girişi sonlandırmak için `Ctrl-d` tuşlarına basmayı unutmayın. Şimdi, metin dosyamızda `uniq komutunu` çalıştırırsak, şu sonucu alırız:

```
[me@linuxbox ~]$ uniq foo.txt a
b
c
a
b
c
```

Sonuçlar orijinal dosyamızdan farklı değildir; yinelenenler kaldırılmamıştır.
`uniq` komutunun çalışması için önce girdinin sıralanması gerekir.

```
[me@linuxbox ~]$ sort foo.txt | uniq a
b
c
```

Bunun nedeni, `uniq komutunun` yalnızca birbirine bitişik olan yinelenen satırları kaldırmasıdır.

`uniq komutunun` birkaç seçeneği vardır. Tablo 20-2 yaygın olanları listelemektedir.

Tablo 20-2: Yaygın `uniq` Seçenekleri

Seçenek	Uzun Seçenek	Açıklama
`-c`	`--count`	Satırın kaç kez tekrarlandığı bilgisini içeren bir listeyi çıktı olarak verir.
`-d`	`--repeated`	Tekil satırlar yerine yalnızca tekrarlanan satırları çıkarır.
`-f` *n*	*`--skip-fields=n`*	Her satırdaki *n adet* baştaki alanı yok sayar. Alanlar, `sort` komutunda olduğu gibi boşluklarla ayrılır; ancak `sort komutundan` farklı olarak, `uniq` komutunda

		alternatif bir alan ayırıcı ayarlama seçeneği yoktur.
`-i`	`--ignore-case`	Satır karşılaştırmaları sırasında büyük/küçük harfleri yok say.
`-s n`	*`--skip-chars=n`*	Her satırın başındaki *n* karakteri atla (yoksay).
`-u`	`--unique`	Yalnızca benzersiz satırları çıkar. Yinelenen satırlar yok sayılır.

Burada, `-c` seçeneğini kullanarak metin dosyamızda bulunan yinelenenlerin sayısını bildirmek için `uniq komutunun` kullanıldığını görüyoruz:

```
[me@linuxbox ~]$ sort foo.txt | uniq -c
       2 a
       2 b
       2 c
```

Kesme ve Dilimleme

Bundan sonra ele alacağımız üç program, dosyalardan metin sütunlarını ayırmak ve bunları yararlı şekillerde yeniden birleştirmek için kullanılır.

`cut`

`cut` programı, bir satırdan bir metin bölümünü ayıklamak ve ayıklanan bölümü standart çıktıya göndermek için kullanılır. Birden fazla dosya argümanı veya standart girişten gelen girdileri kabul edebilir.

Çıkarılacak satır bölümünü belirtmek biraz zahmetlidir ve Tablo 20-3'te listelenen seçenekler kullanılarak belirtilir.

Tablo 20-3: `Kesme` Seçimi Seçenekleri

Seçenek	Uzun Seçenek	Açıklama
`-c` *`list`*	*`--characters=list`*	Listede tanımlanan satırın *list*ede tanımlanan kısmını çıkarır. Liste , virgülle ayrılmış bir veya daha fazla virgülle ayrılmış sayısal aralıklardan oluşabilir.

`-f` *liste*	`--fields=`*liste*	*Listede* tanımlandığı şekilde satırdan bir veya daha fazla alanı çıkarır. Liste, virgülle ayrılmış bir veya daha fazla alan veya alan aralığı içerebilir.
`-d` *sınırlayıcı*	`--delimeter=`*delim*	`-f` belirtildiğinde, *delim'i* alan sınırlayıcı karakter olarak kullanır. Varsayılan olarak, alanlar tek bir sekme karakteriyle ayrılmalıdır.
	`--complement`	`-c` ve/veya `-f` ile belirtilen kısımlar hariç, metnin tüm satırını çıkarır.

Gördüğümüz gibi, `cut` komutunun metin çıkarma yöntemi oldukça esnek değildir. `cut` komutu, insanlar tarafından doğrudan yazılan metinlerden ziyade, diğer programlar tarafından üretilen dosyalardan metin çıkarmak için en uygunudur. `distros.txt` dosyamızı inceleyerek, `cut` örneklerimiz için iyi bir örnek olacak kadar "temiz" olup olmadığını göreceğiz. `cat` komutunu `-A` seçeneği ile kullanırsak, dosyanın sekme ile ayrılmış alanlar gereksinimimizi karşılayıp karşılamadığını görebiliriz:

```
[me@linuxbox ~]$ cat -A distros.txt SUSE^I10.2^I12/07/2006$
Fedora^I10^I11/25/2008$
SUSE^I11.0^I06/19/2008$
Ubuntu^I8.04^I04/24/2008$
Fedora^I8^I11/08/2007$
SUSE^I10.3^I10/04/2007$
Ubuntu^I6.10^I26/10/2006$
Fedora^I7^I31/05/2007$
Ubuntu^I7.10^I18/10/2007$
Ubuntu^I7.04^I19/04/2007$
SUSE^I10.1^I11/05/2006$
Fedora^I6^I24/10/2006$
Fedora^I9^I13/05/2008$
Ubuntu^I6.06^I01/06/2006$
Ubuntu^I8.10^I30/10/2008$
Fedora^I5^I20/03/2006$
```

İyi görünüyor. Gömülü boşluklar yok, sadece

alanları. Dosya boşluklar yerine sekmeler kullandığından, bir alanı çıkarmak için `-f` seçeneğini kullanacağız.

```
[me@linuxbox ~]$ cut -f 3 distros.txt 12/07/2006
25/11/2008
06/19/2008
04/24/2008
08/11/2007
04/10/2007
26.10.2006
31.05.2007
18.10.2007
19/04/2007
11/05/2006
24.10.2006
13.05.2008
01.06.2006
30.10.2008
20/03/2006
```

`Distros` dosyamız sekmeyle ayrılmış olduğundan, karakterleri değil alanları ayıklamak için `cut komutunu` kullanmak en iyisidir. Bunun nedeni, bir dosya sekmeyle ayrılmış olduğunda, her satırın aynı sayıda karakter içermesi olasılığının düşük olmasıdır, bu da satır içindeki karakter konumlarını hesaplamayı zorlaştırır veya imkansız hale getirir. Ancak önceki örneğimizde, şans eseri aynı uzunlukta veriler içeren bir alanı ayıklamış durumdayız, bu nedenle her satırdan yılı ayıklayarak karakter ayıklamanın nasıl çalıştığını gösterebiliriz.

```
[me@linuxbox ~]$ cut -f 3 distros.txt | cut -c 7-10 2006
2008
2008
2008
2007
2007
2006
2007
2007
2007
2006
2006
```

```
2008
2006
2008
2006
```

Listemizde ikinci kez `kesme işlemi` gerçekleştirerek, tarih alanımızdaki yıla karşılık gelen 7 ila 10 arasındaki karakter konumlarını çıkarabiliriz. `7-10` notasyonu bir aralık örneğidir. `Kesme` man sayfası, aralıkların nasıl belirtilebileceğine dair tam bir açıklama içerir.

Sekmeleri Genişletme

`distros.txt` dosyamız, `cut komutunu` kullanarak alanları çıkarmak için ideal bir biçimde biçimlendirilmiştir. Peki, alanlar yerine karakterlere göre `cut komut`uyla tamamen işlenebilen bir dosya istersek ne yapmalıyız? Bunun için dosya içindeki sekme karakterlerini karşılık gelen sayıda boşlukla değiştirmemiz gerekir. Neyse ki, GNU Coreutils paketi bunun için bir araç içerir. `expand` adlı bu program, bir veya daha fazla dosya argümanı ya da standart girişi kabul eder ve değiştirilen metni standart çıktıya gönderir.

`distros.`txt dosyamızı `expand` ile işlersek, `cut -c komutunu` kullanarak dosyadan herhangi bir karakter aralığını çıkarabiliriz. Örneğin, dosyayı genişletip `cut` komutunu kullanarak 23. konumdan satırın sonuna kadar her karakteri çıkarmak suretiyle listemizden yayınlanma yılını çıkarabiliriz:

```
[me@linuxbox ~]$ expand distros.txt | cut -c 23-
```

Coreutils ayrıca sekmeleri boşluklarla değiştirmek için `unexpand` programını da sağlar.

Alanlarla çalışırken, sekme karakteri yerine farklı bir alan sınırlayıcı belirlemek mümkündür. Burada /etc/passwd dosyasından ilk alanı çıkaracağız:

```
[me@linuxbox ~]$ cut -d ':' -f 1 /etc/passwd | head root
daemon
bin sys
sync
games
```

```
man
lp
mail
news
```

`-d` seçeneğini kullanarak, iki nokta üst üste karakterini alan sınırlayıcı olarak belirtebiliriz.

paste

`Paste` komutu, `cut komut`unun tersini yapar. Bir dosyadan bir metin sütununu çıkarmak yerine, bir dosyaya bir veya daha fazla metin sütunu ekler. Bunu, birden fazla dosyayı okuyarak ve her dosyada bulunan alanları standart çıktıda tek bir akışta birleştirerek yapar. `Cut` gibi, `paste de` birden fazla dosya argümanı ve/veya standart girişi kabul eder. `Paste'in` nasıl çalıştığını göstermek için, `distros.`txt dosyamızda bazı işlemler yaparak sürümlerin kronolojik bir listesini oluşturacağız.

Daha önce `sort` ile yaptığımız çalışmadan yola çıkarak, önce tarihe göre sıralanmış bir distros listesi oluşturacak ve sonucu `distros-by-date.txt` adlı bir dosyada saklayacağız.

```
[me@linuxbox ~]$ sort -k 3.7nbr -k 3.1nbr -k 3.4nbr distros.txt > dis tros-by-date.txt
```

Ardından, `cut komutunu` kullanarak dosyadan ilk iki alanı (dağıtım adı ve sürümü) çıkaracağız ve sonucu `distros-versions.txt` adlı bir dosyada saklayacağız.

```
[me@linuxbox ~]$ cut -f 1,2 distros-by-date.txt > distros-versions.t xt
[me@linuxbox ~]$ head distros-versions.txt


Fedora      10
Ubuntu      8.10
SUSE        11.0
Fedora      9
Ubuntu      8.04
Fedora      8
Ubuntu      7.10
SUSE        10.3
Fedora      7
Ubuntu      7.04
```

Son hazırlık aşaması, sürüm tarihlerini ayıklamak ve bunları

distros-dates.txt.

```
[me@linuxbox ~]$ cut -f 3 distros-by-date.txt > distros-dates.txt [me@linuxbox ~]$ head
distros-dates.txt
25/11/2008
30/10/2008
19/06/2008
13.05.2008
24/04/2008
08.11.2007
18.10.2007
04.10.2007
31.05.2007
19.04.2007
```

Artık ihtiyacımız olan parçalara sahibiz. İşlemi tamamlamak için, paste komutunu kullanarak tarih sütununu dağıtım isimlerinin ve sürümlerinin önüne yerleştireceğiz ve böylece kronolojik bir liste oluşturacağız. Bu, paste komutunu kullanarak ve argümanlarını istenen sırayla düzenleyerek kolayca yapılabilir.

```
[me@linuxbox      ~]$ paste     distros-dates.txt       distros-versions.txt
25/11/2008        Fedora       10
30.10.2008        Ubuntu       8.10
19.06.2008        SUSE         11.0
13.05.2008        Fedora       9
24.04.2008        Ubuntu       8.04
08.11.2007        Fedora       8
18/10/2007        Ubuntu       7.10
04.10.2007        SUSE         10.3
31/05/2007        Fedora       7
19.04.2007        Ubuntu       7.04
07.12.2006        SUSE         10.2
26/10/2006        Ubuntu       6.10
24.10.2006        Fedora       6
01.06.2006        Ubuntu       6.06
05/11/2006        SUSE         10.1
20/03/2006        Fedora       5
```

katıl

Bazı açılardan, birleştirme, bir dosyaya sütunlar eklemesi açısından yapıştırmaya benzer, ancak bunu yapmak için benzersiz bir yöntem kullanır.

bir yöntem kullanır. *Join*, genellikle *ilişkisel veritabanlarıyla* ilişkili bir işlemdir ve paylaşılan bir anahtar alanı olan birden çok *tablodaki* veriler istenen sonucu oluşturmak için birleştirilir. `Join` programı da aynı işlemi gerçekleştirir. Paylaşılan bir anahtar alanına göre birden çok dosyadaki verileri birleştirir.

İlişkisel veritabanında birleştirme işleminin nasıl kullanıldığını görmek için, her biri tek bir kayıt içeren iki tablodan oluşan küçük bir veritabanı düşünelim. CUS-TOMERS adlı ilk tablo üç alana sahiptir: müşteri numarası (CUSTNUM), müşterinin adı (FNAME) ve müşterinin soyadı (LNAME):

```
CUSTNUM      FNAME        LNAME
========     =====        ======

4681934      John         Smith
```

İkinci tablo ORDERS olarak adlandırılır ve dört alan içerir: sipariş numarası (ORDER-NUM), müşteri numarası (CUSTNUM), miktar (QUAN) ve sipariş edilen ürün (ITEM).

```
ORDERNUM            CUSTNUM      QUAN ITEM
========            =======      ==== ====

3014953305          4681934      1      Mavi Widget
```

Her iki tablonun da CUSTNUM alanını paylaştığına dikkat edin. Bu, tablolar arasında bir ilişki kurulmasına olanak tanıdığı için önemlidir.

Birleştirme işlemi gerçekleştirmek, iki tablodaki alanları birleştirerek fatura hazırlamak gibi yararlı bir sonuç elde etmemizi sağlar. Her iki tablonun CUSTNUM alanlarındaki eşleşen değerleri kullanarak, birleştirme işlemi şu sonucu verebilir:

```
FNAME        LNAME        QUAN ITEM
=====        =====        ==== ====

John         Smith        1      Mavi Widget
```

`Birleştirme` programını göstermek için, paylaşılan bir anahtara sahip birkaç dosya oluşturmamız gerekecek. Bunu yapmak için `distros-by-date.txt` dosyamızı kullanacağız. Bu dosyadan iki ek dosya oluşturacağız. Biri, burada gösterildiği gibi yayın tarihlerini (bu gösterim için paylaşılan anahtarımız olacak) ve yayın adlarını içerecek

```
[me@linuxbox ~]$ cut -f 1,1 distros-by-date.txt > distros-names.txt [me@linuxbox ~]$ paste
distros-dates.txt distros-names.txt > distros-key-names.txt
[me@linuxbox ~]$ head distros-key-names.txt 25/11/2008
                 Fedora
30/10/2008       Ubuntu
19/06/2008       SUSE
13/05/2008       Fedora
```

```
24.04.2008      Ubuntu
08.11.2007      Fedora
18.10.2007      Ubuntu
04.10.2007      SUSE
31.05.2007      Fedora
19/04/2007      Ubuntu
```

İkinci dosya, burada gösterildiği gibi sürüm tarihlerini ve sürüm numaralarını içerir:

```
[me@linuxbox ~]$ cut -f 2,2 distros-by-date.txt > distros-vernums.txt [me@linuxbox ~]$ paste
distros-dates.txt distros-vernums.txt > distro s-key-vernums.txt


[me@linuxbox      ~]$ head distros-key-vernums.txt
25/11/2008        10
30.10.2008        8,10
19.06.2008        11
13.05.2008        9
24.04.2008        8,04
08.11.2007        8
18/10/2007        7,10
04/10/2007        10,3
31.05.2007        7
19/04/2007        7.04
```

Artık ortak bir anahtara ("yayın tarihi" alanı) sahip iki dosyamız var. `Birleştirmenin` düzgün çalışması için dosyaların anahtar alanına göre sıralanması gerektiğini belirtmek önemlidir.

```
[me@linuxbox ~]$ join distros-key-names.txt distros-key-vernums.txt | head
25/11/2008 Fedora 10
30.10.2008 Ubuntu 8.10
19.06.2008 SUSE 11.0
13.05.2008 Fedora 9
24.04.2008 Ubuntu 8.04
08.11.2007 Fedora 8
18.10.2007 Ubuntu 7.10
04.10.2007 SUSE 10.3
31.05.2007 Fedora 7
19/04/2007 Ubuntu 7.04
```

Ayrıca, varsayılan olarak `join`, giriş alanı sınırlayıcı olarak boşluk karakterini ve tek bir

çıktı alanı sınırlayıcı olarak boşluk. Bu davranış, seçenekler belirtilerek değiştirilebilir. Ayrıntılar için `join` man sayfasına bakın.

tac

`tac` komutu, kelimenin tam anlamıyla `cat` komutunun tersi gibi çalışır. Dosyaları ters sırayla birleştirir. `tac komutunu` kullanmanın bir yolu, günlük dosyalarını yeniden sıralamaktır. Günlük dosyaları genellikle şu şekilde görüntülenir:

```
[me@linuxbox ~]$ tail /var/log/backup.log
Çar   Ağ   14   06:00:32   AM   EDT   2025:    yedek   (0.6) linuxbox başlatıldı.
Çar   Ağ   14   06:08:01   06:0 EDT   2025:  linuxbox Linux kutusunun yedeklemesi tamamlandı.
                           8:01

Per   Ağ   15   07:36:30   07:3 EDT   2025:    yedek   (0.6) linuxbox başlatıldı.
                           6:30
Per   Ağ   15   07:45:57    07  EDT   2025:  linuxbox Linux kutusunun yedeklemesi tamamlandı.


Cuma  Ağ   16   07:37:57   07:3 EDT   2025:    yedek   (0.6) linuxbox başlatıldı.
                           7:57
Cuma  Ağ   16   07:44:11   07:4 EDT   2025:  linuxbox Linux kutusunun yedeklemesi tamamlandı.
                           4:11
```

Gördüğümüz gibi, günlük en son giriş en altta gösterilmektedir. En son olayı listenin en üstünde görmek istersek, bunu şu şekilde yapabiliriz:

```
[me@linuxbox ~]$ tail /var/log/backup.log | tac
Cuma  Ağ   16   07:44:11    AM  EDT   2025:  linuxbox Linux kutusunun yedeklemesi tamamlandı.
Cuma  Ağ   16   07:37:57   07:3 EDT   2025:    yedek   (0.6) linuxbox başlatıldı.
                           7:57

Per   Ağ   15   07:45:57    07  EDT   2025:  linuxbox Linux kutusunun yedeklemesi tamamlandı.
Per   Ağ   15   07:36:30   07:3 EDT   2025:    yedek   (0.6) linuxbox başlatıldı.
                           6:30

Çar   Ağ   14   06:08:01   06:0 EDT   2025:  linuxbox tamamlandı.
                           8:01
Çar   Ağ   14   06:00:32   06:0 EDT   2025:    yedek   (0.6) linuxbox başlatıldı.
                           0:32
```

rev

Benzer şekilde, `rev` komutu bir dizgedeki karakterleri tersine çevirir:

```
[me@linuxbox ~]$ echo "Bu bir testtir." | rev
.tset a si sihT
```

Hoş, ama ne işe yarar? `rev` komutu genellikle `cut` ile birlikte kullanılır. `Yedekleme.log` dosyamızı tekrar ele alalım. Her satırın sonundaki nokta karakterini kaldırmak istediğimizi varsayalım. Satırların uzunlukları farklı olduğundan, satırın sonundan bir şeyi kaldırmak için `cut` komutunu kullanmak mümkün değildir. İşte burada `rev komutu` devreye girer. Her satırdaki karakterleri tersine çevirerek, noktaları başa taşıyabilir ve `cut` komutuyla kolayca kaldırabiliriz:

```
[me@linuxbox ~]$ tail /var/log/backup.log | rev | cut -c 2- | rev
Çar   Ağ   14   06:00:32   AM   EDT   2025:  yedekleme (0.6) linuxbox başlatıldı
Çar   Ağ   14   06:08:01   06:0 EDT   2025:  linuxbox Linux kutusunun yedeklemesi tamamlandı
                           8:01

Per   Ağ   15   07:36:30   07:3 EDT   2025:  yedekleme (0.6) linuxbox başlatıldı
                           6:30
Per   Ağ   15   07:45:57   07:4 EDT   2025:  linuxbox Linux kutusunun yedeklemesi tamamlandı
                           5:57

Cuma  Ağ   16   07:37:57   07:  EDT   2025:    yedek   (0.6) linuxbox başlatıldı
Cuma  Ağ   16   07:44:11   07:4 EDT   2025:  linuxbox Linuxbox yedeği tamamlandı
                           4:11
```

Bu örnekte, `rev komutunu` kullanarak her satırdaki karakterleri tersine çeviriyoruz, satırın içeriğini 2. konumdan (tersine çevrilmiş dizgedeki nokta işaretinden sonraki karakter) satırın sonuna kadar kesip, karakterleri tekrar tersine çevirerek okunabilir hale getiriyoruz.

Metin Karşılaştırma

Metin dosyalarının sürümlerini karşılaştırmak genellikle yararlıdır. Sistem yöneticileri ve yazılım geliştiricileri için bu özellikle önemlidir. Örneğin, bir sistem yöneticisi, bir sistem sorununu teşhis etmek için mevcut bir yapılandırma dosyasını önceki bir sürümle karşılaştırması gerekebilir. Benzer şekilde, bir programcı da zaman içinde programlarda yapılan değişiklikleri sık sık görmesi gerekir.

comm

`comm` programı iki metin dosyasını karşılaştırır ve her birine özgü satırları ve ortak satırları görüntüler. Bunu göstermek için, `cat komutunu` kullanarak neredeyse aynı iki metin dosyası oluşturacağız.

```
[me@linuxbox ~]$ cat > file1.txt a
b
c
d
```

```
[me@linuxbox ~]$ cat > file2.txt b
c
d
e
```

Ardından, `comm komutunu` kullanarak iki dosyayı karşılaştıracağız:

```
[me@linuxbox ~]$ comm file1.txt file2.txt a
          b
          c
          d
    e
```

Gördüğümüz gibi, `comm` üç sütunlu bir çıktı üretir. İlk sütun ilk dosya argümanına özgü satırları, ikinci sütun ikinci dosya argümanına özgü satırları ve üçüncü sütun her iki dosyada da bulunan satırları içerir. `comm`, *n*'nin 1, 2 veya 3 olduğu *-n* biçimindeki seçenekleri destekler. Bu seçenekler kullanıldığında, hangi sütunların gizleneceğini belirtir. Örneğin, her iki dosyada da bulunan satırları çıktılamak istiyorsak, birinci ve ikinci sütunların çıktısını gizleriz.

```
[me@linuxbox ~]$ comm -12 file1.txt file2.txt b
c
d
```

diff

`Comm` programı gibi, `diff` de dosyalar arasındaki farkları tespit etmek için kullanılır. Ancak `diff`, birçok çıktı formatını destekleyen ve büyük metin dosyası koleksiyonlarını tek seferde işleyebilen çok daha karmaşık bir araçtır. `diff`, yazılım geliştiricileri tarafından program kaynak kodunun farklı sürümleri arasındaki değişiklikleri incelemek için sıklıkla kullanılır ve bu sayede, genellikle *kaynak ağaçları* olarak adlandırılan kaynak kod dizinlerini yinelemeli olarak inceleme yeteneğine sahiptir. `Diff'in` yaygın kullanım alanlarından biri, bir dosyanın (veya dosyaların) bir sürümünü başka bir sürüme dönüştürmek için `patch` (birazdan ele alacağız) gibi programlar tarafından kullanılan *diff dosyaları* veya *yamalar* oluşturmaktır.

`diff'i` önceki örnek dosyalarımızı incelemek için kullanırsak, varsayılan çıktı stilini görürüz:

iki dosya arasındaki farkların kısa bir açıklaması.

```
[me@linuxbox ~]$ diff file1.txt file2.txt 1d0
< a
4a4
> e
```

Varsayılan biçimde, her değişiklik grubunun önünde, Tablo 20-4'te özetlendiği gibi, ilk dosyayı ikinci dosyaya dönüştürmek için gereken değişikliklerin konumlarını ve türlerini açıklamak üzere *aralık işlemi aralığı* biçiminde bir *değişiklik komutu* bulunur.

Tablo 20-4: `diff` Değişiklik Komutları

Değişiklik	Açıklama
r1ar2	İkinci dosyadaki *r2* konumundaki satırları, ilk dosyadaki r1 konumuna ekleyin. *r1* konumuna ekleyin.
r1cr2	İkinci dosyadaki *r2* konumundaki satırları, birinci dosyadaki *r1* konumundaki satırlarla değiştirin (değiştirin).
r1dr2	İlk dosyadaki *r1* konumundaki satırları silin; bu satırlar ikinci dosyada *r2* aralığında görünürdü.

Bu formatta, aralık, başlangıç satırı ve bitiş satırının virgülle ayrılmış bir listesidir. Bu format varsayılan format olsa da (çoğunlukla POSIX uyumluluğu ve geleneksel Unix `diff` sürümleriyle geriye dönük uyumluluk için), diğer isteğe bağlı formatlar kadar yaygın olarak kullanılmamaktadır. En popüler formatlardan ikisi *bağlam formatı* ve *birleştirilmiş formattır*.

Bağlam formatı (`-c` seçeneği) kullanılarak görüntülendiğinde, şunu göreceğiz:

```
[me@linuxbox ~]$ diff -c file1.txt file2.txt
*** file1.txt 2025-12-23 06:40:13.000000000 -0500
--- file2.txt 2025-12-23 06:40:34.000000000 -0500
***************
*** 1,4 ****
-  a
   b
   c
   d
      1,4


----      -----
```

```
   b
   c
   d
+ e
```

Çıktı, iki dosyanın adları ve zaman damgalarıyla başlar. İlk dosya yıldız işaretleriyle, ikinci dosya ise tire işaretleriyle işaretlenmiştir. Listenin geri kalanında, bu işaretler ilgili dosyaları belirtir. Ardından, varsayılan çevreleyen bağlam satırlarının sayısı da dahil olmak üzere değişiklik gruplarını görürüz. İlk grupta, ilk dosyadaki 1'den 4'e kadar olan satırları gösteren şunu görürüz:

```
*** 1,4 ***
```

Daha sonra, ikinci dosyadaki 1'den 4'e kadar olan satırları gösteren şunu görürüz: :

```
--- 1,4 ---
```

Bir değişiklik grubu içinde, satırlar Tablo 20-5'te gösterilen dört göstergeden biriyle başlar.

Tablo 20-5: `diff` Bağlam Biçimi Değişiklik Göstergeleri

Gösterge	Anlam
boş	Bağlam için gösterilen bir satır. İki dosya arasındaki farkı göstermez.
-	Silinen bir satır. Bu satır ilk dosyada görünecek, ancak ikinci dosyada görünmeyecektir.
+	Eklenen bir satır. Bu satır ikinci dosyada görünecek, ancak ilk dosyada görünmeyecektir.
!	Bir satır değiştirildi. Satırın iki versiyonu, her biri değişiklik grubunun ilgili bölümünde görüntülenecektir.

Birleştirilmiş biçim, bağlam biçimine benzer ancak daha kısadır. `-u` seçeneği ile belirtilir.

```
[me@linuxbox ~]$ diff -u file1.txt file2.txt
--- file1.txt 2008-12-23 06:40:13.000000000 -0500
+++ file2.txt 2008-12-23 06:40:34.00000000 -0500
@@ -1,4 +1,4 @@
-a
 b
 c
```

```
  d
 +e
```

Bağlam ve birleştirilmiş biçimler arasındaki en belirgin fark, bağlamdaki yinelenen satırların ortadan kaldırılmasıdır, bu da birleştirilmiş biçimin sonuçlarının bağlam biçiminden daha kısa olmasını sağlar. Önceki örneğimizde, bağlam biçimindekine benzer dosya zaman damgaları ve ardından `@@ -1,4 +1,4 @@` dizesini görüyoruz. Bu, değişiklik grubunda açıklanan ilk dosyadaki satırları ve ikinci dosyadaki satırları gösterir. Bunu, varsayılan üç satırlık bağlamla birlikte satırların kendisi izler. Her satır, Tablo 20-6'da listelenen üç olası karakterden biriyle başlar.

Tablo 20-6: `diff` Birleştirilmiş Biçim Değişiklik Göstergeleri

Karakter	Anlam
boş	Bu satır her iki dosya tarafından paylaşılır.
-	Bu satır ilk dosyadan kaldırılmıştır.
+	Bu satır ilk dosyaya eklendi.

patch

`Yama` programı, metin dosyalarına değişiklikleri uygulamak için kullanılır. `Diff programının` çıktısını kabul eder ve genellikle eski sürüm dosyaları yeni sürümlere dönüştürmek için kullanılır. Ünlü bir örneği ele alalım. Linux çekirdeği, kaynak koduna sürekli olarak küçük değişiklikler gönderen, geniş ve gevşek bir şekilde organize olmuş bir katkıcı ekibi tarafından geliştirilmiştir. Linux çekirdeği birkaç milyon satırlık koddan oluşurken, bir katkıcı tarafından bir seferde yapılan değişiklikler oldukça küçüktür. Bir katkıcı, her küçük değişiklik yapıldığında tüm çekirdek kaynak ağacını her geliştiriciye göndermek mantıklı değildir. Bunun yerine, bir diff dosyası gönderilir. Diff dosyası, çekirdeğin önceki sürümünden katkıcıların değişikliklerini içeren yeni sürüme yapılan değişiklikleri içerir. Alıcı daha sonra `patch` programını kullanarak değişikliği kendi kaynak ağacına uygular. `Diff/patch` kullanmanın iki önemli avantajı vardır.

1. Diff dosyası, kaynak ağacının tam boyutuna kıyasla küçüktür.
2. Diff dosyası yapılan değişikliği kısaca gösterir ve yamanın gözden geçirenlerinin hızlı bir şekilde değerlendirmesini sağlar.

Elbette, `diff/patch` sadece kaynak kodunda değil, herhangi bir metin dosyasında da çalışır. Yapılandırma dosyalarına veya diğer metinlere de aynı şekilde uygulanabilir.

`Patch` ile kullanmak üzere bir diff dosyası hazırlamak için, GNU belgeleri (bkz. Daha Fazla Okuma

aşağıda) `diff komutunu` şu şekilde kullanmayı önerir:

`diff -Naur` *`eski_dosya yeni_dosya > diff_dosya`*

burada *eski_dosya* ve *yeni_dosya* tek bir dosya veya dosya içeren dizinlerdir. `r` seçeneği, bir dizin ağacının özyinelemesini destekler.

diff dosyası oluşturulduktan sonra, eski dosyayı yeni dosyaya yamalamak için uygulayabiliriz.

`patch <` *`diff_file`*

Test dosyamızla bunu göstereceğiz.

```
[me@linuxbox ~]$ diff -Naur file1.txt file2.txt > patchfile.txt [me@linuxbox ~]$ patch <
patchfile.txt
file1.txt dosyasını yamalama [me@linuxbox
~]$ cat file1.txt b
c
d
e
```

Bu örnekte, `patchfile.txt` adlı bir diff dosyası oluşturduk ve ardından `patch` programını kullanarak yamayı uyguladık. Diff dosyası (birleştirilmiş biçimde) başlıkta dosya adlarını zaten içerdiğinden, `yamalanacak` hedef dosyayı belirtmemiz gerekmediğini unutmayın. Yamayı uyguladıktan sonra, `file1.txt dosyasının` artık `file2.txt` dosyasıyla eşleştiğini görebiliriz.

`patch` çok sayıda seçeneğe sahiptir ve yamaları analiz etmek ve düzenlemek için kullanılabilecek ek yardımcı programlar da vardır.

Anında Düzenleme

Metin düzenleyicilerle olan deneyimimiz büyük ölçüde *etkileşimli* olmuştur, yani imleci manuel olarak hareket ettirip değişikliklerimizi yazıyoruz. Ancak, metni düzenlemenin *etkileşimli olmayan* yolları da vardır. Örneğin, tek bir komutla birden fazla dosyaya bir dizi değişiklik uygulamak mümkündür.

`tr`

`tr` programı karakterleri *çevirmek* için kullanılır. Bunu bir tür karakter tabanlı arama ve değiştirme işlemi olarak düşünebiliriz. Çeviri, karakterleri bir alfabeden diğerine değiştirme işlemidir. Örneğin, karakterleri küçük harften büyük harfe dönüştürmek çeviridir. `tr` ile böyle bir dönüştürme işlemini şu şekilde gerçekleştirebiliriz:

```
[me@linuxbox ~]$ echo "küçük harfler" | tr a-z A-Z KÜÇÜK HARFLER
```

Gördüğümüz gibi, `tr` standart girişte çalışır ve sonuçlarını standart çıkışta verir. `tr` iki argüman kabul eder: dönüştürülecek karakter kümesi ve dönüştürülecek karşılık gelen karakter kümesi. Karakter kümeleri üç yoldan biriyle ifade edilebilir.

1. Numaralandırılmış bir liste. Örneğin, `ABCDEFGHIJKLMNOPQRSTUVWXYZ`
2. Bir karakter aralığı. Örneğin, `A-Z`. Bu yöntemin, yerel sıralama düzeni nedeniyle bazen diğer komutlarla aynı sorunlara maruz kaldığına dikkat edin ve bu nedenle dikkatli kullanın.
3. POSIX karakter sınıfları. Örneğin, `[:upper:]`.

Çoğu durumda, her iki karakter kümesi de eşit uzunlukta olmalıdır; ancak, özellikle birden fazla karakteri tek bir karaktere dönüştürmek istediğimizde, ilk kümenin ikinci kümeden daha büyük olması mümkündür.

```
[me@linuxbox ~]$ echo "küçük harfler" | tr [:lower:] A AAAAAAAAA AAAAAAA
```

Transliterasyona ek olarak, `tr` karakterlerin giriş akışından basitçe silinmesine de izin verir. Bu bölümün başında, MS-DOS metin dosyalarını Unix tarzı metne dönüştürme sorununu tartışmıştık. Bu dönüştürmeyi gerçekleştirmek için, her satırın sonundan satır başı karakterlerinin kaldırılması gerekir. Bu, `tr` ile aşağıdaki gibi gerçekleştirilebilir:

```
tr -d '\r' < dos_file > unix_file
```

Burada *dos_file* dönüştürülecek dosyadır ve *unix_file* sonuç dosyadır. Bu komut biçimi, satır başı karakterini temsil etmek için `\r` kaçış dizisini kullanır. `tr'nin` desteklediği dizilerin ve karakter sınıflarının tam listesini görmek için aşağıdakileri deneyin:

```
[me@linuxbox ~]$ tr --help
```

ROT13: Çok da Gizli Olmayan Şifre Çözücü

`tr` komutunun eğlenceli kullanımlarından biri, metinlerin *ROT13 kodlamasını* yapmaktır. ROT13, basit bir ikame şifreleme yöntemine dayanan önemsiz bir şifreleme türüdür. ROT13'ü "şifreleme" olarak adlandırmak...

"şifreleme" demek biraz abartılı olur; "metin gizleme" daha doğru bir tanımdır. Bazen metinlerde potansiyel olarak rahatsız edici içeriği gizlemek için kullanılır. Bu yöntem, her karakteri alfabenin 13 basamak yukarısına taşır. Bu, 26 karakterin yarısı kadar bir mesafe olduğundan, algoritmayı metin üzerinde ikinci kez uyguladığınızda metin orijinal haline geri döner. `tr` ile bu kodlamayı gerçekleştirmek için aşağıdakileri kullanın:

```
echo "gizli metin" | tr a-zA-Z n-za-mN-ZA-M
frperg grkg
```

Aynı prosedürü ikinci kez uyguladığınızda, aşağıdaki çeviri elde edilir:

```
echo "frperg grkg" | tr a-zA-Z n-za-mN-ZA-M
gizli metin
```

Birçok e-posta programı ve Usenet haber okuyucusu ROT13 kodlamasını desteklemektedir. Wikipedia bu konu hakkında iyi bir makale içermektedir:

https://en.wikipedia.org/wiki/ROT13

`tr` başka bir numara da yapabilir. `-s` seçeneğini kullanarak, `tr` bir karakterin tekrar eden örneklerini "sıkıştırabilir" (silebilir).

```
[me@linuxbox ~]$ echo "aaabbbccc" | tr -s ab abccc
```

Burada, tekrarlanan karakterler içeren bir dize var. `tr'ye` "ab" kümesini belirterek, kümedeki harflerin tekrarlanan örneklerini ortadan kaldırırken, kümede eksik olan karakteri ("c") değiştirmeden bırakıyoruz. Tekrarlanan karakterlerin bitişik olması gerektiğine dikkat edin. Bitişik değillerse, sıkıştırma işlemi etkili olmaz.

```
[me@linuxbox ~]$ echo "abcabcabc" | tr -s ab abcabcabc
```

sed

`sed` adı, *stream editor* (akış *düzenleyici*) kelimesinin kısaltmasıdır. Belirli bir dosya kümesi veya standart giriş gibi bir metin akışında metin düzenleme işlemi gerçekleştirir. `sed,` güçlü ve biraz karmaşık bir programdır (bu programla ilgili kitaplar bile vardır), bu nedenle burada tam olarak ele almayacağız.

Genel olarak, `sed'in` çalışma şekli, tek bir düzenleme komutu (

komut satırında) veya birden fazla komut içeren bir komut dosyasının adı verilir ve ardından metin akışındaki her satırda bu komutları gerçekleştirir. İşte sed'in çalışmasına dair basit bir örnek:

```
[me@linuxbox ~]$ echo "ön" | sed 's/ön/arka/' arka
```

Bu örnekte, echo kullanarak tek kelimelik bir metin akışı oluşturuyor ve bunu sed'e aktarıyoruz. sed ise akıştaki metin üzerinde s/front/back/ komutunu uyguluyor ve sonuç olarak "back" çıktısını üretiyor. Bu komutu, vi'deki "substitution" (ara ve değiştir) komutuna benzetebiliriz.

sed'deki komutlar tek bir harfle başlar. Önceki örnekte, değiştirme komutu s harfiyle temsil edilir ve ardından, sınırlayıcı olarak eğik çizgi karakteriyle ayrılmış arama ve değiştirme dizeleri gelir. Sınırlayıcı karakterin seçimi keyfidir. Geleneksel olarak, eğik çizgi karakteri sıklıkla kullanılır, ancak sed komutun hemen ardından gelen herhangi bir karakteri sınırlayıcı olarak kabul eder. Aynı komutu şu şekilde de gerçekleştirebiliriz:

```
[me@linuxbox ~]$ echo "front" | sed 's_front_back_' back
```

Komutun hemen ardından alt çizgi karakterini kullanarak, bu karakter sınırlayıcı olur. Sınırlayıcıyı ayarlama özelliği, göreceğimiz gibi komutları daha okunabilir hale getirmek için kullanılabilir.

sed'deki çoğu komutun önüne, giriş akışının hangi satırlarının düzenleneceğini belirten bir *adres* eklenebilir. Adres atlanırsa, düzenleme komutu giriş akışındaki her satırda uygulanır. Adresin en basit biçimi satır numarasıdır. Örneğimize bir tane ekleyebiliriz.

```
[me@linuxbox ~]$ echo "front" | sed '1s/front/back/' back
```

Komutumuza 1 adresini eklemek, tek satırlık giriş akışımızın ilk satırında değiştirme işleminin gerçekleştirilmesine neden olur. Başka bir sayı belirtirsek ve giriş akışımızda 2. satır olmadığı için düzenleme işleminin gerçekleştirilmediğini görürüz.

```
[me@linuxbox ~]$ echo "ön" | sed '2s/ön/arka/' ön
```

Adresler birçok şekilde ifade edilebilir. Tablo 20-7 en yaygın olanları listelemektedir.

Tablo 20-7: `sed` Adres Notasyonu

Adres	Açıklama
n	*n'nin* pozitif bir tamsayı olduğu satır numarası.
$	Son satır.
/regexp/	POSIX temel düzenli ifadesine uyan satırlar. Düzenli ifadenin eğik çizgi karakterleriyle sınırlandırıldığına dikkat edin. İsteğe bağlı olarak, düzenli ifade alternatif bir karakterle sınırlandırılabilir; bunun için ifadeyi *cregexpc* ile belirtin, burada *c* alternatif karakterdir.
addr1,*addr2*	*addr1* ile *addr2* arasındaki satır aralığı. Adresler, daha önce listelenen tek adres biçimlerinden herhangi biri olabilir.
first~*step*	*Önce* sayı ile temsil edilen satırı, ardından *adım* aralıklarıyla sonraki her satırı eşleştirin. Örneğin, 1~2 her tek sayılı satırı, 5~5 ise beşinci satırı ve ondan sonraki her beşinci satırı ifade eder.
addr1,+n	*addr1* ve sonraki *n* satırları eşleştirin.
addr!	*addr* hariç tüm satırları eşleştirin; addr, daha önce listelenen biçimlerden herhangi biri olabilir.

Bu bölümün başında yer alan `distros.txt` dosyasını kullanarak farklı adres türlerini göstereceğiz. İlk olarak, bir dizi satır numarası:

```
[me@linuxbox      ~]$ sed -n '1,5p' distros.txt
SUSE         10.2        12/07/2006
Fedora    10             25/11/2008
SUSE        11.0         19/06/2008
Ubuntu    8.04           24.04.2008
Fedora    8        08.11.2007
```

Bu örnekte, 1. satırdan başlayıp 5. satıra kadar olan bir dizi satırı yazdırıyoruz. Bunu yapmak için, eşleşen satırın yazdırılmasını sağlayan `p` komutunu kullanıyoruz.

Bunun etkili olabilmesi için, `sed'in` varsayılan olarak her satırı yazdırmaması için `-n` seçeneğini ("otomatik yazdırma yok" seçeneği) eklemeliyiz.

Şimdi, bir düzenli ifade deneyeceğiz.

```
[me@linuxbox ~]$ sed -n '/SUSE/p' distros.txt
SUSE         10.2        12/07/2006
SUSE        11.0         19/06/2008
SUSE         10.3        04.10.2007
SUSE         10.1        05/11/2006
```

Eğik çizgiyle ayrılmış düzenli ifade `/SUSE/`'yi ekleyerek, `grep` ile hemen hemen aynı şekilde bunu içeren satırları ayırabiliriz.

Son olarak, adrese ünlem işareti (`!`) ekleyerek olumsuzlama yapmayı deneyeceğiz.

```
[me@linuxbox ~]$ sed -n '/SUSE/!p' distros.txt
Fedora   10           25/11/2008
Ubuntu   8.04         24/04/2008
Fedora   8            08.11.2007
Ubuntu   6.10         26.10.2006
Fedora   7            31.05.2007
Ubuntu   7.10         18.10.2007
Ubuntu   7.04         19.04.2007
Fedora   6            24.10.2006
Fedora   9            13.05.2008
Ubuntu   6.06         01.06.2006
Ubuntu   8.10         30/10/2008
Fedora   5            20/03/2006
```

Burada beklenen sonucu görüyoruz: düzenli ifadeyle eşleşenler hariç, dosyadaki tüm satırlar.

Şimdiye kadar, `sed` düzenleme komutlarından ikisini, `s` ve `p'yi` inceledik. Tablo 20-8, temel düzenleme komutlarının daha eksiksiz bir listesini sunmaktadır.

Tablo 20-8: `sed` Temel Düzenleme Komutları

Komut	Açıklama
`=`	Geçerli satır numarasını görüntüler.
`a`	Geçerli satırın sonuna metin ekler.

`d`	Geçerli satırı siler.
`i`	Mevcut satırın önüne metin ekler.
`p`	Geçerli satırı yazdırır. Varsayılan olarak, `sed` her satırı yazdırır ve yalnızca dosya içinde belirtilen adresle eşleşen satırları düzenler. Varsayılan davranış, `-n` seçeneği belirtilerek geçersiz kılınabilir.
`q`	Başka satır işlemeyi bırakarak `sed`'den çıkın. `-n` seçeneği belirtilmezse, geçerli satırı çıkar.
`Q`	Başka satır işlemeyi bırakarak `sed`'den çıkın.
`s/regexp/replacement/`	*regexp* bulunduğu her yerde *replacement* içeriğini değiştirir. *replacement*, *regexp* ile eşleşen metne eşdeğer olan `&` özel karakterini içerebilir. Ayrıca, *replacement*, *regexp* içindeki karşılık gelen alt ifadelerin içeriği olan `\1` ile `\9` arasındaki dizileri içerebilir. Bununla ilgili daha fazla bilgi için aşağıdaki *geri referanslar* bölümüne bakın. *Replacement'ın* sonundaki eğik çizgiden sonra, `s` komutunun davranışını değiştirmek için isteğe bağlı bir bayrak belirtilebilir.
`y/set1/set2`	Karakterleri *set1'den set2'deki* karşılık gelen karakterlere dönüştürerek transliterasyon gerçekleştirin. `tr`'den farklı olarak, `sed`'in her iki setin de aynı uzunlukta olmasını gerektirdiğini unutmayın.

`s` komutu, en sık kullanılan düzenleme komutudur. `distros.txt` dosyamızda bir düzenleme yaparak bu komutun bazı özelliklerini göstereceğiz. Daha önce `distros.txt` dosyasındaki tarih alanının "bilgisayar dostu" bir formatta olmadığını tartışmıştık. Tarih MM/DD/YYYY formatında olsa da, (sıralama kolaylığı açısından) YYYY-MM-DD formatında olması daha iyi olurdu. Bu değişikliği dosyada elle yapmak hem zaman alıcı hem de hataya açık olurdu, ancak `sed` ile bu değişiklik tek adımda gerçekleştirilebilir.

```
[me@linuxbox ~]$ sed 's/\([0-9]\{2\}\)\/\([0-9]\{2\}\)\/\([0-9]\{4\}\
)$/\3-\1-\2/' distros.txt
SUSE        10.2            2006-12-07
```

```
Fedora      10               2008-11-25
SUSE        11.0            19.06.2008
Ubuntu      8.04             2008-04-24
Fedora      8                2007-11-08
SUSE        10.3            04.10.2007
Ubuntu      6.10             2006-10-26
Fedora      7               31.05.2007
Ubuntu      7.10          18 Ekim 2007
Ubuntu      7.04            19.04.2007
SUSE        10.1             2006-05-11
Fedora      6                2006-10-24
Fedora      9                2008-05-13
Ubuntu      6.06             2006-06-01
Ubuntu      8.10            30.10.2008
Fedora      5                2006-03-20
```

Vay canına! Ne çirkin bir komut. Ama işe yarıyor. Tek bir adımda, dosyamızdaki tarih formatını değiştirdik. Bu, düzenli ifadelerin neden bazen şakayla "sadece yazma" ortamı olarak adlandırıldığının mükemmel bir örneğidir. Onları yazabiliriz, ama bazen okuyamayız. Bu komuttan korkarak kaçmak istemeden önce, nasıl oluşturulduğuna bir bakalım. İlk olarak, komutun şu temel yapıya sahip olacağını biliyoruz.

```
sed 's/regexp/replacement/' distros.txt
```

Bir sonraki adımımız, tarihi ayıracak bir düzenli ifade bulmaktır. Tarih MM/DD/YYYY biçiminde ve satırın sonunda yer aldığı için, şu ifadeyi kullanabiliriz:

```
[0-9]{2}/[0-9]{2}/[0-9]{4}$
```

Bu ifade iki rakam, bir eğik çizgi, iki rakam, bir eğik çizgi, dört rakam ve satır sonunu eşleştirir. Böylece *düzenli ifade* sorunu çözülmüş oldu, peki ya *değiştirme*? Bunu halletmek için, BRE kullanan bazı uygulamalarda görünen yeni bir düzenli ifade özelliği tanıtmalıyız. Bu özellik *geriye dönük referanslar* olarak adlandırılır ve şu şekilde çalışır: *n* dizisi, *n'nin* 1 ile 9 arasında bir sayı olduğu bir *değiştirmede* görünürse, dizi önceki düzenli ifadede karşılık gelen alt ifadeye referans verir. Alt ifadeleri oluşturmak için, bunları şu şekilde parantez içine alırız:

```
([0-9]{2})/([0-9]{2})/([0-9]{4})$
```

Artık üç alt ifademiz var. İlki ayı, ikincisi ayın gününü ve üçüncüsü yılı içerir. Şimdi aşağıdaki gibi *bir değiştirme* oluşturabiliriz:

```
\3-\1-\2
```

Bu bize yılı, bir tire, ayı, bir tire ve günü verir. Şimdi komutumuz şöyle

görünüyor:

```
sed 's/([0-9]{2})/([0-9]{2})/([0-9]{4})$/\3-\1-\2/' distros.txt
```

İki sorunumuz kaldı. Birincisi, düzenli ifademizdeki fazladan eğik çizgiler, sed komutunu yorumlamaya çalışırken sed'i karıştıracaktır. İkincisi, `sed` varsayılan olarak yalnızca temel düzenli ifadeleri kabul ettiğinden, düzenli ifademizdeki bazı karakterler meta karakterler yerine gerçek karakterler olarak alınacaktır. Bu iki sorunu da, sorunlu karakterleri kaçırmak için ters eğik çizgileri bolca kullanarak çözebiliriz.

```
sed 's/\([0-9]\{2\}\)\/\([0-9]\{2\}\)\/\([0-9]\{4\}\)$/\3-\1-\2/' dis tros.txt
```

İşte bu kadar!

s komutunun bir başka özelliği de, değiştirme dizesinin ardından gelen isteğe bağlı bayrakların kullanılmasıdır. Bunların en önemlisi g bayrağıdır. Bu bayrak, `sed'e` arama ve değiştirme işlemini varsayılan olarak ilk örneğe değil, satırın tamamına uygulaması talimatını verir. İşte bir örnek:

```
[me@linuxbox ~]$ echo "aaabbbccc" | sed 's/b/B/' aaaBbbccc
```

Değiştirme işleminin gerçekleştirildiğini görüyoruz, ancak sadece `b` harfinin ilk örneğine uygulanmış, geri kalan örnekler ise değiştirilmemiş. `g` bayrağını ekleyerek, tüm örnekleri değiştirebiliyoruz.

```
[me@linuxbox ~]$ echo "aaabbbccc" | sed 's/b/B/g' aaaBBBccc
```

Şimdiye kadar, komut satırı üzerinden `sed'e` tekil komutlar verdik. `-f` seçeneğini kullanarak bir komut dosyasında daha karmaşık komutlar oluşturmak da mümkündür. Bunu göstermek için, `sed'i` distros`.txt` dosyamızla birlikte kullanarak bir rapor oluşturacağız. Raporumuzun üst kısmında bir başlık, değiştirme tarihlerimiz ve büyük harfe dönüştürülmüş tüm dağıtım adları yer alacak. Bunu yapmak için bir komut dosyası yazmamız gerekecek, bu yüzden metin düzenleyicimizi açıp aşağıdakileri gireceğiz:

```
# Linux dağıtımları raporu oluşturmak için sed komut dosyası

1 i\
\
Linux Dağıtımları Raporu\

s/\([0-9]\{2\}\)\/\([0-9]\{2\}\)\/\([0-9]\{4\}\)$/\3-\1-\2/
y/abcdefghijklmnopqrstuvwxyz/ABCDEFGHIJKLMNOPQRSTUVWXYZ/
```

`Sed` komut dosyasını `distros.sed` olarak kaydedeceğiz ve şu şekilde çalıştıracağız:

```
SUSE        10.2            2006-12-07
FEDORA      10              2008-11-25
SUSE        11.0            19.06.2008
UBUNTU      8.04            24.04.2008
FEDORA      8               2007-11-08
SUSE        10.3            04.10.2007
UBUNTU      6.10            2006-10-26
FEDORA      7               31.05.2007
UBUNTU      7.10            2007-10-18
UBUNTU      7.04            19.04.2007
SUSE        10.1            2006-05-11
FEDORA      6               2006-10-24
FEDORA      9               2008-05-13
UBUNTU      6.06            2006-06-01
UBUNTU      8.10            30.10.2008
```

```
FEDORA    5                  2006-03-20
```

Gördüğümüz gibi, komut dosyamız istenen sonuçları veriyor, ama bunu nasıl yapıyor? Komut dosyamıza bir kez daha bakalım. Satırları numaralandırmak için `cat komutunu` kullanacağız.

```
[me@linuxbox ~]$ cat -n distros.sed
     1  Linux dağıtımları raporu oluşturmak için sed komut dosyası
     2
     3
     4  1 i\
     5  \
     6  Linux Dağıtımları Raporu\
     7
     8  s/\([0-9]\{2\}\)\/\([0-9]\{2\}\)\/\([0-9]\{4\}\)$/\3-\1-\2/
        y/abcdefghijklmnopqrstuvwxyz/ABCDEFGHIJKLMNOPQRSTUVWXYZ/
```

Komut dosyasının birinci satırı *bir yorumdur*. Linux sistemlerindeki birçok yapılandırma dosyası ve programlama dili gibi, yorumlar # karakteriyle başlar ve ardından insan tarafından okunabilir metin gelir. Yorumlar komut dosyasının herhangi bir yerine yerleştirilebilir (komutların içine değil) ve komut dosyasını tanımlamak ve/veya bakımını yapmak isteyen kişiler için yararlıdır.

2. satır boş bir satırdır. Yorumlar gibi, okunabilirliği artırmak için boş satırlar da eklenebilir.

Birçok `sed` komutu satır adreslerini destekler. Bunlar, girdinin hangi satırlarının işleneceğini belirtmek için kullanılır. Satır adresleri tek satır numaraları, satır numarası aralıkları ve girdinin son satırını belirten özel satır numarası `$` olarak ifade edilebilir.

3, 4, 5 ve 6. satırlar, girişin ilk satırı olan adres 1'e eklenecek metni içerir. i komutunun ardından, ters eğik çizgi ve ardından satır başı karakteri gelir ve bu, *kaçış karakteri olarak* adlandırılan satır başı *karakterini* oluşturur. Kabuk betikleri dahil birçok durumda kullanılabilen bu dizi, satır sonuna ulaşıldığını yorumlayıcıya (bu durumda `sed`) bildirmeden satır başı karakterinin metin akışına gömülmesini sağlar. `i`, `a` (metni eklemek yerine ekleyen) ve `c` (metni değiştiren) komutları, son satır hariç her satırın satır devam karakteriyle bitmesi koşuluyla birden fazla satır metin kullanılmasına izin verir. Komut dosyamızın altıncı satırı aslında eklediğimiz metnin sonudur ve satır devam karakteri yerine düz bir satır başı karakteriyle biter, bu da `i` komutunun sonunu belirtir.

Not: Satır devam karakteri, ters eğik çizgi ve *hemen* ardından gelen satır başı karakterinden oluşur.

. Arada boşluk bırakılamaz.

7. satır, arama ve değiştirme komutumuzdur. Önünde bir adres bulunmadığından, giriş akışındaki her satır bu komutun etkisine tabidir.

8. satır, küçük harfleri büyük harflere çevirir. `tr komutundan` farklı olarak, `sed komutundaki y` komutu karakter aralıklarını (örneğin, `[a-z]`) veya POSIX karakter sınıflarını desteklemez. Yine, `y` komutunun önünde bir adres bulunmadığından, giriş akışındaki her satıra uygulanır.

`sed'i` Sevenler Şunları da Sever...

`sed`, metin akışlarına oldukça karmaşık düzenleme işlemleri uygulayabilen yetenekli bir programdır. Genellikle uzun komut dosyaları yerine basit, tek satırlık işlemler için kullanılır. Birçok kullanıcı, daha büyük görevler için başka araçları tercih eder. Bunların en popüler olanları `awk` ve `perl'dir`. Bunlar, burada ele alınan programlar gibi basit araçların ötesine geçer ve tam programlama dilleri alanına uzanır. Özellikle `perl`, birçok sistem yönetimi ve idare görevinde kabuk komut dosyaları yerine sıklıkla kullanılır ve web geliştirme için popüler bir ortamdır. `awk` biraz daha özeldir. Özel gücü, tablo verilerini işleme yeteneğidir. `Awk` programları normalde metin dosyalarını satır satır işler ve `sed`'in adresin ardından eylem kavramına benzer bir şema kullanır. Hem `awk` hem de `perl` bu kitabın kapsamı dışındadır, ancak Linux komut satırı kullanıcılarının öğrenmesi gereken iyi becerilerdir.

`aspell`

Son olarak inceleyeceğimiz araç, etkileşimli bir yazım denetleyicisi olan `aspell`'dir. `aspell` programı, `ispell` adlı eski bir programın halefidir ve çoğunlukla onun yerine kullanılabilir. `aspell` programı çoğunlukla yazım denetimi özelliği gerektiren diğer programlar tarafından kullanılır, ancak komut satırından bağımsız bir araç olarak da etkili bir şekilde kullanılabilir. HTML belgeleri, C/C++ programları, e-posta mesajları ve diğer özel metin türleri dahil olmak üzere çeşitli metin dosyalarını akıllıca denetleme özelliğine sahiptir. Tüm Linux dağıtımları varsayılan olarak aspell'i içermez, bu nedenle onu yüklememiz gerekebilir.

Basit düz metin içeren bir metin dosyasının yazım denetimi için şu şekilde kullanılabilir:

```
aspell check textfile
```

burada *textfile*, kontrol edilecek dosyanın adıdır. Pratik bir örnek olarak, bazı kasıtlı yazım hataları içeren `foo.txt` adlı basit bir metin dosyası oluşturalım.

```
[me@linuxbox ~]$ cat > foo.txt
Hızlı kahverengi tilki tembel köpeğin üzerinden atlar.
```

Şimdi dosyayı `aspell` kullanarak kontrol edeceğiz:

```
[me@linuxbox ~]$ aspell check foo.txt
```

`aspell` kontrol modunda etkileşimli olduğundan, aşağıdaki gibi bir ekran göreceğiz:

```
Hızlı kahverengi tilki tembel köpeğin üzerinden atlar.
                                                    
1) atlar                                6) topallamak
2) imps                                 7) pezevenkler
3) gimps                                8) wimps
4) atlar                                9) comps
5) Jim'in                               0) sakatlar
i) Yoksay                               I) Tümünü yok say
r) Değiştir                             R) Tümünü değiştir
a) Ekle                                 l) Alt ekle
b) İptal                                x) Çıkış

?
```

Ekranın üst kısmında, şüpheli bir şekilde yazılmış kelime vurgulanmış olarak metnimiz görünüyor. Ortada, sıfırdan dokuza kadar numaralandırılmış 10 yazım önerisi ve ardından diğer olası eylemlerin bir listesi görünüyor. Son olarak, alt kısımda seçimimizi kabul etmeye hazır bir komut istemi görünüyor.

`1` tuşuna basarsak, `aspell` hatalı kelimeyi "jumps" kelimesiyle değiştirir ve bir sonraki yanlış yazılmış kelimeye, yani laxy'ye geçer. lazy kelimesini seçersek, `aspell` onu değiştirir ve sonlandırır. `aspell` bittiğinde, dosyamızı inceleyebilir ve yanlış yazımların düzeltildiğini görebiliriz:

```
[me@linuxbox ~]$ cat foo.txt
Hızlı kahverengi tilki tembel köpeğin üzerinden atlar.
```

Komut satırı seçeneği `--dont-backup` ile aksi belirtilmedikçe, `aspell` dosya adına `.bak` uzantısını ekleyerek orijinal metni içeren bir yedek dosya oluşturur.

`Sed` düzenleme becerilerimizi sergilemek için, yazım hatalarımızı geri koyarak dosyamızı yeniden kullanabileceğiz.

```
[me@linuxbox ~]$ sed -i 's/lazy/laxy/; s/jumps/jimps/' foo.txt
```

`sed` seçeneği `-i`, `sed'e` dosyayı "yerinde" düzenlemesini söyler, yani düzenlenmiş çıktıyı standart çıktıya göndermek yerine, değişiklikleri uygulayarak dosyayı yeniden yazar. Ayrıca, satırda birden fazla düzenleme komutunu noktalı virgülle ayırarak yerleştirebileceğimizi de görüyoruz.

Şimdi, `aspell'in` farklı türdeki metin dosyalarını nasıl işleyebildiğine bakacağız. `Vim` gibi bir metin düzenleyici kullanarak (cesur olanlar `sed'i` denemek isteyebilir), dosyamıza bazı HTML işaretlemeleri ekleyeceğiz.

```
<html>
     <head>
          <title>Yazım hatası olan HTML dosyası</title>
     </head>
     <body>
          <p>Hızlı kahverengi tilki tembel köpeğin üzerinden atlar.
     </body>
</html>
```

Şimdi, değiştirilmiş dosyamızın yazım denetimini yapmaya çalıştığımızda bir sorunla karşılaşıyoruz. Bunu şu şekilde yaparsak:

```
[me@linuxbox ~]$ aspell check foo.txt
```

şunu alırız:

```
<html>
```

```
          <head>
                    <title>Yazım hatası içeren HTML dosyası</title>
          </head>
          <body>
                    <p>Hızlı kahverengi tilki tembel köpeğin üzerinden atlar.</p>
          </body>
</html>

1) HTML                                                 4) Hamil
2) ht ml                                                5) Hamil
3) ht-ml                                                6) otel
i) Yoksay                                               I) Tümünü yok say
r) Değiştir                                             R) Tümünü değiştir
a) Ekle                                                 l) Alt ekle
b) İptal                                                x) Çık

?
```

`aspell`, HTML etiketlerinin içeriğini yazım hatası olarak görecektir. Bu sorun, `-H` (HTML) kontrol modu seçeneğini ekleyerek aşılabilir, örneğin:

```
[me@linuxbox ~]$ aspell -H check foo.txt
```

Bu, şu sonucu verecektir:

```
<html>
          <head>
                    <title>Yazım hatası olan HTML dosyası</title>
          </head>
          <body>

                    <p>Hızlı kahverengi tilki tembel köpeğin üzerinden atlar.
          </body>
</html>

1)  Mi yazım hatası                                     6)  Yanlış
                                                            kullanılmış
2)  Yanlış yazılmış                                     7)  Yanlış
                                                            adlandırılmış
3)  Yanlış yazılmış                                     8)  Yeniden yazım
4)  Giderildi                                           9)  Yanlış yazım
5)  Yazım                                               0)  Yanlış
                                                            yönlendirilmiş
i)  Görmezden gelmek                                    I)  Tümünü yok say
```

```
r) Değiştir                                   R) Tümünü değiştir
a) Ekle                                       l) Alt ekle
b) İptal                                      x) Çıkış
[redacted]
?
```

HTML yok sayılır ve dosyanın yalnızca işaretleme içermeyen kısımları kontrol edilir. Bu modda, HTML etiketlerinin içeriği yok sayılır ve yazım denetimi yapılmaz. Ancak, denetimden yararlanan `ALT` etiketlerinin içeriği bu modda denetlenir.

Not: Varsayılan olarak, `aspell` metindeki URL'leri ve e-posta adreslerini yok sayar. Bu davranış, komut satırı seçenekleriyle geçersiz kılınabilir. Hangi biçimlendirme etiketlerinin kontrol edileceğini ve atlanacağını belirtmek de mümkündür. Ayrıntılar için `aspell` man sayfasına bakın.

Özet

Bu bölümde, metin üzerinde çalışan birçok komut satırı aracından birkaçını inceledik. Bir sonraki bölümde, birkaç tane daha inceleyeceğiz. Kabul etmek gerekir ki, bu araçların bazılarını günlük hayatta nasıl ve neden kullanabileceğiniz hemen anlaşılmayabilir, ancak bunların kullanımına ilişkin bazı pratik örnekler göstermeye çalıştık. Sonraki bölümlerde, bu araçların bir dizi pratik sorunu çözmek için kullanılan bir araç setinin temelini oluşturduğunu göreceğiz. Bu, özellikle bu araçların gerçek değerini göstereceği kabuk betikleme konusuna girdiğimizde geçerli olacaktır.

Daha Fazla Okuma

GNU Projesi web sitesi, bu bölümde ele alınan araçlarla ilgili birçok çevrimiçi kılavuz içerir.

- Coreutils paketinden: https://www.gnu.org/software/coreutils/manual/coreutils.html#Output-of-entire-files
  https://www.gnu.org/software/coreutils/manual/coreutils.html#Operating-on-sorted-files
  https://www.gnu.org/software/coreutils/manual/coreutils.html#Operating-on-fields
  https://www.gnu.org/software/coreutils/manual/coreutils.html#Operating-on-char-acters
- Diffutils paketinden:

https://www.gnu.org/software/diffutils/manual/html_mono/diff.html

- `sed`: https://www.gnu.org/software/sed/manual/sed.html
- `aspell`:
  https://aspell.net/man-html/index.html
- `Sed` için birçok başka çevrimiçi kaynak da bulunmaktadır, özellikle:
  https://www.grymoire.com/Unix/Sed.html
  https://www.pement.org/sed/sed1line.txt
- Ayrıca "sed one liners" ve "sed cheat sheets" kelimelerini Google'da aratmayı deneyin.

Ekstra Puan

Araştırmaya değer birkaç ilginç metin işleme komutu daha vardır. Bunlar arasında `split` (dosyaları parçalara ayırma), `csplit` (dosyaları bağlama göre parçalara ayırma) ve `sdiff` (dosya farklılıklarını yan yana birleştirme) bulunur.

21 – Çıktı Biçimlendirme

Bu bölümde, metinle ilgili araçlara bakmaya devam ederek, metnin kendisini değiştirmek yerine metin çıktısını biçimlendirmek için kullanılan programlara odaklanacağız. Bu araçlar genellikle metni baskıya hazırlamak için kullanılır; bu konuyu bir sonraki bölümde ele alacağız. Bu bölümde aşağıdaki programları ele alacağız:

- `nl` – Satır numaralandırma
- `fold` – Her satırı belirli bir uzunlukta sarmalama
- `fmt` – Basit bir metin biçimlendirici
- `pr` – Metni yazdırmaya hazırlar
- `printf` – Verileri biçimlendirme ve yazdırma
- `groff` – Bir belge biçimlendirme sistemi

Basit Biçimlendirme Araçları

Öncelikle bazı basit biçimlendirme araçlarına bakacağız. Bunlar çoğunlukla tek amaçlı programlardır ve yaptıkları işlerde biraz basit kalırlar, ancak küçük görevler için ve boru hatları ve komut dosyalarının parçaları olarak kullanılabilirler.

`nl` – Satırları numaralandırır

`nl` programı, basit bir görevi yerine getirmek için kullanılan oldukça gizemli bir araçtır. Satırları numaralandırır. En basit kullanımında, `cat -n` komutuna benzer.

```
[me@linuxbox ~]$ nl distros.txt | head
      1    SUSE         10.2             12/07/2006
      2    Fedora       10               25/11/2008
      3    SUSE         11.0             19.06.2008
      4    Ubuntu       8.04             24.04.2008
      5    Fedora       8                08.11.2007
      6    SUSE         10.3             10/04/2007
```

```
     7   Ubuntu       6.10             26/10/2006
     8   Fedora       7                31.05.2007
     9   Ubuntu       7.10             18/10/2007
    10   Ubuntu       7.04             19.04.2007
```

`cat` gibi, `nl` de komut satırı argümanları veya standart giriş olarak birden fazla dosyayı kabul edebilir. Ancak, `nl` bir dizi seçeneğe sahiptir ve daha karmaşık numaralandırma türlerine izin vermek için ilkel bir biçimlendirme biçimini destekler.

`nl`, numaralandırma sırasında "mantıksal sayfalar" adlı bir kavramı destekler. Bu, `nl'nin` numaralandırma sırasında sayısal dizisini sıfırlamasına (yeniden başlatmasına) olanak tanır. Seçenekleri kullanarak, başlangıç numarasını belirli bir değere ve sınırlı ölçüde formatını ayarlamak mümkündür. Mantıksal sayfa, başlık, gövde ve altbilgi olarak daha da ayrılır. Bu bölümlerin her birinde, satır numaralandırması sıfırlanabilir ve/veya farklı bir stil atanabilir. `nl'ye` birden fazla dosya verilirse, bunları tek bir metin akışı olarak ele alır. Metin akışındaki bölümler, Tablo 21-1'de açıklandığı gibi, metne eklenen oldukça garip görünümlü bazı işaretlemelerle belirtilir.

Tablo 21-1: nl İşaretlemesi

İşaretleme	Anlam
`\:\:\:`	Mantıksal sayfa başlığının başlangıcı
`\:\:`	Mantıksal sayfa gövdesinin başlangıcı
`\:`	Mantıksal sayfa altbilgisinin başlangıcı

Tablo 21-1'de listelenen her bir işaretleme öğesi kendi satırında tek başına görünmelidir. Bir işaretleme öğesini işledikten sonra, `nl` onu metin akışından siler.

Tablo 21-2, `nl` için yaygın seçenekleri listeler.

Tablo 21-2: Yaygın `nl` Seçenekleri

Seçenek	Anlam
`-b` *`stil`*	Gövde numaralandırmasını *style* olarak ayarlayın, burada *style* aşağıdakilerden biridir: `a` = Tüm satırları numaralandır `t` = Boş olmayan satırları numaralandır. Bu varsayılan ayardır. `n` = Yok *`pregexp`* = Yalnızca temel düzenli ifade *regexp* ile eşleşen satırları numaralandır.
`-f` *`stil`*	Altbilgi numaralandırmasını *stile* ayarla. Varsayılan ayar n (yok) şeklindedir.

`-h` *`stil`*	Başlık numaralandırmasını *stile* ayarlayın. Varsayılan değer n (yok) şeklindedir.
`-i` *`number`*	Sayfa numaralandırma artışını *sayıya* ayarlayın. Varsayılan değer birdir.
`-n` *`format`*	Numaralandırma biçimini *format* olarak ayarlar. *format* aşağıdakilerden biridir: `ln` = Sıfırlarla başlamadan sola hizalanır. `rn` = Sağda hizalanmış, baştaki sıfırlar olmadan. Bu varsayılan ayardır. `rz` = Sağda hizalanmış, baştaki sıfırlar dahil.
`-p`	Her mantıksal sayfanın başında sayfa numaralandırmasını sıfırlamayın.
`-s` *`dize`*	Her satır numarasının sonuna bir ayırıcı oluşturmak için *bir dize* ekleyin. Varsayılan ayırıcı tek bir sekme karakteridir.
`-v` *`number`*	Her mantıksal sayfanın ilk satır numarasını *sayı* olarak ayarlayın. Varsayılan değer birdir.
`-w` *`genişlik`*	Satır numarası alanının genişliğini *genişlik* olarak ayarlar. Varsayılan değer 6'dır.

Kabul etmek gerekir ki, satırları çok sık numaralandırmayacağız, ancak `nl'yi` kullanarak birden fazla aracı birleştirerek daha karmaşık görevleri nasıl gerçekleştirebileceğimizi görebiliriz. Önceki bölümdeki çalışmamızı temel alarak bir Linux dağıtımları raporu oluşturacağız. `nl` kullanacağımız için, başlık/gövde/altbilgi işaretlemesini eklemek faydalı olacaktır. Bunu yapmak için, önceki bölümdeki sed betiğine ekleyeceğiz. Metin düzenleyicimizi kullanarak betiği aşağıdaki gibi değiştireceğiz ve `distros-nl.sed` olarak kaydedeceğiz:

```
# Linux dağıtımları raporu oluşturmak için sed komut dosyası

1 i\
\\:\\:\\:\
\
Linux Dağıtımları Raporu\
\
Ad                Sürüm Yayınlanma Tarihi\
-----------------------------------------------------\
\\:\\:
s/\([0-9]\{2\}\)\/\([0-9]\{2\}\)\/\([0-9]\{4\}\)$/\3-\1-\2/
$ a\
\\:\
\
Raporun Sonu
```

Komut dosyası şimdi `nl` mantıksal sayfa işaretlemesini ekler ve raporun sonuna bir altbilgi ekler. İşaretlememizde ters eğik çizgileri iki katına çıkarmak zorunda kaldığımızı unutmayın, çünkü bunlar normalde `sed` tarafından kaçış karakteri olarak yorumlanır.

Ardından, `sort`, `sed` ve `nl'yi` birleştirerek gelişmiş raporumuzu oluşturacağız.

```
[me@linuxbox ~]$ sort -k 1,1 -k 2n distros.txt | sed -f distros-nl.s
ed | nl



         Linux Dağıtımları Raporu

         Ad          Sürüm       Yayınlanma
         ----        ----        Tarihi
                                 --------

       1   Fedora    5              2006-03-20
       2   Fedora    6            24 Ekim 2006
       3   Fedora    7           31 Mayıs 2007
       4   Fedora    8              2007-11-08
       5   Fedora    9              2008-05-13
       6   Fedora    10             2008-11-25
       7   SUSE      10.1           2006-05-11
       8   SUSE      10.2           07.12.2006
       9   SUSE      10.3           04.10.2007
      10   SUSE      11.0           19.06.2008
      11   Ubuntu    6.06           01.06.2006
      12   Ubuntu    6.10           2006-10-26
      13   Ubuntu    7.04           19.04.2007
      14   Ubuntu    7.10           18.10.2007
      15   Ubuntu    8.04           2008-04-24
      16   Ubuntu    8.10           30.10.2008



         Raporun Sonu
```

Raporumuz, komut dizimizin sonucudur. İlk olarak, listeyi dağıtım adı ve sürümüne göre sıralıyoruz (alan 1 ve 2) ve ardından sonuçları `sed` ile işleyerek rapor başlığını (`nl` için mantıksal sayfa işaretlemesi dahil) ve altbilgiyi ekliyoruz. Son olarak, sonucu `nl` ile işliyoruz. nl, varsayılan olarak, mantıksal sayfanın gövde bölümüne ait metin akışının satırlarını numaralandırır.

Komutu tekrarlayabilir ve `nl` için farklı seçenekleri deneyebiliriz. Bazı ilginç

olanlardan bazıları şunlardır

```
nl -n rz
```

ve aşağıdakiler:

```
nl -w 3 -s ' '
```

`fold` – Her satırı belirli bir uzunlukta sar

Katlama, metin satırlarını belirli bir genişlikte bölme işlemidir. Diğer komutlarımız gibi, `fold` da bir veya daha fazla metin dosyasını veya standart girişi kabul eder. `Fold'a` basit bir metin akışı gönderirsek, nasıl çalıştığını görebiliriz.

```
[me@linuxbox ~]$ echo "Hızlı kahverengi tilki tembel köpeğin üzerinden atlar."
|  fold  -w  12
Hızlı  kahverengi
tilki     tembel
köpeğin
üzerinden atlar.
```

Burada `fold'un` nasıl çalıştığını görüyoruz. `Echo` komutu ile gönderilen metin, `-w` seçeneği ile belirtilen parçalara bölünür. Bu örnekte, satır genişliğini 12 karakter olarak belirledik. Genişlik belirtilmezse, varsayılan değer 80 karakterdir. Satırların kelime sınırlarına bakılmaksızın nasıl bölündüğüne dikkat edin. `-s` seçeneğinin eklenmesi, `fold'un` satır genişliğine ulaşılmadan önce son kullanılabilir boşlukta satırı bölmesine neden olur.

```
[me@linuxbox ~]$ echo "Hızlı kahverengi tilki tembel köpeğin üzerinden atlar."
| fold -w 12 -s
Hızlı kahverengi
tilki tembel
köpeğin
üzerinden
atlar.
```

`fmt` – Basit Bir Metin Biçimlendirici

`fmt` programı da metni katlar, ayrıca çok daha fazlasını yapar. Dosyaları veya standart girişi kabul eder ve metin akışında paragraf biçimlendirme gerçekleştirir. Temel olarak, boş satırları ve girintileri koruyarak metindeki satırları doldurur ve birleştirir.

Bunu göstermek için biraz metne ihtiyacımız var. `fmt` bilgi sayfasından biraz alalım.

```
    `fmt', belirtilen FILE argümanlarından (veya hiçbiri belirtilmezse standart girdiden)
okur ve standart çıktıya yazar.

    Varsayılan olarak, boş satırlar, kelimeler arasındaki boşluklar ve girintiler çıktıda
korunur; farklı girintilere sahip ardışık giriş satırları birleştirilmez; sekmeler girişte
genişletilir ve çıktıda eklenir.

    `fmt', cümlenin sonunda satır sonu yapmayı tercih eder ve cümlenin ilk kelimesinden sonra
veya son kelimesinden önce satır sonu yapmaktan kaçınır. "Cümle sonu", paragrafın sonu veya
`.?!' ile biten ve ardından iki boşluk veya satır sonu gelen bir kelime olarak tanımlanır;
aradaki parantezler veya tırnak işaretleri dikkate alınmaz.  TeX gibi, `fmt' de satır sonlarını
seçmeden önce tüm "paragrafları" okur; algoritma, Donald E. Knuth ve Michael F. Plass'ın
"Breaking Paragraphs Into Lines" (Paragrafları Satırlara Bölmek) adlı makalesinde verdiği
algoritmanın bir çeşididir.
`Software--Practice & Experience' 11, 11 (Kasım 1981), 1119-1184.
```

Bu metni metin düzenleyicimize kopyalayalım ve dosyayı `fmt-info.txt` olarak kaydedelim. Şimdi, bu metni 50 karakter genişliğinde bir sütuna sığacak şekilde yeniden biçimlendirmek istediğimizi varsayalım. Bunu, dosyayı `fmt` ve `-w` seçeneği ile işleyerek yapabiliriz.

```
[me@linuxbox ~]$ fmt -w 50 fmt-info.txt | head
    `fmt' belirtilen FILE argümanlarından (veya
hiçbiri belirtilmezse standart girdiden) okur ve standart çıktıya yazar.

    Varsayılan olarak, boş satırlar, kelimeler arasındaki boşluklar
    ve girintiler
çıkışta korunur; farklı girintilere sahip ardışık giriş satırları
birleştirilmez; sekmeler girişte genişletilir ve çıkışta eklenir.
```

Bu, pek de hoş olmayan bir sonuç. Belki de bu metni okumalıyız, çünkü neler olduğunu açıklıyor

neler olduğunu açıklıyor.

> *Varsayılan olarak, boş satırlar, kelimeler arasındaki boşluklar ve girintiler çıktıda korunur; farklı girintilere sahip ardışık giriş satırları birleştirilmez; sekmeler girişte genişletilir ve çıktıda eklenir.*

Yani, `fmt` ilk satırın girintisini koruyor. Neyse ki, `fmt` bunu düzeltmek için bir seçenek sunuyor.

```
[me@linuxbox ~]$ fmt -cw 50 fmt-info.txt
    `fmt' belirtilen FILE argümanlarından (veya hiçbiri
belirtilmezse standart girdiden) okur ve standart çıktıya yazar.

    Varsayılan olarak, boş satırlar, kelimeler arasındaki boşluklar
ve girintiler çıktıda korunur; farklı girintilere sahip ardışık giriş
satırları birleştirilmez; sekmeler girişte genişletilir ve çıktıda
eklenir.

    `fmt', cümlenin sonunda satır sonu yapmayı tercih eder ve
cümlenin ilk kelimesinden sonra veya cümlenin son kelimesinden
önce satır sonu yapmaktan kaçınır.
"Cümle sonu", paragrafın sonu veya `.?!' ile biten ve
ardından iki boşluk veya satır sonu gelen, aradaki parantez
veya tırnak işaretlerini yok sayan kelime olarak tanımlanır.
parantez veya tırnak işaretleri göz ardı edilir. TeX gibi,
`fmt' satır sonlarını seçmeden önce tüm "paragrafları" okur;
algoritma, Donald E. Knuth ve Michael F. Plass'ın "Breaking
Paragraphs Into Lines" adlı eserinde
Donald E. Knuth ve Michael F. Plass'ın "Breaking Paragraphs
Into Lines" (Paragrafları Satırlara Bölmek) adlı makalesinde
`Software--Practice & Experience' 11, 11
(Kasım 1981), 1119-1184'te verilen algoritmanın bir varyantıdır.
```

Bu çok daha iyi. `-c` seçeneğini ekleyerek, istediğimiz sonucu elde ettik.

`fmt`, Tablo 21-3'te açıklanan bazı ilginç seçeneklere sahiptir.

Tablo 21-3: `fmt` Seçenekleri

Seçenek	Açıklama
`-c`	*Taç kenar boşluğu* modunda çalışır. Bu, paragrafin ilk iki satırının girintisini korur.

	Paragrafın sonraki satırları, ikinci satırın girintisiyle hizalanır.
`-p` *`string`*	Yalnızca önek *string* ile başlayan satırları biçimlendirir. Biçimlendirme işleminden sonra, *string* içeriği yeniden biçimlendirilen her satırın başına eklenir. Bu seçenek, kaynak kod yorumlarındaki metni biçimlendirmek için kullanılabilir. Örneğin, yorumları belirtmek için "#" karakterini kullanan herhangi bir programlama dili veya yapılandırma dosyası, yalnızca yorumların biçimlendirilmesi için `-p '# '` belirtilerek biçimlendirilebilir. Aşağıdaki örneği inceleyin.
`-s`	Yalnızca bölme modu. Bu modda, satırlar yalnızca belirtilen sütun genişliğine sığacak şekilde bölünür. Kısa satırlar, satırları doldurmak için birleştirilmez. Bu mod, birleştirmenin istenmediği kod gibi metinleri biçimlendirirken kullanışlıdır.
`-u`	Tek tip boşluk bırakma. Bu, metne geleneksel "daktilo stili" biçimlendirme uygular. Bu, kelimeler arasında tek boşluk ve cümleler arasında iki boşluk anlamına gelir. Bu mod, "hizalama"yı, yani metnin sol ve sağ kenarlarında hizalamayı zorlamak için boşluklarla doldurulmuş metni kaldırmak için kullanışlıdır.
`-w` *`genişlik`*	Metni, karakter genişliğindeki bir sütuna sığacak şekilde biçimlendirir. Varsayılan değer 75 karakterdir. Not: `fmt`, satır dengelemesi için belirtilen genişlikten biraz daha kısa satırlar biçimlendirir.

`-p` seçeneği özellikle ilginçtir. Bu seçenekle, biçimlendirilecek satırların tümü aynı karakter dizisiyle başlıyorsa, bir dosyanın seçilen bölümlerini biçimlendirebiliriz. Birçok programlama dili, yorumun başlangıcını belirtmek için pound işareti (#) kullanır ve bu nedenle bu seçenek kullanılarak biçimlendirilebilir. Yorum kullanan bir programı simüle eden bir dosya oluşturalım.

```
[me@linuxbox ~]$ cat > fmt-code.txt
# Bu dosya yorum içeren kodları içerir.

# Bu satır bir yorumdur.
# Ardından başka bir yorum satırı geliyor. # Ve
bir tane daha.

Bu ise bir kod satırıdır. Ve bir başka kod satırı.
Ve bir tane daha.
```

Örnek dosyamızda, "# " (bir # ve ardından bir boşluk) dizesiyle başlayan yorumlar ve bu dizeyi içermeyen "kod" satırları bulunmaktadır. Şimdi, `fmt komutunu` kullanarak yorumları biçimlendirebilir ve kodu olduğu gibi bırakabiliriz.

```
[me@linuxbox ~]$ fmt -w 50 -p '# ' fmt-code.txt # Bu dosya
yorumlar içeren kodlar içerir.

# Bu satır bir yorumdur. Ardından başka bir # yorum satırı
gelir. Ve bir tane daha.

Bu ise bir kod satırıdır. Ve bir başka kod satırı.
Ve bir tane daha.
```

Bitişik yorum satırlarının birleştirildiğini, boş satırların ve belirtilen önekle başlamayan satırların korunduğunu unutmayın.

`pr` – Yazdırma için Metni Biçimlendirme

`pr` programı metni *sayfalandırmak* için kullanılır. Metin yazdırılırken, her sayfa için üst ve alt kenar boşluğu sağlamak amacıyla çıktı sayfalarını birkaç satır boşlukla ayırmak genellikle tercih edilir. Ayrıca, bu boşluk her sayfaya üstbilgi ve altbilgi eklemek için kullanılabilir.

`distros.txt` dosyamızı bir dizi kısa sayfaya biçimlendirerek `pr'yi` göstereceğiz (sadece ilk iki sayfa gösterilmektedir).

```
[me@linuxbox ~]$ pr -l 15 -w 65 distros.txt



2025-12-11 18:27                        distros.txt                        Sayfa 1



SUSE        10.2            12/07/2006
Fedora      10              25/11/2008
SUSE        11.0            19.06.2008
Ubuntu      8.04            24.04.2008
Fedora      8               08.11.2007
```

```
2025-12-11 18:27                          distros.txt                          Sayfa 2




SUSE        10.3            10/04/2007
Ubuntu      6.10            26/10/2006
Fedora      7               31.05.2007
Ubuntu      7.10            18/10/2007
Ubuntu      7.04            19.04.2007
```

Bu örnekte, 65 sütun genişliğinde ve 15 satır uzunluğunda bir "sayfa" tanımlamak için `-l` seçeneği (sayfa uzunluğu) ve `-w` seçeneği (sayfa genişliği) kullanıyoruz. `pr`, `distros.txt` dosyasının içeriğini sayfalara ayırır, her sayfayı birkaç satır boşlukla ayırır ve dosya değiştirilme zamanını, dosya adını ve sayfa numarasını içeren varsayılan bir başlık oluşturur. `pr` programı, sayfa düzenini kontrol etmek için birçok seçenek sunar. Bunları 22. bölümde inceleyeceğiz.

`printf` – Verileri Biçimlendirme ve Yazdırma

Bu bölümdeki diğer komutların aksine, `printf` komutu boru hatları için kullanılmaz (standart girişi kabul etmez) ve komut satırında sıkça kullanılmaz (çoğunlukla komut dosyalarında kullanılır). Peki neden önemlidir? Çünkü çok yaygın olarak kullanılır.

`printf` ("formatlanmış yazdır" ifadesinden gelir), orijinal olarak C programlama dili için geliştirilmiştir ve kabuk dahil birçok programlama dilinde uygulanmıştır. Aslında, `bash'ta printf` yerleşik bir komuttur.

`printf` şu şekilde çalışır:

`printf [-v` *`var`*`] "format"` *`argümanları`*

Komuta, bir format açıklaması içeren bir dize verilir ve bu açıklama daha sonra bir argüman listesine uygulanır. Biçimlendirilmiş sonuç, `-v` seçeneği belirtilmedikçe standart çıktıya gönderilir; -v seçeneği belirtildiğinde ise biçimlendirilmiş sonuç *`var`* değişkeninde saklanır. İşte `printf'in` kullanımına ilişkin basit bir örnek:

```
[me@linuxbox ~]$ printf "Dizgiyi biçimlendirdim: %s\n" foo Dizgiyi
biçimlendirdim: foo
```

İşte yine `-v` seçeneği kullanılarak, sonuç standart çıktıya gönderilmek yerine bir değişkene yerleştirilmiştir:

```
[me@linuxbox ~]$ printf -v result "Dizgiyi biçimlendirdim: %s\n" foo [me@linuxbox ~]$ echo
"$result"
Dizgiyi biçimlendirdim: foo
```

Biçim dizesi, gerçek metinler (örneğin "Dizgiyi biçimlendirdim:"), kaçış dizileri (örneğin `\n`, satır sonu karakteri) ve % karakteriyle başlayan, *dönüştürme özellikleri* olarak adlandırılan diziler içerebilir. Yukarıdaki örnekte, dönüştürme özelliği `%s`, "foo" dizisini biçimlendirmek ve komutun çıktısına yerleştirmek için kullanılır. İşte tekrar:

```
[me@linuxbox ~]$ printf "I formatted '%s' as a string.\n" foo 'foo' dizisini
biçimlendirdim.
```

Gördüğümüz gibi, `%s` dönüşüm belirtmeni komutun çıktısında "foo" dizesi ile değiştirilmiştir. `s` dönüşümü, dize verilerini biçimlendirmek için kullanılır. Diğer veri türleri için başka belirtmenler de vardır. Tablo 21-4, yaygın olarak kullanılan veri türlerini listeler.

Tablo 21-4: Yaygın `printf` Veri Türü Belirleyicileri

Özellik	Açıklama
`d`	Bir sayıyı işaretli ondalık tamsayı olarak biçimlendirir.
`f`	Bir kayan nokta sayısını biçimlendirir ve görüntüler.
`o`	Bir tamsayıyı sekizlik sayı olarak biçimlendirir.
`s`	Bir dizgiyi biçimlendirir.
`x`	Bir tamsayıyı, gerektiğinde küçük harf `a` ile `f` kullanarak onaltılık sayı olarak biçimlendirir. kullanarak onaltılık sayı olarak biçimlendirin.
`X`	`x` ile aynıdır, ancak büyük harfler kullanılır.
`%`	Gerçek `%` sembolünü yazdırır (yani, `%%`) belirtir.

`380` dizgesinde her bir dönüştürme belirtecinin etkisini göstereceğiz.

```
[me@linuxbox ~]$ printf "%d, %f, %o, %s, %x, %X\n" 380 380 380 380
```

```
380 380
380, 380.000000, 574, 380, 17c, 17C
```

Altı dönüştürme belirteci belirlediğimiz için,
`printf işlevinin` işlemesi için altı argüman sağlamalıyız. Altı sonuç, her bir belirleyicinin etkisini gösterir.

Dönüştürme belirtecine, çıktısını ayarlamak için birkaç isteğe bağlı bileşen eklenebilir. Tam bir dönüştürme belirtimi aşağıdakilerden oluşabilir:

`%[`*`bayraklar`*`][`*`genişlik`*`][.`*`kesinlik`*`]dönüştürme_belirteci`

Birden fazla isteğe bağlı bileşen kullanıldığında, doğru şekilde yorumlanabilmesi için daha önce belirtilen sırayla görünmelidir. Tablo 21-5 her birini açıklamaktadır.

Tablo 21-5: `printf` Dönüştürme Belirteci Bileşenleri

Bileşen	Açıklama
`bayraklar`	Beş farklı bayrak vardır:
	`#` : Çıktı için "alternatif format" kullanın. Bu, veri türüne göre değişir. `o` (sekizlik sayı) dönüştürme için, çıktının başına `0` eklenir. `x` ve `X` (onaltılık sayı) dönüşümleri için, çıktının başına sırasıyla `0x` veya `0X` eklenir.
	`0` (sıfır): Çıktıyı sıfırlarla doldurun. Bu, alanın `000380` gibi baştaki sıfırlarla doldurulacağı anlamına gelir.
	`-` (tire): Çıktıyı sola hizalayın. Varsayılan olarak, `printf` çıktıyı sağa hizalar.
	‘ ’ (boşluk): Pozitif sayılar için başlangıçta boşluk oluşturur.
	`+` (artı işareti): Pozitif sayıları işaretler. Varsayılan olarak, `printf` yalnızca negatif sayıları işaretler.
`genişlik`	Minimum alan genişliğini belirten bir sayı.
`.precision`	Kayan noktalı sayılar için, ondalık noktadan sonra çıktılacak basamak sayısını belirtin. Dize dönüştürme için, *hassasiyet* çıktılacak karakter sayısını belirtir.

Tablo 21-6, farklı biçimlerin kullanımına ilişkin bazı örnekleri listeler.

Tablo 21-6: `printf` Dönüştürme Özellikleri Örnekleri

Argüman	Biçim	Sonuç	Not
`380`	`%d`	`380`	Bir tamsayının basit biçimlendirilmesi.
`380`	`%#x`	`0x17c`	"Alternatif biçim" bayrağı kullanılarak onaltılık sayı olarak biçimlendirilmiş tamsayı.
`380`	`%05d`	`00380`	Başında sıfırlar (dolgu) ve minimum beş karakterlik alan genişliği ile biçimlendirilmiş tamsayı.
`380`	`%05.5f`	`380.0000`	Dolgu ve beş ondalık basamak hassasiyeti ile kayan nokta sayısı olarak biçimlendirilmiş sayı. Belirtilen minimum alan genişliği (5) biçimlendirilmiş sayının gerçek genişliğinden daha az olduğundan, dolgu hiçbir etkiye sahip değildir.
`380`	`%010.5f`	`0380`	Minimum alan genişliğini 10'a yükselterek, dolgu artık görünür hale geldi.
`380`	`%+d`	`+380`	+ işareti pozitif bir sayıyı belirtir.
`380`	`%-d`	`380`	- bayrağı biçimlendirmeyi sola hizalar.
`abcdefghijk`	`%5s`	`abcdefghijk`	Minimum alan genişliği ile biçimlendirilmiş bir dize.
`abcdefghijk`	`%.5s`	`abcde`	Bir dizgiye hassasiyet uygulandığında, dizgi kesilir.

Yine, `printf` çoğunlukla komut satırında doğrudan değil, tablo verilerini biçimlendirmek için kullanılan komut dosyalarında kullanılır. Ancak yine de çeşitli biçimlendirme sorunlarını çözmek için nasıl kullanılabileceğini gösterebiliriz. İlk olarak, sekme karakterleriyle ayrılmış bazı alanları çıktı olarak alalım.

```
[me@linuxbox ~]$ printf "%s\t%s\t%s\n" str1 str2 str3
str1      str2      str3
```

`\t` (sekme için kaçış dizisi) ekleyerek istenen etkiyi elde ediyoruz. Aşağıda, düzgün biçimlendirilmiş bazı sayılar verilmiştir:

```
[me@linuxbox ~]$ printf "Satır: %05d %15.3f Sonuç: %+15d\n" 1071 3.14156295 32589
Satır: 01071                   3,142 Sonuç:                   +32589
```

Bu, minimum alan genişliğinin alanların aralıklarına etkisini gösterir. Ya da küçük bir web sayfasını biçimlendirmeye ne dersiniz?

```
[me@linuxbox ~]$ printf "<html>\n\t<head>\n\t\t<title>%s</title>\n
\t</head>\n\t<body>\n\t\t<p>%s</p>\n\t</body>\n</html>\n" "Sayfa Başlığı" "Sayfa İçeriği"
<html>
     <head>
           <title>Sayfa Başlığı</title>
     </head>
     <body>
           <p>Sayfa İçeriği</p>
     </body>
</html>
```

Belge Biçimlendirme Sistemleri

Şimdiye kadar basit metin biçimlendirme araçlarını inceledik. Bunlar küçük, basit görevler için iyidir, peki ya daha büyük işler? Unix'in teknik ve bilimsel kullanıcılar arasında popüler bir işletim sistemi haline gelmesinin nedenlerinden biri (her türlü yazılım geliştirme için güçlü bir çoklu görev, çoklu kullanıcı ortamı sağlamasının yanı sıra), birçok türde belge, özellikle bilimsel ve akademik yayınlar üretmek için kullanılabilecek araçlar sunmasıdır. Aslında, GNU belgelerinde de açıklandığı gibi, belge hazırlama Unix'in geliştirilmesinde önemli bir rol oynamıştır.

> *UNIX'in ilk sürümü, Bell Labs'ta bulunan bir PDP-7 üzerinde geliştirildi. 1971'de geliştiriciler, işletim sistemi üzerinde daha fazla çalışmak için bir PDP-11 almak istediler. Bu sistemin maliyetini haklı çıkarmak için, AT&T patent bölümü için bir belge biçimlendirme sistemi uygulayacaklarını önerdiler. Bu ilk biçimlendirme programı, J. F. Ossanna tarafından yazılan McIllroy'un "roff" programının yeniden uygulanmasıydı.*

.

Bu alanda iki ana belge biçimlendirici ailesi hakimdir: `nroff` ve `troff` dahil olmak üzere orijinal roff programından türeyenler ve Donald Knuth'un TEX (tek olarak okunur) dizgi sistemine dayananlar. Evet, ortadaki "E" harfi ismin bir parçasıdır.

"Roff" adı, "Senin için bir kopya çıkaracağım" gibi "çıkarmak" teriminden türemiştir. `Nroff` programı, karakter terminalleri ve daktilo tarzı yazıcılar gibi tek aralıklı yazı tiplerini kullanan cihazlara çıktı vermek üzere belgeleri biçimlendirmek için kullanılır. Piyasaya sürüldüğü dönemde, bu neredeyse bilgisayarlara bağlı tüm yazıcıları kapsıyordu. Daha sonra geliştirilen `troff` programı, ticari baskı için "kamera hazır" tip üretmek için kullanılan cihazlar olan *dizgi makinelerine* çıktı vermek üzere belgeleri biçimlendirir. Günümüzde çoğu bilgisayar yazıcısının dizgi makinelerinin çıktısını simüle etme özelliği vardır. `roff` ailesi, belgelerin bazı bölümlerini hazırlamak için kullanılan başka programları da içerir. Bunlar arasında `eqn` (matematiksel denklemler için) ve `tbl` (tablolar için) bulunur.

TEX sistemi (kararlı haliyle) ilk olarak 1989'da ortaya çıktı ve bir dereceye kadar, dizgi makinesi çıktısı için tercih edilen araç olarak `troff'un` yerini aldı. TEX'i burada ele almayacağız, hem karmaşıklığı nedeniyle (bu konuda kitaplar yazılmıştır) hem de çoğu modern Linux sisteminde varsayılan olarak yüklü olmadığı için.

İpucu: TEX'i yüklemek isteyenler, çoğu dağıtım deposunda bulunan `texlive` paketini ve LyX grafiksel içerik düzenleyicisini inceleyebilirler.

`groff`

`groff`, `troff'un` GNU uygulamasını içeren bir program paketidir. Ayrıca `nroff` ve `roff` ailesinin geri kalanını taklit etmek için kullanılan bir komut dosyası da içerir.

`roff` ve onun türevleri biçimlendirilmiş belgeler oluşturmak için kullanılır, ancak bunu modern kullanıcılar için oldukça yabancı bir şekilde yaparlar. Günümüzde çoğu belge, bir belgenin hem yazımını hem de mizanpajını tek bir adımda gerçekleştirebilen kelime işlemcileri kullanılarak üretilmektedir. Grafiksel kelime işlemcilerin ortaya çıkmasından önce, belgeler genellikle iki aşamalı bir süreçte üretilirdi: yazım için bir metin düzenleyici ve biçimlendirme için `troff` gibi bir işlemci kullanılırdı. Biçimlendirme programı için talimatlar, bir işaretleme dili kullanılarak oluşturulan metne gömülürdü. Bu sürecin modern benzeri, bir tür metin düzenleyicisi kullanılarak oluşturulan ve ardından web tarayıcısı tarafından HTML'yi işaretleme dili olarak kullanarak nihai sayfa düzenini tanımlayan web sayfasıdır.

Groff'un tamamını ele almayacağız, çünkü onun işaretleme dilinin birçok öğesi tipografinin oldukça gizemli ayrıntılarıyla ilgilidir. Bunun yerine, yaygın olarak kullanılan *makro paketlerinden* birine odaklanacağız. Bu makro paketleri, düşük seviyeli komutlarının çoğunu, groff'u kullanmayı çok daha kolay hale getiren daha küçük bir yüksek seviyeli komutlar kümesine yoğunlaştırır.

Bir an için, mütevazı man sayfasını ele alalım. Bu sayfa, /usr/share/man dizininde gzip sıkıştırılmış metin dosyası olarak bulunur. Sıkıştırılmamış içeriğini incelersek, aşağıdakileri görürüz (bölüm 1'deki ls için man sayfası gösterilmektedir):

```
[me@linuxbox ~]$ zcat /usr/share/man/man1/ls.1.gz | head
.\" BU DOSYAYI DEĞİŞTİRMEYİN! Bu dosya help2man 1.47.3 tarafından oluşturulmuştur.
.TH LS "1" "Ocak 2025" "GNU coreutils 8.28" "Kullanıcı Komutları"
.SH AD
ls \- dizin içeriğini listeler
.SH ÖZET
.B ls
[\fI\,OPTION\/\fR]... [\fI\,FILE\/\fR]...
.SH AÇIKLAMA
.\" Buraya ek açıklamalar ekleyin
.PP
```

Normal sunumundaki man sayfasıyla karşılaştırıldığında, işaretleme dili ile sonuçları arasındaki korelasyonu görmeye başlayabiliriz.

```
[me@linuxbox ~]$ man ls | head
LS(1)                            Kullanıcı Komutları                                LS(1)



AD
          ls - dizin içeriğini listeler

ÖZET
          ls [SEÇENEK]... [DOSYA]...
```

Bunun ilgi çekici olmasının nedeni, man sayfalarının man-doc makro paketi kullanılarak groff tarafından görüntülenmesidir. Aslında, man komutunu aşağıdaki boru hattıyla simüle edebiliriz:

```
[me@linuxbox ~]$ zcat /usr/share/man/man1/ls.1.gz | groff -mandoc -T
```

```
ascii | head
LS(1)                                Kullanıcı Komutları                                LS(1)



AD
          ls - dizin içeriğini listeler

ÖZET
          ls [SEÇENEK]... [DOSYA]...
```

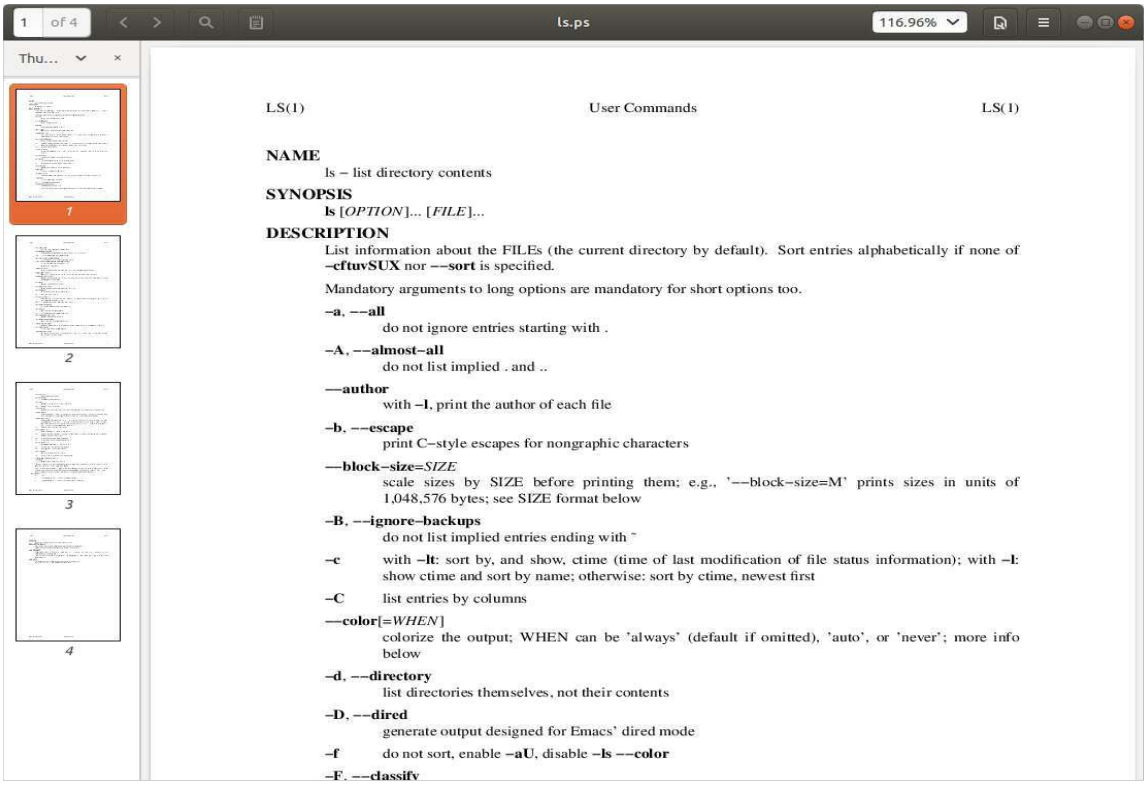
Burada, `mandoc` makro paketini ve ASCII için çıktı sürücüsünü belirtmek üzere seçenekleri ayarlanmış `groff` programını kullanıyoruz. `groff`, çeşitli biçimlerde çıktı üretebilir. Biçim belirtilmezse, varsayılan olarak PostScript çıktısı üretilir.

```
[me@linuxbox ~]$ zcat /usr/share/man/man1/ls.1.gz | groff -mandoc | head
%!PS-Adobe-3.0
%%Creator: groff sürüm 1.18.1
%%CreationDate: Thu Feb 5 13:44:37 2025
%%DocumentNeededResources: font Times-Roman
%%+ font Times-Bold
%%+ font Times-Italic
%%Belge Sağlanan Kaynaklar: procset grops 1.18 1
%%Sayfa Sayısı: 4
%%Sayfa Sırası: Artan
%%Yönelim: Dikey
```

Önceki bölümde PostScript'ten kısaca bahsetmiştik ve bir sonraki bölümde tekrar bahsedeceğiz. PostScript, basılı bir sayfanın içeriğini dizgi makinesi benzeri bir cihaza tanımlamak için kullanılan bir sayfa tanımlama dilidir. Komutumuzun çıktısını alıp bir dosyaya kaydedersek (`Desktop` dizini olan bir grafik masaüstü kullandığımızı varsayarsak), çıktı dosyasının simgesi masaüstünde görünmelidir.

```
[me@linuxbox ~]$ zcat /usr/share/man/man1/ls.1.gz | groff -mandoc >
~/Desktop/ls.ps
```

Simgeye çift tıklayarak, bir sayfa görüntüleyici başlatılmalı ve Şekil 5'te gösterildiği gibi dosyayı işlenmiş biçiminde göstermelidir.



Şekil 4: GNOME'da bir sayfa görüntüleyici ile PostScript çıktısını görüntüleme

Gördüğümüz şey, `ls` için güzelce dizilmiş bir man sayfası! Aslında, bu komutla Post-Script dosyasını *Taşınabilir Belge Formatı* (PDF) dosyasına dönüştürmek mümkündür:

```
[me@linuxbox ~]$ ps2pdf ~/Desktop/foo.ps ~/Desktop/ls.pdf
```

`ps2pdf` programı, yazdırmayı destekleyen çoğu Linux sisteminde yüklü olan `ghostscript` paketinin bir parçasıdır.

İpucu: Linux sistemleri genellikle dosya formatı dönüştürme için birçok komut satırı programı içerir. Bu programlar genellikle *format2format* kuralına göre adlandırılır. Bunları tanımlamak için `ls /usr/bin/*[[:alpha:]]2[[:alpha:]]*` komutunu kullanmayı deneyin. Ayrıca *formattoformat* adlı programları da aramayı deneyin.

`Groff` ile son alıştırmamızda, eski dostumuz `distros.txt dosyasını` tekrar ele alacağız. Bu sefer, listemizi dizgiye hazırlamak için tabloları biçimlendirmek için kullanılan `tbl` programını kullanacağız.

Linux dağıtımları. Bunu yapmak için, daha önce yazdığımız `sed` betiğini kullanarak `groff'a` besleyeceğimiz metin akışına işaretleme ekleyeceğiz.

İlk olarak, `tbl'nin` gerektirdiği gerekli işaretleme öğelerini (`groff'ta` *re-quests* olarak adlandırılır) eklemek için `sed` betiğimizi değiştirmemiz gerekiyor. Bir metin düzenleyici kullanarak `distros.sed` dosyasını aşağıdaki şekilde değiştireceğiz:

```
# Linux dağıtımları raporu oluşturmak için sed komut dosyası

1 i\
.TS\
merkez kutusu;\ cb
s s\
cb cb cb\ 1
n c.\
Linux Dağıtımları Raporu\
=\


Ad          Sürüm         Yayınlanma
_                         Tarihi\
s/\([0-9]\{2\}\)\/\([0-9]\{2\}\)\/\([0-9]\{4\}\)$/\3-\1-\2/
$ a\
.TE
```

Komut dosyasının düzgün çalışması için, `Name Version Released` kelimelerinin boşluklarla değil sekmelerle ayrıldığına dikkat edilmesi gerekir. Ortaya çıkan dosyayı `distros-tbl.sed` olarak kaydedeceğiz. `tbl,` tabloyu başlatmak ve bitirmek için `.TS` ve `.TE` isteklerini kullanır. `.TS` isteğini takip eden satırlar, tablonun genel özelliklerini tanımlar. Örneğimizde, tablo sayfada yatay olarak ortalanmış ve bir kutu ile çevrilidir. Tanımın geri kalan satırları, her bir tablo satırının düzenini tanımlar. Şimdi, rapor oluşturma boru hattımızı yeni `sed` komut dosyası ile tekrar çalıştırırsak, aşağıdakileri elde ederiz:

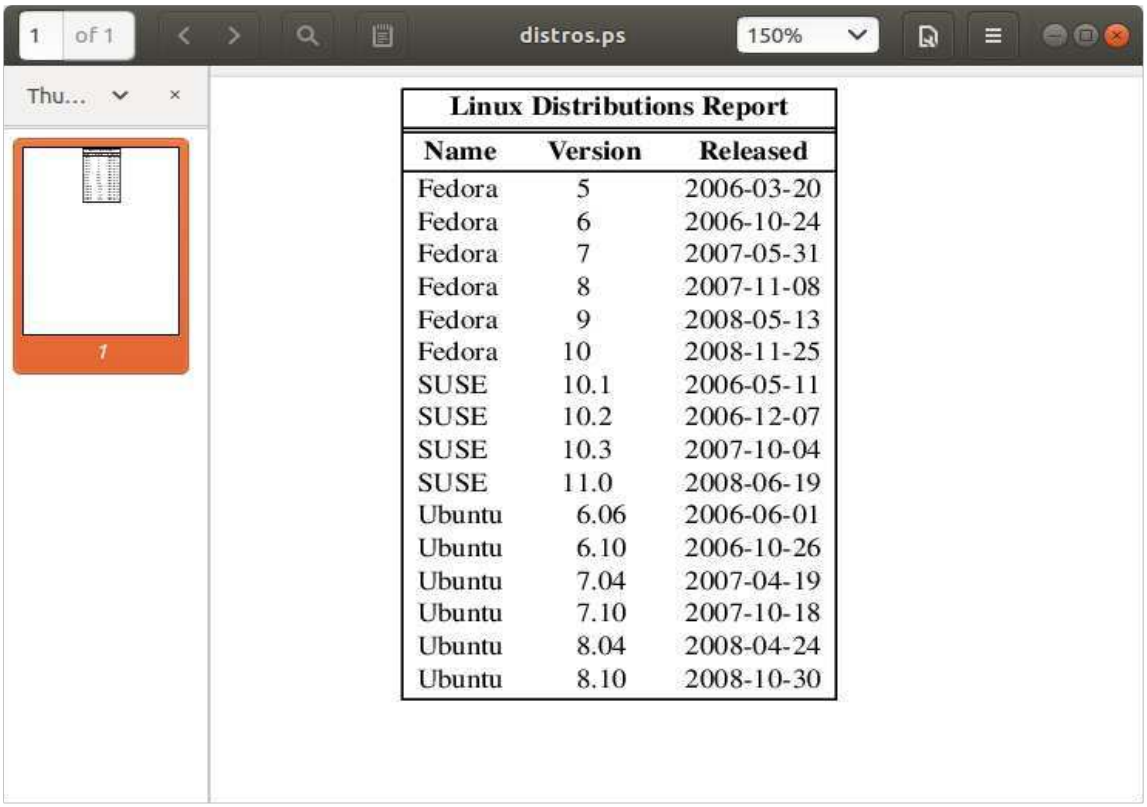
```
[me@linuxbox ~]$ sort -k 1,1 -k 2n distros.txt | sed -f distros-tbl
.sed | groff -t -T ascii
                      +------------------------------------------------------------------+
                      | Linux Dağıtımları Raporu                                         |
                      |Fedora          5               2006-03-20  |
                      |Fedora          6               2006-10-24  |
                      |Fedora          7            31 Mayıs 2007  |
```

```
|Fedora          8              2007-11-08  |
|Fedora          9              2008-05-13  |
|Fedora         10              2008-11-25  |
|SUSE           10.1            2006-05-11  |
|SUSE           10.2            07.12.2006  |
|SUSE           10.3            04.10.2007  |
|SUSE           11.0            19.06.2008  |
|Ubuntu            6.06         01.06.2006  |
|Ubuntu            6.10         2006-10-26  |
|Ubuntu            7.04         19.04.2007  |
|Ubuntu            7.10         2007-10-18  |
|Ubuntu            8.04         2008-04-24  |
|Ubuntu            8.10         30.10.2008  |
+-------------------------------------------------------------------------+
```

`groff'a -t` seçeneği eklenerek metin akışının `tbl` ile işlenmesi sağlanır. Benzer şekilde, `-T` seçeneği varsayılan çıktı ortamı olan PostScript yerine ASCII'ye çıktı vermek için kullanılır.

Çıktı biçimi, terminal ekranı veya daktilo tarzı yazıcının yetenekleriyle sınırlıysak, bekleyebileceğimiz en iyi biçimdir. PostScript çıktısını belirleyip çıktıyı grafiksel olarak görüntülersek, Şekil 6'da gösterildiği gibi çok daha tatmin edici bir sonuç elde ederiz.

```
[me@linuxbox ~]$ sort -k 1,1 -k 2n distros.txt | sed -f distros-tbl
.sed | groff -t > ~/Desktop/foo.ps
```



Linux Distributions Report

Name	Version	Released
Fedora	5	2006-03-20
Fedora	6	2006-10-24
Fedora	7	2007-05-31
Fedora	8	2007-11-08
Fedora	9	2008-05-13
Fedora	10	2008-11-25
SUSE	10.1	2006-05-11
SUSE	10.2	2006-12-07
SUSE	10.3	2007-10-04
SUSE	11.0	2008-06-19
Ubuntu	6.06	2006-06-01
Ubuntu	6.10	2006-10-26
Ubuntu	7.04	2007-04-19
Ubuntu	7.10	2007-10-18
Ubuntu	8.04	2008-04-24
Ubuntu	8.10	2008-10-30

Şekil 5: Bitmiş tablonun görüntülenmesi

Özet

Metin, Unix benzeri işletim sistemlerinin karakterinde çok önemli bir yer tuttuğundan, metni işlemek ve biçimlendirmek için kullanılan birçok araç olması mantıklıdır. Gördüğümüz gibi, bunlar gerçekten var! `fmt` ve `pr` gibi basit biçimlendirme araçları, kısa belgeler üreten komut dosyalarında birçok kullanım alanı bulurken, `groff` (ve benzerleri) kitap yazmak için kullanılabilir. Komut satırı araçlarını kullanarak teknik bir makale yazmayabiliriz (bunu yapan birçok kişi olsa da!), ancak bunu yapabileceğimizi bilmek iyidir.

Daha Fazla Okuma

- `groff` Kullanıcı Kılavuzu
  https://www.gnu.org/software/groff/manual/
- *nroff ile Makale Yazma* `-me` *Kullanımı*:
  https://docs.freebsd.org/44doc/usd/19.memacros/paper.pdf

- `-me` *Referans Kılavuzu*:
  https://docs.freebsd.org/44doc/usd/20.meref/paper.pdf
- Ve tabii ki, Wikipedia'daki şu makaleleri de inceleyin:
  https://en.wikipedia.org/wiki/TeX
  https://en.wikipedia.org/wiki/Donald_Knuth
  https://en.wikipedia.org/wiki/Typesetting

22 – Yazdırma

Son birkaç bölümde metinleri işledikten sonra, şimdi bu metinleri kağıda dökme zamanı geldi. Bu bölümde, dosyaları yazdırmak ve yazıcıyı kontrol etmek için kullanılan komut satırı araçlarını inceleyeceğiz. Yazdırmayı nasıl yapılandıracağımızı incelemeyeceğiz, çünkü bu dağıtımdan dağıtıma değişir ve genellikle kurulum sırasında otomatik olarak ayarlanır. Bu bölümdeki alıştırmaları yapmak için çalışan bir yazıcı yapılandırmasına ihtiyacımız olacağını unutmayın.

Aşağıdaki komutları ele alacağız:

- `pr` – Metin dosyalarını yazdırmak için dönüştürür
- `lp` / `lpr` – Dosyaları yazdır
- `a2ps` – PostScript yazıcıda yazdırmak için dosyaları biçimlendirir
- `lpstat` – Yazıcı durum bilgilerini gösterir
- `lpq` – Yazıcı kuyruğu durumunu göster
- `lprm` – Yazdırma işlerini iptal etme

Yazdırmanın Kısa Tarihi

Unix benzeri işletim sistemlerinde bulunan yazdırma özelliklerini tam olarak anlamak için önce biraz tarih öğrenmemiz gerekir. Unix benzeri sistemlerde yazdırma, işletim sisteminin başlangıcına kadar uzanır. O günlerde yazıcılar ve kullanım şekilleri bugünkünden çok farklıydı.

Karanlık Çağlarda Yazdırma

Bilgisayarlar gibi, PC öncesi dönemde yazıcılar da genellikle büyük, pahalı ve merkeziydi. 1980'lerin tipik bilgisayar kullanıcısı, uzaktaki bir bilgisayara bağlı bir terminalde çalışıyordu. Yazıcı, bilgisayarın yakınında bulunuyordu ve bilgisayar operatörlerinin gözetimindeydi.

Yazıcılar pahalı ve merkeziyken, Unix'in ilk günlerinde sıklıkla olduğu gibi, birçok kullanıcının bir yazıcıyı paylaşması yaygın bir uygulamaydı. Belirli bir kullanıcıya ait yazdırma işlerini tanımlamak için

bilgisayar destek personeli, o günün yazdırma işlerini içeren bir arabayı yükler ve bunları bireysel kullanıcılara teslim ederdi.

Karakter Tabanlı Yazıcılar

80'lerin yazıcı teknolojisi, iki açıdan bugünkünden çok farklıydı. İlk olarak, o dönemin yazıcıları neredeyse her zaman *darbeli yazıcılardı*. Darbeli yazıcılar, sayfada karakter izleri oluşturmak için kağıda bir şerit vuran mekanik bir mekanizma kullanır. O dönemin popüler teknolojilerinden ikisi, *papatya tekerleği* baskı ve *nokta vuruşlu* baskıydı.

Erken dönem yazıcıların ikinci ve daha önemli özelliği, yazıcıların cihaza özgü sabit bir karakter seti kullanmasıydı. Örneğin, papatya tekerleği yazıcı, papatya tekerleğinin yapraklarına kalıplanmış karakterleri yazdırabiliyordu. Bu, yazıcıları yüksek hızlı daktilolara çok benzetiyordu. Çoğu daktiloda olduğu gibi, tek aralıklı (sabit genişlikli) yazı tipleri kullanılarak yazdırılıyordu. Bu, her karakterin aynı genişliğe sahip olduğu anlamına geliyordu. Yazdırma, sayfanın sabit konumlarında yapılırdı ve bir sayfanın yazdırılabilir alanı sabit sayıda karakter içerirdi. Çoğu yazıcı, yatay olarak inç başına 10 karakter (CPI) ve dikey olarak inç başına altı satır (LPI) yazdırırdı. Bu şemayı kullanarak, bir ABD mektup kağıdı 85 karakter genişliğinde ve 66 satır yüksekliğindedir. Her iki tarafta küçük bir kenar boşluğu dikkate alındığında, 80 karakter bir baskı satırının maksimum genişliği olarak kabul edildi. Bu, terminal ekranlarının (ve terminal emülatörlerimizin) normalde 80 karakter genişliğinde olmasının nedenini açıklar. Tek aralıklı yazı tipi ve 80 karakter genişliğinde bir terminal kullanmak, basılı çıktının *Ne Görüyorsan Onu Alırsın* (*WYSIWYG*) görünümünü sağlar.

Veriler, yazdırılacak karakterleri içeren basit bir bayt akışında daktilo benzeri bir yazıcıya gönderilir. Örneğin, "a" harfini yazdırmak için ASCII karakter kodu 97 gönderilir. Ayrıca, düşük numaralı ASCII kontrol kodları, satır başı, satır besleme, form besleme vb. kodları kullanarak yazıcının taşıyıcısını ve kağıdı hareket ettirme imkanı sağlar. Kontrol kodlarını kullanarak, yazıcıya bir karakteri yazdırıp, geri al tuşuna basıp, karakteri tekrar yazdırarak sayfada daha koyu bir baskı izlenimi elde etmek gibi bazı sınırlı yazı tipi efektleri elde etmek mümkündür. `Nroff'u` kullanarak bir man sayfasını görüntüleyip `cat -A` komutuyla çıktıyı incelersek bunu görebiliriz.

```
[me@linuxbox ~]$ zcat /usr/share/man/man1/ls.1.gz | nroff -P -c -man
| cat -A | head
LS(1)                                Kullanıcı Komutları                                LS(1)
$
$
$ N^HNA^HAM^HME^HE$


         ls - dizin içeriğini listele$
```

```
$ S^HSY^HYN^HNO^HOP^HPS^HSI^HIS^HS$
          l^Hls^Hs [_^HO_^HP_^HT_^HI_^HO_^HN]... [_^HF_^HI_^HL_^HE]...$
```

^H (Ctrl-h) karakterleri, kalın yazı efekti oluşturmak için kullanılan geri alma tuşlarıdır. Benzer şekilde, altı çizili yazı efekti oluşturmak için kullanılan geri alma tuşu/alt çizgi dizisini de görebiliriz.

Grafik Yazıcılar

GUI'lerin geliştirilmesi, yazıcı teknolojisinde büyük değişikliklere yol açtı. Bilgisayarlar daha çok resim tabanlı ekranlara geçtikçe, yazdırma da karakter tabanlı tekniklerden grafik tekniklere geçti. Bu, sabit karakterleri yazdırmak yerine sayfanın yazdırılabilir alanının herhangi bir yerine küçük noktalar yazdırabilen düşük maliyetli lazer yazıcıların ortaya çıkmasıyla kolaylaştı. Bu, orantılı yazı tiplerinin (dizgiciler tarafından kullanılanlar gibi) ve hatta fotoğrafların ve yüksek kaliteli diyagramların yazdırılmasını mümkün kıldı.

Ancak, karakter tabanlı bir şemadan grafiksel bir şemaya geçmek, büyük bir teknik zorluk oluşturdu. Nedeni şudur: Karakter tabanlı bir yazıcı kullanarak bir sayfayı doldurmak için gereken bayt sayısı şu şekilde hesaplanabilir (sayfa başına 60 satır ve her satırda 80 karakter olduğu varsayılırsa):

60 X 80 = 4.800 bayt

Buna karşılık, 300 nokta/inç (DPI) lazer yazıcı (sayfa başına 8 x 10 inç baskı alanı olduğu varsayılırsa) şu kadar bayt gerektirir:

(8 X 300) X (10 X 300) / 8 = 900.000 bayt

Yavaş PC ağlarının çoğu, lazer yazıcıda tam bir sayfayı yazdırmak için gereken yaklaşık bir megabaytlık veriyi işleyemiyordu, bu nedenle akıllı bir icatın gerekli olduğu açıktı.

Bu icat, *sayfa tanımlama dili* (PDL) olarak ortaya çıktı. Sayfa tanımlama dili, bir sayfanın içeriğini tanımlayan bir programlama dilidir. Temel olarak, "Bu konuma git, 10 punto Helvetica ile 'a' karakterini çiz, bu konuma git..." şeklinde devam eder ve sayfadaki her şey tanımlanana kadar devam eder. İlk büyük PDL, Adobe Systems'ın *PostScript'iydi* ve bugün hala yaygın olarak kullanılmaktadır. PostScript dili, tipografi ve diğer grafik ve görüntüleme türleri için özel olarak tasarlanmış eksiksiz bir programlama dilidir. 35 standart, yüksek kaliteli yazı tipini destekler ve çalışma sırasında ek yazı tipi tanımlarını kabul etme özelliğine sahiptir. İlk başta, PostScript desteği yazıcılara entegre edilmişti. Bu, veri aktarım sorununu çözdü. Tipik bir PostScript programı, karakter tabanlı yazıcıların basit bayt akışına kıyasla çok ayrıntılı olsa da, basılı sayfanın tamamını temsil etmek için gereken bayt sayısından çok daha küçüktü.

Bir *PostScript yazıcı*, PostScript programını girdi olarak kabul ederdi. Yazıcı, kendi işlemcisini ve belleğini içerirdi (çoğu zaman yazıcıyı, bağlı olduğu bilgisayardan daha güçlü bir bilgisayar haline getirirdi) ve gelen PostScript programını okuyan ve sonuçları yazıcının dahili belleğine *aktaran*, böylece kağıda aktarılacak bit (nokta) desenini oluşturan *PostScript yorumlayıcı* adlı özel bir programı çalıştırırdı. Bir şeyi büyük bir bit desenine (*bitmap* olarak adlandırılır) dönüştürme işleminin genel adı *raster görüntü işlemcisi* (RIP)dir.

Yıllar geçtikçe, hem bilgisayarlar hem de ağlar çok daha hızlı hale geldi. Bu, RIP'in yazıcıdan ana bilgisayara taşınmasını sağladı ve bu da yüksek kaliteli yazıcıların çok daha ucuz olmasını mümkün kıldı.

Günümüzde birçok yazıcı hala karakter tabanlı akışları kabul etmektedir, ancak birçok düşük maliyetli yazıcı bunu yapmamaktadır. Bu yazıcılar, nokta olarak yazdırmak üzere bit akışı sağlamak için ana bilgisayarın RIP'sine güvenmektedir. Hala bazı PostScript yazıcılar da bulunmaktadır.

Linux ile yazdırma

Modern Linux sistemleri, yazdırma işlemlerini gerçekleştirmek ve yönetmek için iki yazılım paketi kullanır. İlki, Common Unix Printing System (CUPS) yazdırma sürücüleri ve yazdırma işi yönetimi sağlar, ikincisi ise PostScript yorumlayıcısı olan Ghostscript, RIP görevi görür.

CUPS, *yazdırma kuyrukları* oluşturarak ve sürdürerek yazıcıları yönetir. Daha önceki tarih dersinde tartıştığımız gibi, Unix yazdırma, başlangıçta birden fazla kullanıcı tarafından paylaşılan merkezi bir yazıcıyı yönetmek için tasarlanmıştı. Yazıcılar, onları besleyen bilgisayarlara kıyasla doğası gereği yavaş olduğundan, yazdırma sistemlerinin birden fazla yazdırma işini planlamak ve işleri düzenli tutmak için bir yola ihtiyacı vardır. CUPS ayrıca farklı veri türlerini (makul sınırlar içinde) tanıma ve dosyaları yazdırılabilir bir biçime dönüştürme yeteneğine sahiptir.

Yazdırma için Dosyaları Hazırlama

Komut satırı kullanıcıları olarak, çoğunlukla metin yazdırmakla ilgileniriz, ancak diğer veri biçimlerini de yazdırmak elbette mümkündür.

`pr` – Yazdırma için Metin Dosyalarını Dönüştürme

Önceki bölümde `pr komutunu` biraz incelemiştik. Şimdi, yazdırma ile birlikte kullanılan birçok seçeneğinden bazılarını inceleyeceğiz. Yazdırma tarihçesinde, karakter tabanlı yazıcıların tek aralıklı yazı tiplerini nasıl kullandığını ve bunun sonucunda satır başına sabit karakter sayısı ve sayfa başına sabit satır sayısı elde edildiğini görmüştük. `pr`, metni belirli bir sayfa boyutuna sığdırmak için kullanılır ve isteğe bağlı sayfa üstbilgileri ve kenar boşlukları da eklenebilir. Tablo 22-1, en sık kullanılan seçenekleri özetlemektedir.

Tablo 22-1: Yaygın `pr` Seçenekleri

Seçenek	Açıklama
`+first[:last]`	*first* ile başlayan ve isteğe bağlı olarak *last* ile biten bir sayfa aralığını çıkarır.
`-columns`	Sayfanın içeriğini *columns* ile belirtilen sütun sayısına göre düzenler.
`-a`	Varsayılan olarak, çok sütunlu çıktı dikey olarak listelenir. -a (across) seçeneği eklenerek içerik yatay olarak listelenir.
`-d`	Çıktıyı çift boşluklu yapar.
`-D "format"`	Sayfa başlıklarında görüntülenen tarihi *format* kullanarak biçimlendirin. Biçim dizesi açıklaması için `date` komutunun man sayfasına bakın.
`-f`	Sayfaları ayırmak için satır başı yerine sayfa başı kullanın.
`-h "başlık"`	Sayfa başlığının orta kısmında, işlenmekte olan dosyanın adı yerine *başlığı* kullanın.
`-l uzunluk`	Sayfa uzunluğunu *uzunluk* olarak ayarlayın. Varsayılan değer 66'dır (inç başına altı satırlık ABD mektup kağıdı).
`-n`	Satırları numaralandırır.
`-o ofset`	Sol kenar boşluğunu karakter genişliğinde oluştur*un*.
`-w genişlik`	Sayfa genişliğini *genişlik* olarak ayarlayın. Varsayılan değer 72'dir.

`pr` genellikle filtre olarak boru hatlarında kullanılır. Bu örnekte, `/usr/bin` dizinini listeleyecek ve `pr` kullanarak sayfalanmış, üç sütunlu bir çıktı formatlayacağız:

```
[me@linuxbox ~]$ ls /usr/bin | pr -3 -w 65 | head



2025-02-18 14:00 [                                                Sayfa 1
411toppm                apturl ar               bsd-yazma
a2p                     arecord                 bsh btcflash
a2ps                    arecordmidi             hata-arkadaşı
a2ps-lpr-wrapper        ark                     buildhash
```

Yazıcıya Yazdırma İşini Gönderme

CUPS yazdırma paketi, Unix benzeri sistemlerde tarihsel olarak kullanılan iki yazdırma yöntemini destekler. Berkeley veya LPD (Unix'in Berkeley Software Distribution sürümünde kullanılır) olarak adlandırılan bir yöntem `lpr` programını kullanırken, SysV (Unix'in System V sürümünden gelir) olarak adlandırılan diğer yöntem `lp` programını kullanır. Her iki program da kabaca aynı şeyi yapar. Birini diğerine tercih etmek kişisel bir tercihtir.

`lpr` – Dosyaları Yazdırma (Berkeley Stili)

`lpr` programı, dosyaları yazıcıya göndermek için kullanılabilir. Standart girişi kabul ettiği için, boru hatlarında da kullanılabilir. Örneğin, önceki çok sütunlu dizin listelemesinin sonuçlarını yazdırmak için şunu yapabiliriz:

```
[me@linuxbox ~]$ ls /usr/bin | pr -3 | lpr
```

Rapor, sistemin varsayılan yazıcısına gönderilecektir. Dosyayı farklı bir yazıcıya göndermek için `-P` seçeneği şu şekilde kullanılabilir:

```
lpr -P yazıcı_adı
```

Burada, *yazıcı_adı* istenen yazıcının adıdır. Sistem tarafından tanınan yazıcıların listesini görmek için şunu kullanın:

```
[me@linuxbox ~]$ lpstat -a
```

> **İpucu:** Birçok Linux dağıtımı, fiziksel yazıcıda yazdırmak yerine, dosyaları Taşınabilir Belge Biçimi (PDF) olarak çıktı veren bir "yazıcı" belirlemenize izin verir. Bu, yazdırma komutlarını denemek için kullanışlıdır. Yazıcı yapılandırma programınızı kontrol ederek bu yapılandırmayı destekleyip desteklemediğini görün. Bazı dağıtımlarda, bu özelliği etkinleştirmek için ek paketler (`cups-pdf` gibi) yüklemeniz gerekebilir.

Tablo 22-2, `lpr` için yaygın seçenekleri açıklamaktadır.

Tablo 22-2: Yaygın `lpr` Seçenekleri

Seçenek	Açıklama

`-#` *sayı*	Kopyaların sayısını *sayıya* ayarlayın.
`-p`	Her sayfayı tarih, saat, iş adı ve sayfa numarası içeren gölgeli bir üst bilgi ile yazdırın. Bu "güzel yazdırma" seçeneği, metin dosyalarını yazdırırken kullanılabilir.
`-P` *yazıcı*	Çıktı için kullanılan yazıcının adını belirtin. Yazıcı belirtilmezse, sistemin varsayılan yazıcı kullanılır.
`-r`	Yazdırma işleminden sonra dosyaları silin. Bu, geçici yazıcı çıktı dosyaları üreten programlar için yararlıdır.

`lp` – Dosyaları Yazdır (System V Stili)

`lpr` gibi, `lp` de yazdırma için dosyaları veya standart girişi kabul eder. `lpr'den` farklı olarak, farklı (ve biraz daha gelişmiş) bir seçenek setini destekler. Tablo 22-3 yaygın seçenekleri açıklamaktadır.

Tablo 22-3: Yaygın `lp` Seçenekleri

Seçenek	Açıklama
`-d` *yazıcı*	Hedefi (yazıcı) *yazıcı* olarak ayarlayın. d seçeneği belirtilmezse, sistem varsayılan yazıcı kullanılır.
`-n` *sayı*	Kopyaların sayısını *sayı* olarak ayarlayın.
`-o yatay`	Çıktıyı yatay yönlendirmeye ayarlayın.
`-o fitplot`	Dosyayı sayfaya sığacak şekilde ölçeklendirir. Bu, JPEG dosyaları gibi görüntüleri yazdırırken kullanışlıdır.
`-o` *ölçekleme=sayı*	Dosyayı *sayıya* göre ölçeklendirir. 100 değeri sayfayı doldurur. 100'den küçük değerler küçültülürken, 100'den büyük değerler dosyanın birden fazla sayfaya yazdırılmasına neden olur.
`-o` *cpi=sayı*	İnç başına çıktı karakterlerini *sayıya* ayarlayın. Varsayılan değer 10'dur.
`-o` *lpi=sayı*	İnç başına çıktı satırlarını *sayıya* ayarlayın. Varsayılan değer 6'dır.
`-P` *sayfalar*	Sayfa listesini belirtin. *Sayfalar* virgülle ayrılmış bir liste ve/veya bir aralık olarak ifade edilebilir,

Örneğin, `1,3,5,7-10`

Dizin listemizi tekrar oluşturacağız, bu sefer 12 CPI ve 8 LPI ile yarım inç sol kenar boşluğu olacak şekilde yazdıracağız. Yeni sayfa boyutunu hesaba katmak için `pr` seçeneklerini ayarlamamız gerektiğini unutmayın:

```
[me@linuxbox ~]$ ls /usr/bin | pr -4 -w 90 -l 88 | lp -o page-left=36
-o cpi=12 -o lpi=8
```

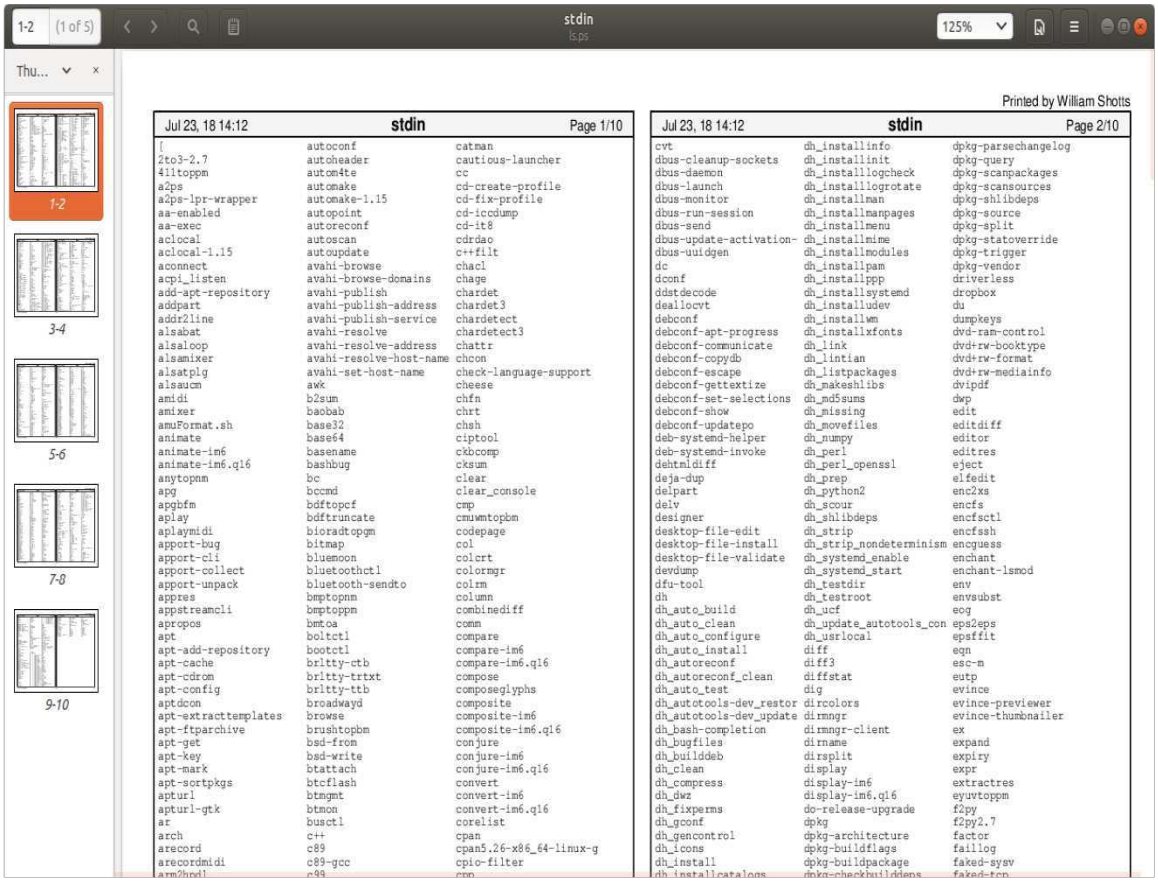
Bu boru hattı, varsayılan ayardan daha küçük bir yazı tipi kullanarak dört sütunlu bir liste oluşturur. İnç başına karakter sayısının artması, sayfaya daha fazla sütun sığdırmamızı sağlar.

Başka bir seçenek: `a2ps`

`a2ps` programı (çoğu depoda mevcuttur) ilginçtir. Adından da anlaşılabileceği gibi, bu bir format dönüştürme programıdır, ancak çok daha fazlasını da yapar. Adı orijinal olarak "ASCII to PostScript" anlamına geliyordu ve PostScript yazıcılarda yazdırmak için metin dosyalarını hazırlamak için kullanılıyordu. Ancak yıllar geçtikçe programın yetenekleri arttı ve şimdi adı "Anything to PostScript" anlamına geliyor. Adı bir format dönüştürme programı olduğunu düşündürse de, aslında bir yazdırma programıdır. Varsayılan çıktısını standart çıktı yerine sistemin varsayılan yazıcısına gönderir. Programın varsayılan davranışı, çıktının görünümünü iyileştiren bir "güzel yazıcı"dır. Bu programı masaüstümüzde bir PostScript dosyası oluşturmak için kullanıyoruz.

```
[me@linuxbox ~]$ ls /usr/bin | pr -3 -t | a2ps -o ~/Desktop/ls.ps -L 66
[stdin (düz): 6 sayfada 11 sayfa]
[Toplam: 6 sayfada 11 sayfa] `/home/me/Desktop/ ls.ps' dosyasına kaydedildi
```

Burada, `-t` seçeneğini (başlıkları ve altbilgileri atla) kullanarak akışı `pr` ile filtreliyoruz ve ardından `a2ps` ile `pr'nin` çıktı sayfalandırmasına uyacak şekilde bir çıktı dosyası (`-o` seçeneği) ve sayfa başına 66 satır (`-L` seçeneği) belirtiyoruz. Ortaya çıkan dosyayı uygun bir dosya görüntüleyici ile görüntülersek, Şekil 7'deki çıktıyı göreceğiz.



Şekil 6: a2ps çıktısını görüntüleme

Gördüğümüz gibi, varsayılan çıktı düzeni "iki yukarı" biçimindedir. Bu, iki sayfanın içeriğinin her bir kağıda yazdırılmasına neden olur. `a2ps` ayrıca güzel sayfa üstbilgileri ve altbilgileri de uygular.

`a2ps'nin` birçok seçeneği vardır. Tablo 22-4 bir özet sunmaktadır.

Tablo 22-4: `a2ps` Seçenekleri

Seçenek	Açıklama
`--center-title=metin`	Sayfa başlığını *metin* olarak ortala.
`--columns=sayı`	Sayfaları *sayı* sütunlarına ayırın. Varsayılan değer 2'dir.
`--footer=metin`	Sayfa altbilgisini *metin* olarak ayarlayın.
`--guess`	Argüman olarak verilen dosya türlerini bildirir.

	`a2ps` tüm veri türlerini dönüştürmeye ve biçimlendirmeye çalıştığından, bu seçenek belirli bir dosya verildiğinde `a2ps'nin` ne yapacağını tahmin etmek için yararlı olabilir.
`--left-footer=metin`	Sol sayfa altbilgisini *metin* olarak ayarlayın.
`--left-title=metin`	Sol sayfa başlığını *metin* olarak ayarlayın.
`--line-numbers=aralık`	Her satır *aralığı* için çıktı satırlarını numaralandır.
`--list=varsayılanlar`	Varsayılan ayarları görüntüle.
`--pages=aralık`	Aralıktaki sayfaları yazdır.
`--right-footer=metin`	Sağ sayfa alt bilgisini *metin* olarak ayarla.
`--right-title=metin`	Sağ sayfa başlığını *metin* olarak ayarlayın.
`--rows=sayı`	Sayfaları *sayı* satırlarına göre düzenleyin. Varsayılan değer 1'dir.
`-B`	Sayfa üstbilgileri yok.
`-b` *`metin`*	Sayfa başlığını *metin* olarak ayarlayın.
`-f` *`boyut`*	*Boyut* noktalı yazı tipi kullanın.
`-l` *`sayı`*	Satır başına karakter sayısını *sayıya* ayarlayın. Bu ve `-L` seçeneği (sonraki girişe bakın) `pr` gibi diğer programlarla sayfalandırılmış dosyaların sayfaya doğru şekilde sığması için kullanılabilir.
`-L` *`sayı`*	Sayfa başına satır sayısını *sayıya* ayarlayın.
`-M` *`ad`*	Medya *adını* kullanın. Örneğin, `A4`.
`-n` *`sayı`*	Her sayfanın kopya *sayısını* belirtin.
`-o` *`dosya`*	Çıktıyı *dosyaya* gönderir. *Dosya* `-` olarak belirtilirse, standart çıktı kullanılır.
`-P` *`yazıcı`*	*Yazıcıyı* kullan. Yazıcı belirtilmezse, sistem varsayılan yazıcı kullanılır.
`-R`	Dikey yönlendirme.
`-r`	Yatay yönlendirme.
`-T` *`sayı`*	Her *sayı* karakterine sekme durağı ayarla.
`-u` *`metin`*	Sayfalara *metin* içeren alt tabaka (filigran) ekleyin.

Bu sadece bir özet. `a2ps'nin` birkaç seçeneği daha var.

> **Not:** Metni PostScript'e dönüştürmek için kullanışlı olan başka bir çıktı biçimlendirici daha vardır. `Enscript` adı verilen bu program, `a2ps ile` aynı türden birçok biçimlendirme ve yazdırma işlemini gerçekleştirebilir, ancak `a2ps'den` farklı olarak yalnızca metin girdisini kabul eder.

Yazdırma İşlerini İzleme ve Denetleme

Unix yazdırma sistemleri, birden fazla kullanıcıdan gelen birden fazla yazdırma işini işlemek üzere tasarlanmıştır; CUPS de aynı amaca hizmet eder. Her yazıcıya bir *yazdırma kuyruğu* atanır ve işler yazıcıya *aktarılana* kadar bu kuyrukta bekletilir. CUPS, yazıcı durumunu ve yazdırma kuyruklarını yönetmek için kullanılan birkaç komut satırı programı sağlar. `lpr` ve `lp` programları gibi, bu yönetim programları da Berkeley ve System V yazdırma sistemlerindeki karşılık gelen programları örnek alarak tasarlanmıştır.

`lpstat` – Yazdırma Sistemi Durumunu Görüntüleme

`lpstat` programı, sistemdeki yazıcıların adlarını ve kullanılabilirliğini belirlemek için kullanışlıdır. Örneğin, hem fiziksel bir yazıcıya ("printer" adlı) hem de bir PDF sanal yazıcıya ("PDF" adlı) sahip bir sistemimiz varsa, durumlarını şu şekilde kontrol edebiliriz:

```
[me@linuxbox ~]$ lpstat -a
PDF, 08 Aralık 2024 Pazartesi 03:05:59 EST'den beri istekleri kabul ediyor printer, 24
Şubat 2025 Salı 08:43:22 EST'den beri istekleri kabul ediyor
```

Ayrıca, bu şekilde yazdırma sistemi yapılandırmasının daha ayrıntılı bir açıklamasını belirleyebiliriz:

```
[me@linuxbox ~]$ lpstat -s
sistem varsayılan hedefi: yazıcı PDF için aygıt:
cups-pdf:/
yazıcı için aygıt: ipp://print-server:631/printers/printer
```

Bu örnekte, "yazıcı"nın sistemin varsayılan yazıcı olduğunu ve "print-server" adlı bir sisteme bağlı Internet Yazdırma Protokolü (ipp://) kullanan bir ağ yazıcı olduğunu görüyoruz.

Tablo 22-5, yaygın olarak kullanılan seçenekleri listelemektedir.

Tablo 22-5: Yaygın `lpstat` Seçenekleri

Seçenek	Açıklama
`-a [yazıcı...]`	*Yazıcı* için yazıcı kuyruğunun durumunu görüntüler. Bunun, fiziksel yazıcıların durumu değil, yazıcı kuyruğunun işleri kabul etme yeteneğinin durumu olduğunu unutmayın. Hiçbir yazıcı belirtilmezse, tüm yazdırma kuyrukları gösterilir.
`-d`	Sistemin varsayılan yazıcısının adını görüntüler.
`-p [yazıcı...]`	Belirtilen *yazıcının* durumunu görüntüler. Hiçbir yazıcı belirtilmezse, tüm yazıcılar gösterilir.
`-r`	Yazdırma sunucusunun durumunu görüntüler.
`-s`	Durum özetini görüntüler.
`-t`	Tam durum raporunu görüntüler.

`lpq` – Yazıcı Kuyruğu Durumunu Görüntüleme

Yazıcı kuyruğunun durumunu görmek için `lpq` programı kullanılır. Bu, kuyruğun durumunu ve içerdiği yazdırma işlerini görüntülememizi sağlar. Aşağıda, "printer" adlı sistem varsayılan yazıcısının boş kuyruğuna bir örnek verilmiştir:

```
[me@linuxbox ~]$ lpq printer hazır
giriş yok
```

Bir yazıcı belirtmezsek (`-P` seçeneğini kullanarak), sistemin varsayılan yazıcı gösterilir. Yazıcıya bir iş gönderir ve ardından kuyruğa bakarsak, bu işin listelendiğini görürüz.

```
[me@linuxbox ~]$ ls *.txt | pr -3 | lp istek
kimliği printer-603 (1 dosya) [me@linuxbox ~]$ lpq
yazıcı hazır ve yazdırıyor Sıra    Sahibi   İş
          Dosya
aktif ben             60         (stdin)
                                                                    Toplam Boyut
                                                                    1024 bayt
```

`lprm` / `cancel` – Yazdırma İşlerini İptal Et

CUPS, yazdırma işlerini sonlandırmak ve yazdırma kuyruğundan kaldırmak için kullanılan iki program sağlar. Bunlardan biri Berkeley stili (`lprm`), diğeri ise System V (`cancel`) stili. Destekledikleri seçenekler açısından biraz farklılık gösterirler, ancak temel olarak aynı işlevi görürler. Önceki yazdırma işimizi örnek olarak kullanarak, işi şu şekilde durdurabilir ve kaldırabiliriz:

```
[me@linuxbox ~]$ cancel 603
[me@linuxbox ~]$ lpq yazıcı hazır
giriş yok
```

Her komutun, belirli bir kullanıcıya, belirli bir yazıcıya ve birden fazla iş numarasına ait tüm işleri kaldırmak için seçenekleri vardır. İlgili man sayfalarında tüm ayrıntılar bulunmaktadır.

Özet

Bu bölümde, geçmişteki yazıcıların Unix benzeri makinelerdeki yazdırma sistemlerinin tasarımını nasıl etkilediğini ve komut satırında yazdırma işlerinin zamanlamasını ve yürütülmesini kontrol etmek için ne kadar kontrolün mevcut olduğunu ve ayrıca çeşitli çıktı seçeneklerini gördük.

Daha Fazla Okuma

- PostScript sayfa betimleme dili hakkında iyi bir makale:
  https://en.wikipedia.org/wiki/PostScript
- Ortak Unix Yazdırma Sistemi (CUPS):
  https://en.wikipedia.org/wiki/CUPS
  https://openprinting.github.io/cups/
- Berkeley ve System V Yazdırma Sistemleri:
  https://en.wikipedia.org/wiki/Berkeley_printing_system
  https://en.wikipedia.org/wiki/System_V_printing_system

23 – Programları Derleme

Bu bölümde, kaynak kodunu derleyerek programların nasıl oluşturulduğuna bakacağız. Kaynak kodunun kullanılabilirliği, Linux'u mümkün kılan temel özgürlüktür. Linux geliştirmenin tüm ekosistemi, geliştiriciler arasındaki serbest bilgi alışverişine dayanır. Birçok masaüstü kullanıcısı için derleme, kaybolmuş bir sanattır. Eskiden oldukça yaygın olan derleme, günümüzde dağıtım sağlayıcıları tarafından indirilmeye ve kullanıma hazır, önceden derlenmiş ikili dosyaların bulunduğu devasa depolarla gerçekleştirilmektedir. Bu yazının yazıldığı tarihte, Debian deposu (dağıtımlar arasında en büyüklerinden biri) 68.000'den fazla paket içermektedir.

Öyleyse neden yazılım derlemek gerekir? Bunun iki nedeni vardır:

1. **Kullanılabilirlik**. Dağıtım depolarında önceden derlenmiş programların sayısına rağmen, bazı dağıtımlar istenen tüm uygulamaları içermeyebilir. Bu durumda, istenen programı elde etmenin tek yolu, onu kaynak koddan derlemektir.
2. **Zamanında ulaşılabilirlik**. Bazı dağıtımlar en yeni program sürümlerinde uzmanlaşmış olsa da, çoğu bunu yapmamaktadır. Bu, bir programın en son sürümüne sahip olmak için derleme yapmanın gerekli olduğu anlamına gelir.

Yazılımı kaynak kodundan derlemek oldukça karmaşık ve teknik hale gelebilir ve birçok kullanıcının ulaşabileceği bir şey olmayabilir. Ancak, birçok derleme görevi kolaydır ve sadece birkaç adımdan oluşur. Her şey pakete bağlıdır. Sürecin genel bir özetini sunmak ve daha fazla araştırma yapmak isteyenler için bir başlangıç noktası sağlamak amacıyla basit bir örneği inceleyeceğiz.

Yeni bir komut tanıtacağız:

- `make` – Programları korumak için yardımcı program

Derleme nedir?

Basitçe ifade etmek gerekirse, derleme, *kaynak kodunu* (programcı tarafından yazılan, insan tarafından okunabilir bir program açıklaması) bilgisayarın işlemcisinin ana diline çevirme işlemidir.

Bilgisayarın işlemcisi (veya *CPU*), temel düzeyde çalışır ve programları *makine dili* olarak adlandırılan dilde yürütür. Bu, "bu baytı ekle", "bellekteki bu konuma işaret et" veya "bu baytı kopyala" gibi son derece küçük işlemleri tanımlayan sayısal bir koddur.

byte'ı kopyala" gibi son derece küçük işlemleri tanımlayan sayısal bir koddur. Bu talimatların her biri ikili (birler ve sıfırlar) olarak ifade edilir. İlk bilgisayar programları bu sayısal kod kullanılarak yazılmıştır, bu da bu kodu yazan programcıların çok sigara içtiği, litrelerce kahve içtiği ve kalın gözlükler taktığı söylenmesinin nedenini açıklayabilir.

Bu sorun, sayısal kodları CPY (kopyala) ve MOV (taşı) gibi (biraz) daha kolay kullanılan karakter *mnemonikleriyle* değiştiren *assembler dilinin* ortaya çıkmasıyla aşılmıştır. Assembler dilinde yazılan programlar, *assembler* adı verilen bir program tarafından makine diline dönüştürülür. Assembler dili, *aygıt sürücüleri* ve *gömülü sistemler* gibi belirli özel programlama görevleri için bugün hala kullanılmaktadır.

Sırada, *yüksek seviyeli programlama dilleri* olarak adlandırılan *diller* var. Bu diller, programcının işlemcinin yaptığı işin ayrıntılarıyla daha az ilgilenip, elindeki sorunu çözmeye daha fazla odaklanmasını sağladıkları için bu şekilde adlandırılırlar. İlk geliştirilenler (1950'lerde geliştirilenler) arasında *FORTRAN* (bilimsel ve teknik görevler için tasarlanmış) ve *COBOL* (iş uygulamaları için tasarlanmış) bulunur. Her ikisi de bugün hala sınırlı olarak kullanılmaktadır.

Birçok popüler programlama dili olsa da, iki tanesi öne çıkmaktadır. Modern sistemler için yazılan programların çoğu *C* veya *C++* ile yazılmıştır. Aşağıdaki örneklerde, bir C programını derleyeceğiz.

Yüksek seviyeli programlama dillerinde yazılan programlar, *derleyici* adı verilen başka bir programla işlenerek makine diline dönüştürülür. Bazı derleyiciler, yüksek seviyeli komutları assembler diline çevirir ve ardından bir assembler kullanarak makine diline çevirinin son aşamasını gerçekleştirir.

Derleme ile birlikte sıklıkla kullanılan bir işlem, *bağlama olarak* adlandırılır. Programlar tarafından gerçekleştirilen birçok ortak görev vardır. Örneğin, bir dosyayı açmak. Birçok program bu görevi yerine getirir, ancak her programın dosya açmak için kendi rutinini uygulaması israf olur. Dosyaları nasıl açacağını bilen tek bir programlama parçası olması ve buna ihtiyaç duyan tüm programların bunu paylaşmasına izin verilmesi daha mantıklıdır. Yaygın görevler için destek sağlamak, *kütüphaneler* olarak adlandırılan şeylerle gerçekleştirilir. Kütüphaneler, birden fazla programın paylaşabileceği bazı yaygın görevleri yerine getiren birden fazla *rutin* içerir. `/lib` ve /usr/lib dizinlerine bakarsak, bunların çoğunun nerede bulunduğunu görebiliriz. Derleyici çıktısı ile derlenen programın ihtiyaç duyduğu kütüphaneler arasındaki bağlantıları oluşturmak için *linker* adlı bir program kullanılır. Bu işlemin nihai sonucu, kullanıma hazır olan *yürütülebilir program dosyasıdır*.

Tüm Programlar Derlenir mi?

Hayır. Gördüğümüz gibi, derleme gerektirmeyen kabuk betikleri gibi programlar vardır. Bunlar doğrudan çalıştırılır. Bunlar, *betik* veya *yorumlanan* diller olarak bilinen dillerde yazılır. Bu diller son yıllarda popülerlik kazanmıştır ve *Perl, Python, PHP, Ruby* ve diğerleri gibi birçok dili içerir.

Komut dosyası dilleri, *yorumlayıcı* adı verilen özel bir program tarafından yürütülür. Yorumlayıcı

program dosyasını alır ve içindeki her komutu okur ve yürütür. Genel olarak, yorumlanan programlar derlenen programlardan çok daha yavaş çalışır. Bunun nedeni, yorumlanan bir programdaki her kaynak kodu komutunun her yürütüldüğünde çevrilmesidir. Oysa derlenen bir programda, kaynak kodu komutu yalnızca bir kez çevrilir ve bu çeviri nihai yürütülebilir dosyada kalıcı olarak kaydedilir.

Yorumlanan diller neden bu kadar popülerdir? Birçok programlama görevi için sonuçlar "yeterince hızlıdır", ancak asıl avantaj, yorumlanan programların derlenen programlara göre genellikle daha hızlı ve daha kolay geliştirilebilmesidir. Programlar genellikle kod, derleme, test döngüsünün tekrarlandığı bir süreçte geliştirilir. Programın boyutu büyüdükçe, döngünün derleme aşaması oldukça uzun sürebilir. Yorumlanan diller derleme adımını ortadan kaldırarak program geliştirmeyi hızlandırır.

C Programını Derleme

Bir şey derleyelim. Ancak bunu yapmadan önce, derleyici, bağlayıcı ve `make` gibi bazı araçlara ihtiyacımız olacak. Linux ortamında neredeyse evrensel olarak kullanılan C derleyicisi, Richard Stallman tarafından yazılan `gcc` (GNU C Compiler) olarak adlandırılır. Çoğu dağıtım, `gcc'yi` varsayılan olarak yüklemez. Derleyicinin mevcut olup olmadığını şu şekilde kontrol edebiliriz:

```
[me@linuxbox ~]$ which gcc
/usr/bin/gcc
```

Bu örnekteki sonuçlar, derleyicinin kurulu olduğunu göstermektedir.

İpucu: Dağıtımınızda yazılım geliştirme için bir meta paket (paketler koleksiyonu) bulunabilir. Öyleyse, sisteminizde programları derlemek istiyorsanız bunu yüklemeyi düşünün. Sisteminiz bir meta paket sağlamıyorsa, `gcc` ve `make` paketlerini yüklemeyi deneyin. Birçok dağıtımda, bu aşağıdaki alıştırmayı gerçekleştirmek için yeterlidir.

Kaynak Kodunu Elde Etme

Derleme alıştırmamız için, GNU Projesi'nden `diction` adlı bir programı derleyeceğiz. Bu kullanışlı küçük program, metin dosyalarının yazım kalitesini ve stilini kontrol eder. Programlar arasında oldukça küçük ve derlemesi kolaydır.

Geleneklere uygun olarak, önce kaynak kodumuz için
`src adlı bir dizin oluşturacağız` ve ardından `ftp` kullanarak kaynak kodunu bu dizine indireceğiz.

```
[me@linuxbox ~]$ mkdir src [me@linuxbox ~]$ cd
src [me@linuxbox src]$ ftp ftp.gnu.org
ftp.gnu.org'a bağlandı.
220 GNU FTP sunucusu hazır.
Ad (ftp.gnu.org:me): anonymous
230 Giriş başarılı. Uzak sistem türü
UNIX.
Dosyaları aktarmak için ikili mod kullanılıyor. ftp>
cd gnu/diction
250 Dizin başarıyla değiştirildi. ftp> ls
200 PORT komutu başarılı. PASV kullanmayı düşünün.
150 Dizin listesi geliyor.
-rw-r--r--          1  1003     65534       68940   Ağ   28    1998  diction-0.7.tar.gz
-rw-r--r--          1  1003     65534       90957  Mar   04    2002  diction-1.02.tar.gz
-rw-r--r--          1  1003     65534      141062  Eyl   17    2007  diction-1.11.tar.gz
226 Dizin gönderildi.
ftp> get diction-1.11.tar.gz
yerel: diction-1.11.tar.gz uzak: diction-1.11.tar.gz
200 PORT komutu başarılı. PASV kullanmayı düşünün.
150 diction-1.11.tar.gz (141062 bayt) için BINARY modu veri bağlantısı açılıyor.
226 Dosya gönderimi tamamlandı.
0,16 saniyede 141062 bayt alındı (847,4 kB/s) ftp> hoşça kal
221 Hoşça kalın. [me@linuxbox
src]$ ls diction-1.11.tar.gz
```

Önceki örnekte geleneksel olan `ftp` kullandık, ancak kaynak kodunu indirmek için başka yollar da vardır. Örneğin, GNU Projesi HTTPS kullanarak indirmeyi de destekler. `Diction` kaynak kodunu `curl` programını kullanarak indirebiliriz.

```
[me@linuxbox ~]$ curl -O https://ftp.gnu.org/gnu/diction/diction-1.11.tar.gz
  % Toplam    % Alınan % Aktarılan Ortalama Hız Mevcut        Zaman     Zaman      Zaman
                                   İndirme Yükleme
                                                             Toplam    Harcan     Kalan Hız
                                                                       an
100 137k 100 137k             0       0    652k         0 --:--:-- --:--:-- --:--:-- 652k
```

Not: Bu kaynak kodunu derlerken onun "bakımcısı" olduğumuz için, onu `~/src` içinde saklayacağız. Dağıtımınız tarafından yüklenen kaynak kodu `/usr/src` içine yüklenecek, ancak bizim bakımını yaptığımız ve birden fazla kullanıcı tarafından kullanılması amaçlanan kaynak kodu genellikle `/usr/local/src` içine yüklenir.

Gördüğümüz gibi, kaynak kodu genellikle sıkıştırılmış bir tar dosyası biçiminde sağlanır. Bazen *tarball olarak* da adlandırılan bu dosya, kaynak kodunu oluşturan *kaynak ağacını* veya dizin ve dosyaların hiyerarşisini içerir. FTP sitesine ulaştıktan sonra, mevcut tar dosyalarının listesini inceler ve indirmek için en yeni sürümü seçeriz. `FTP` içindeki `get` komutunu kullanarak, dosyayı FTP sunucusundan yerel makineye kopyalarız.

Tar dosyası indirildikten sonra, açılması gerekir. Bu, `tar` programı ile yapılır.

```
[me@linuxbox src]$ tar xzf diction-1.11.tar.gz [me@linuxbox src]$ ls
diction-1.11               diction-1.11.tar.gz
```

İpucu: `diction` programı, tüm GNU Projesi yazılımları gibi, kaynak kodu paketleme konusunda belirli standartlara uyar. Linux ekosisteminde bulunan diğer kaynak kodların çoğu da bu standarda uyar. Standardın bir unsuru, kaynak kodu tar dosyası açıldığında, kaynak ağacını içeren bir dizin oluşturulması ve bu dizinin *project-x.xx* olarak adlandırılması, böylece hem projenin adını hem de sürüm numarasını içermesidir. Bu şema, aynı programın birden fazla sürümünün kolayca kurulmasını sağlar. Ancak, açmadan önce ağacın düzenini incelemek genellikle iyi bir fikirdir. Bazı projeler dizini oluşturmaz, bunun yerine dosyaları doğrudan geçerli dizine teslim eder. Bu, aksi takdirde iyi organize edilmiş `src` dizinimizde karışıklığa neden olur. Bunu önlemek için, tar dosyasının içeriğini incelemek üzere aşağıdaki komutu kullanın:

```
tar tzvf tarfile | head
```

Kaynak Ağacını İnceleme

Tar dosyasının açılması, `diction-1.11` adlı yeni bir dizinin oluşturulmasına neden olur. Bu dizin kaynak ağacını içerir. İçine bakalım.

```
[me@linuxbox src]$ cd diction-1.11
```

```
[me@linuxbox diction-1.11]$ ls
config.guess diction.c config.h.in       getopt.c          nl nl.po
                  diction.pot             getopt.h          README
config.sub        diction.spec            getopt_int.h      cümle.c
configure         diction.spec.in         KURULUM           cümle.h
configure.in diction.texi.in COPYING      install-sh        stil.1.in
                  en                      Makefile.in       stil.c
de                en_GB                   misc.c misc.h     test
de.po             en_GB.po                HABER
diction.1.in getopt1.c
```

Burada bir dizi dosya görüyoruz. GNU Projesi'ne ait programlar ve diğer birçok program, `README`, `INSTALL`, `NEWS` ve `COPYING` adlı dokümantasyon dosyalarını sağlar. Bu dosyalar programın açıklamasını, nasıl derleneceği ve kurulacağına dair bilgileri ve lisans koşullarını içerir. Programı derlemeye başlamadan önce `README` ve `INSTALL` dosyalarını dikkatlice okumak her zaman iyi bir fikirdir.

Bu dizindeki diğer ilginç dosyalar `.c` ve `.h` ile biten dosyalardır.

```
[me@linuxbox diction-1.11]$ ls *.c
diction.c getopt1.c getopt.c misc.c sentence.c style.c [me@linuxbox diction-
1.11]$ ls *.h
getopt.h getopt_int.h misc.h sentence.h
```

`.c` dosyaları, paket tarafından sağlanan iki C programını (`style` ve `dic-tion`) içerir ve modüllere ayrılmıştır. Büyük programların daha küçük, yönetimi daha kolay parçalara bölünmesi yaygın bir uygulamadır. Kaynak kod dosyaları sıradan metin dosyalarıdır ve `less` komutuyla incelenebilir.

```
[me@linuxbox diction-1.11]$ less diction.c
```

`.h` dosyaları *başlık dosyaları* olarak bilinir. Bunlar da sıradan metin dosyalarıdır. Başlık dosyaları, bir kaynak kodu dosyasında veya kütüphanede bulunan rutinlerin açıklamalarını içerir. Derleyicinin modülleri birbirine bağlayabilmesi için, tüm programı tamamlamak için gerekli olan tüm modüllerin açıklamalarını alması gerekir. `diction.c` dosyasının başlangıcına yakın bir yerde şu satırı görürüz:

```
#include "getopt.h"
```

Bu, derleyiciye,

`diction.c` dosyasını okurken `getopt.c` dosyasını da okumasını ve `getopt.c` dosyasında ne olduğunu "bilmesini" söyler. `getopt.c` dosyası, `style` ve `diction` programları tarafından paylaşılan rutinleri sağlar.

`getopt.h` için `include` deyimi öncesinde, aşağıdakiler gibi başka `include` deyimleri görürüz:

```
#include <regex.h> #include
<stdio.h> #include <stdlib.h>
#include <string.h> #include
<unistd.h>
```

Bunlar da başlık dosyalarına atıfta bulunur, ancak mevcut kaynak ağacının dışında bulunan başlık dosyalarına atıfta bulunur. Her programın derlenmesini desteklemek için sistem tarafından sağlanır. `/usr/include` dizinine bakarsak bunları görebiliriz.

```
[me@linuxbox diction-1.11]$ ls /usr/include
```

Bu dizindeki başlık dosyaları, derleyiciyi yüklediğimizde yüklenmiştir.

Programı Oluşturma

Çoğu program, basit, iki komutlu bir dizi ile oluşturulur.

```
./configure make
```

`configure` programı, kaynak ağacı ile birlikte sağlanan bir kabuk betiğidir. Görevi, *derleme ortamını* analiz etmektir. Çoğu kaynak kodu *taşınabilir* olacak şekilde tasarlanmıştır. Yani, birden fazla Unix benzeri sistemde derlenebilecek şekilde tasarlanmıştır. Ancak bunun için, kaynak kodunun derleme sırasında sistemler arasındaki farklılıkları gidermek için küçük ayarlamalar yapılması gerekebilir. `configure` ayrıca gerekli harici araçların ve bileşenlerin kurulu olup olmadığını da kontrol eder. `configure komutunu` çalıştıralım. `configure`, kabuğun normalde programların bulunmasını beklediği yerde bulunmadığından, komutun başına `./` ekleyerek programın geçerli çalışma dizininde bulunduğunu belirtmek suretiyle kabuğa konumunu açıkça belirtmeliyiz.

```
[me@linuxbox diction-1.11]$ ./configure
```

`configure`, derlemeyi test edip yapılandırırken birçok mesaj görüntüler. İşlem bittiğinde, ekran aşağıdaki gibi görünür:

```
libintl.h varlığı kontrol ediliyor... evet
libintl.h kontrol ediliyor... evet
gettext içeren kütüphane kontrol ediliyor... gerekli değil configure:
./config.status oluşturuluyor
config.status: Makefile oluşturuluyor config.status:
diction.1 oluşturuluyor config.status: diction.texi
oluşturuluyor config.status: diction.spec oluşturuluyor
config.status: style.1 oluşturuluyor config.status:
test/rundiction oluşturuluyor config.status: config.h
oluşturuluyor [me@linuxbox diction-1.11]$
```

Burada önemli olan, hata mesajı olmamasıdır. Hata mesajı varsa, yapılandırma başarısız olmuştur ve hatalar düzeltilene kadar program derlenemez.

`Configure'un` kaynak dizinimizde birkaç yeni dosya oluşturduğunu görüyoruz. Bunlardan en önemlisi *makefile* dosyasıdır. Makefile, make programına programı nasıl derleyeceğini tam olarak belirten bir yapılandırma dosyasıdır. Bu dosya olmadan `make` çalışmayı reddeder. Makefile sıradan bir metin dosyasıdır, bu nedenle onu görüntüleyebiliriz:

```
[me@linuxbox diction-1.11]$ less Makefile
```

`make` programı, bitmiş programı oluşturan bileşenler arasındaki ilişkileri ve bağımlılıkları açıklayan bir *makefile* dosyasını (genellikle `Makefile` olarak adlandırılır) girdi olarak alır.

Makefile'ın ilk kısmı, makefile'ın sonraki bölümlerinde ikame edilen değişkenleri tanımlar. Örneğin, aşağıdaki satırı görüyoruz:

```
CC=                 gcc
```

Bu, C derleyicisini `gcc` olarak tanımlar. Makefile'ın ilerleyen kısımlarında, bunun kullanıldığı bir örnek görüyoruz.

```
diction:            diction.o sentence.o misc.o getopt.o getopt1.o
```

```
                    $(CC) -o $@ $(LDFLAGS) diction.o sentence.o misc.o \ getopt.o getopt1.o
                    $(LIBS)
```

Burada bir ikame yapılır ve `$(CC)` değeri çalışma zamanında `gcc` ile değiştirilir.

Makefile'ın çoğu, bir *hedefi* tanımlayan satırlardan oluşur; bu durumda, yürütülebilir dosya `diction` ve onun bağımlı olduğu dosyalar. Kalan satırlar, bileşenlerinden hedefi oluşturmak için gereken komutları açıklar. Bu örnekte, yürütülebilir dosya `diction'ın` (son ürünlerden biri) `diction.o`, `sentence.o`, `misc.o`, `getopt.o` ve `getopt1.o` dosyalarının varlığına bağlı olduğunu görüyoruz. Daha sonra, makefile'da bunların her birinin hedef olarak tanımlarını görüyoruz.

```
diction.o:          diction.c config.h getopt.h misc.h sentence.h getopt.c
getopt.o:           getopt.h getopt_int.h
getopt1.o:          getopt1.c getopt.h getopt_int.h misc.c config.h
misc.o:             misc.h
sentence.o:         sentence.c config.h misc.h sentence.h
style.o:            style.c config.h getopt.h misc.h sentence.h
```

Ancak, bunlar için belirtilmiş herhangi bir komut görmüyoruz. Bu, dosyada daha önce, herhangi bir `.c` dosyasını `.o` dosyasına derlemek için kullanılan komutu tanımlayan genel bir tar-get tarafından ele alınır.

```
.c.o:
                    $(CC) -c $(CPPFLAGS) $(CFLAGS) $<
```

Bu çok karmaşık görünüyor. Neden parçaları derlemek için tüm adımları listeleyip bu işi bitirmiyoruz? Bunun cevabı birazdan netleşecek. Bu arada, `make komutunu` çalıştırıp programlarımızı derleyelim.

```
[me@linuxbox diction-1.11]$ make
```

`make` programı, `Makefile'ın` içeriğini kullanarak eylemlerini yönlendirecek şekilde çalışacaktır. Çok sayıda mesaj üretecektir.

İşlem bittiğinde, tüm hedeflerin artık dizinimizde mevcut olduğunu göreceğiz.

```
[me@linuxbox diction-1.11]$ ls
```

```
config.guess        de.po                en                install-sh        cümle.c
config.h            diction              en_GB             Makefile          cümle.h
config.h.in         diction.1            en_GB.mo          Makefile.in       cümle.o
config.log          diction.1.in         en_GB.po          misc.c            style
config.status       diction.c            getopt1.c         misc.h            stil.1
config.sub          diction.o            getopt1.o         misc.o            stil.1.in
configure           diction.pot          getopt.c          NEWS              style.c
configure.in        diction.spec         getopt.h          nl                style.o
COPYING             diction.spec.in      getopt_int.h      nl.mo             test
de                  diction.texi         getopt.o          nl.po
de.mo               diction.texi.in      YÜKLE             README
```

Dosyalar arasında, oluşturmaya çalıştığımız programlar olan `diction` ve `style`'ı görüyoruz. Tebrikler! Kaynak koddan ilk programlarımızı derledik!

Ama meraktan, `make komutunu` tekrar çalıştıralım.

```
[me@linuxbox diction-1.11]$ make make: `all'
için yapılacak bir şey yok.
```

Sadece bu garip mesajı veriyor. Neler oluyor? Neden programı yeniden derlemedi? Ah, bu `make'in` sihri. `Make`, her şeyi yeniden derlemek yerine, sadece derlenmesi gerekenleri derler. Tüm hedefler mevcut olduğundan, `make` yapılacak bir şey olmadığına karar verdi. Bunu, hedeflerden birini silip make'i yeniden çalıştırarak ne yaptığını görerek gösterebiliriz. Ara hedeflerden birini kaldıralım.

```
[me@linuxbox diction-1.11]$ rm getopt.o [me@linuxbox diction-
1.11]$ make
```

`Make`'in, eksik modüle bağlı oldukları için `sözlük` ve `stil` programlarını yeniden derlediğini ve yeniden bağladığını görüyoruz. Bu davranış, `make'in` bir başka önemli özelliğini de ortaya koymaktadır: hedefleri güncel tutar. `Make`, hedeflerin bağımlılıklarından daha yeni olmasını şart koşar. Bu çok mantıklıdır, çünkü bir programcı genellikle kaynak kodun bir kısmını günceller ve ardından `make'i` kullanarak bitmiş ürünün yeni bir sürümünü oluşturur. `make`, güncellenen koda dayalı olarak oluşturulması gereken her şeyin oluşturulmasını sağlar. `touch` programını kullanarak kaynak kod dosyalarından birini "güncellemek" istersek, şunu görebiliriz:

```
[me@linuxbox diction-1.11]$ ls -l diction getopt.c
-rwxr-xr-x 1 me             me         37164 2025-03-05 06:14 diction
```

```
-rw-r--r-- 1 ben            me         33125 2007-03-30 17:45 getopt.c
[me@linuxbox diction-1.11]$ touch getopt.c [me@linuxbox diction-1.11]$
ls -l diction getopt.c
-rwxr-xr-x 1 me             me         37164 2025-03-05 06:14 diction
-rw-r--r-- 1 me             me         33125 2025-03-05 06:23 getopt.c
[me@linuxbox diction-1.11]$ make
```

`make` çalıştırıldıktan sonra, hedefin bağımlılıktan daha yeni bir hale getirildiğini görüyoruz:

```
[me@linuxbox diction-1.11]$ ls -l diction getopt.c
-rwxr-xr-x 1 ben            ben        37164 2025-03-05 06:24 diction
-rw-r--r-- 1 ben            ben        33125 2025-03-05 06:23 getopt.c
```

`Make'in` sadece yapılması gerekenleri akıllıca derleme yeteneği, programcılar için büyük bir avantajdır. Küçük projelerimizde zaman tasarrufu çok belirgin olmayabilir, ancak daha büyük projelerde çok önemlidir. Unutmayın, Linux çekirdeği (sürekli değiştirilip geliştirilen bir program) birkaç *milyon* satırlık kod içerir.

Programı Yükleme

İyi paketlenmiş kaynak kodu genellikle `install` adlı özel bir `make` hedefi içerir. Bu hedef, nihai ürünü kullanım için bir sistem dizinine yükler. Genellikle bu dizin, yerel olarak oluşturulan yazılımlar için geleneksel konum olan `/usr/local/bin'dir`. Ancak, bu dizin normalde sıradan kullanıcılar tarafından yazılabilir değildir, bu nedenle yüklemeyi gerçekleştirmek için süper kullanıcı olmalıyız.

```
[me@linuxbox diction-1.11]$ sudo make install
```

Yüklemeyi gerçekleştirdikten sonra, programın çalışmaya hazır olduğunu kontrol edebiliriz.

```
[me@linuxbox diction-1.11]$ which diction
/usr/local/bin/diction
[me@linuxbox diction-1.11]$ man diction
```

İşte bu kadar!

Özet

Bu bölümde, üç basit komutun nasıl çalıştığını gördük:

```
./configure make
make install
```

birçok kaynak kodu paketini oluşturmak için nasıl kullanılabileceğini gördük. Ayrıca, `make komutunun` programların bakımında oynadığı önemli rolü de gördük. `Make` programı, sadece kaynak kodunu derlemek için değil, hedef/bağımlılık ilişkisini sürdürmek gereken her türlü görev için kullanılabilir.

Daha Fazla Okuma

- Wikipedia'da derleyiciler ve `make` programı hakkında iyi makaleler bulunmaktadır: https://en.wikipedia.org/wiki/Compiler https://en.wikipedia.org/wiki/Make_(software)
- *GNU Make Kılavuzu*: https://www.gnu.org/software/make/manual/html_node/index.html

Bölüm 4 – Kabuk Betikleri Yazma

24 – İlk Komut Dosyanızı Yazmak

Önceki bölümlerde, bir dizi komut satırı aracı bir araya getirdik. Bu araçlar birçok türde bilgisayar sorununu çözebilse de, bunları komut satırında tek tek manuel olarak kullanmakla sınırlıyız. Kabuğun daha fazla işi yapmasını sağlayabilsek ne kadar harika olurdu, değil mi? Bunu yapabiliriz. Araçlarımızı kendi tasarladığımız programlarda bir araya getirerek, kabuk karmaşık görev dizilerini tek başına gerçekleştirebilir. Bunu, *kabuk komut dosyaları* yazarak sağlayabiliriz.

Kabuk Betikleri Nedir?

En basit ifadeyle, kabuk betiği bir dizi komut içeren bir dosyadır. Kabuk bu dosyayı okur ve komutları sanki komut satırına doğrudan girilmiş gibi yerine getirir.

Kabuk, hem sisteme yönelik güçlü bir komut satırı arayüzü hem de bir komut dosyası dili yorumlayıcısı olması bakımından bir bakıma benzersizdir. Göreceğimiz gibi, komut satırında yapılabileceklerin çoğu komut dosyalarında da yapılabilir ve komut dosyalarında yapılabileceklerin çoğu komut satırında da yapılabilir.

Kabuğun birçok özelliğini ele aldık, ancak komut satırında en sık kullanılan özelliklere odaklandık. Kabuk ayrıca, program yazarken genellikle (ama her zaman değil) kullanılan bir dizi özellik de sağlar.

Kabuk Komut Dosyası Nasıl Yazılır

Bir kabuk betiğini başarıyla oluşturmak ve çalıştırmak için üç şey yapmamız gerekir.

1. **Bir komut** dosyası **yazın.** Kabuk komut dosyaları sıradan metin dosyalarıdır. Bu nedenle, bunları yazmak için bir metin düzenleyicisine ihtiyacımız var. En iyi metin düzenleyicileri, komut dosyasının öğelerini renk kodlu bir görünümde görmemizi sağlayan *sözdizimi vurgulaması özelliğini* sunar. Sözdizimi vurgulaması, belirli türdeki yaygın hataları tespit etmemize yardımcı olur. `vim`, `gedit`, `kate` ve diğer birçok düzenleyici, komut dosyası yazmak için iyi seçeneklerdir.
2. **Komut dosyasını çalıştırılabilir hale getirin.** Sistem, eski metin dosyalarının program olarak değerlendirilmesine izin vermeme konusunda oldukça titizdir ve bunun iyi bir nedeni vardır! Komut dosyasının çalıştırılmasına izin vermek için izinlerini ayarlamamız gerekir.

3. **Komut dosyasını kabuğun bulabileceği bir yere koyun.** Kabuk, açık bir yol adı belirtilmediğinde belirli dizinlerde yürütülebilir dosyaları otomatik olarak arar. Maksimum kolaylık için komut dosyalarımızı bu dizinlere yerleştireceğiz.

Komut Dosyası Biçimi

Programlama geleneğine uygun olarak, son derece basit bir komut dosyasını göstermek için bir "Hello World" programı oluşturacağız. Metin editörlerimizi açalım ve aşağıdaki komut dosyasını girelim:

```
#!/bin/bash

# Bu bizim ilk komut

dosyamız. echo 'Hello World!'
```

Komut dosyasının son satırı oldukça tanıdık; sadece bir string argümanı olan bir `echo` komutu. İkinci satır da tanıdık. İncelediğimiz ve düzenlediğimiz birçok yapılandırma dosyasında gördüğümüz bir yorum gibi görünüyor. Kabuk komut dosyalarındaki yorumlarla ilgili bir şey, en az bir boşluk karakteri ile başlamak kaydıyla satırların sonunda da görünebilirler, şöyle:

```
echo 'Hello World!' # Bu da bir yorumdur
```

Satırda # sembolünden sonraki her şey yok sayılır. Diğer birçok

şey gibi, bu da komut satırında çalışır:

```
[me@linuxbox ~]$ echo 'Hello World!' # Bu da bir yorumdur Hello
World!
```

Yorumlar komut satırında pek kullanışlı olmasa da, çalışırlar.

Komut dosyasının ilk satırı biraz gizemlidir. # ile başladığı için yorum gibi görünse de, sadece yorum olmak için fazla anlamlıdır. #! karakter dizisi, aslında *shebang* adı verilen özel bir yapıdır. Shebang, çekirdeğe, takip eden komut dosyasını çalıştırmak için kullanılması gereken yorumlayıcının adını bildirmek için kullanılır. Her kabuk komut dosyası, ilk satırında bunu içermelidir.

Komut dosyasını `hello_world` olarak kaydedelim.

Yürütülebilir İzinler

Şimdi yapmamız gereken şey, komut dosyasını çalıştırılabilir hale getirmek. Bu, `chmod komutunu kullanarak kolayca yapılabilir`.

```
[me@linuxbox ~]$ ls -l hello_world
-rw-r--r-- 1 me       me       63 2025-03-07 10:10 hello_world
[me@linuxbox ~]$ chmod 755 hello_world
[me@linuxbox ~]$ ls -l hello_world
-rwxr-xr-x 1 ben      ben      63 2025-03-07 10:10 hello_world
```

Komut dosyaları için iki yaygın izin ayarı vardır: herkesin çalıştırabileceği komut dosyaları için 755 ve yalnızca sahibinin çalıştırabileceği komut dosyaları için 700. Komut dosyalarının çalıştırılabilmesi için okunabilir olması gerektiğini unutmayın.

Komut Dosyası Konumu

İzinler ayarlandıktan sonra, artık komut dosyasını çalıştırabiliriz.

```
[me@linuxbox ~]$ ./hello_world
Merhaba Dünya!
```

Komut dosyasının çalışması için, komut dosyasının adının önüne açık bir yol eklemeliyiz. Bunu yapmazsak, şu mesajı alırız:

```
[me@linuxbox ~]$ hello_world
bash: hello_world: komut bulunamadı
```

Bunun nedeni nedir? Komut dosyamızı diğer programlardan ayıran nedir? Aslında hiçbir şey. Komut dosyamızda bir sorun yok. Sorun, komut dosyasının konumunda. 11. bölümde, `PATH` ortam değişkenini ve bunun sistemin yürütülebilir programları arama şekline etkisini ele aldık. Özetlemek gerekirse, sistem, açık bir yol belirtilmediğinde, yürütülebilir bir program bulması gerektiğinde her seferinde bir dizin listesini arar. Komut satırına `ls` yazdığımızda sistemin `/bin/ls komutunu` çalıştırmasını sağlayan şey budur. `/bin` dizini, sistemin otomatik olarak aradığı dizinlerden biridir. Dizinlerin listesi, `PATH` adlı bir ortam değişkeninde tutulur. `PATH` değişkeni, aranacak dizinlerin iki nokta üst üste ile ayrılmış bir listesini içerir. `PATH'ın` içeriğini görüntüleyebiliriz.

```
[me@linuxbox ~]$ echo $PATH
/home/me/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:
/bin:/usr/games
```

Burada dizinlerimizin listesini görüyoruz. Eğer komut dosyamız listedeki dizinlerden herhangi birinde bulunuyorsa, sorunumuz çözülmüş olur. Listedeki ilk dizin olan `/home/me/bin'e` dikkat edin. Çoğu Linux dağıtımı, kullanıcıların kendi programlarını çalıştırabilmeleri için `PATH` değişkenini kullanıcının ana dizininde bir `bin` dizini içerecek şekilde yapılandırır. Dolayısıyla, `bin` dizinini oluşturup komut dosyasını bu dizine yerleştirirsek, diğer programlar gibi çalışmaya başlaması gerekir.

```
[me@linuxbox ~]$ mkdir bin
[me@linuxbox ~]$ mv hello_world bin
[me@linuxbox ~]$ hello_world
Merhaba Dünya!
```

Ve çalışıyor.

`PATH` değişkeni dizini içermiyorsa, `.bashrc` dosyamıza şu satırı ekleyerek kolayca ekleyebiliriz:

```
export PATH=~/bin:"$PATH"
```

Bu değişiklik yapıldıktan sonra, her yeni terminal oturumunda geçerli olacaktır. Değişikliği mevcut terminal oturumuna uygulamak için, kabuğun `.bashrc` dosyasını yeniden okumasını sağlamalıyız. Bu, dosyayı "kaynaklayarak" yapılabilir.

```
[me@linuxbox ~]$ . .bashrc
```

Nokta (`.`) komutu, `kaynak` komutunun eşanlamlısıdır. Kaynak komutu, belirtilen bir kabuk komut dosyasını okuyan ve onu klavyeden gelen girdi gibi işleyen bir kabuk yerleşik komutudur.

Not**:** Ubuntu (ve diğer Debian tabanlı dağıtımların çoğu), kullanıcının `.bashrc` dosyası çalıştırıldığında `~/bin` dizini mevcutsa, `PATH` değişkenine otomatik olarak `~/bin` dizinini ekler. Dolayısıyla, Ubuntu sistemlerinde `~/bin` dizinini oluşturup oturumu kapatıp tekrar oturum açarsak, her şey çalışır.

Komut Dosyaları için Uygun Konumlar

`~/bin` dizini, kişisel kullanım amaçlı komut dosyalarını yerleştirmek için uygun bir yerdir. Sistemdeki herkesin kullanmasına izin verilen bir komut dosyası yazarsak, geleneksel konum `/usr/local/bin'dir`. Sistem yöneticisi tarafından kullanılmak üzere tasarlanan komut dosyaları genellikle `/usr/local/sbin` dizininde bulunur. Çoğu durumda, yerel olarak sağlanan yazılımlar, ister komut dosyaları ister derlenmiş programlar olsun, `/bin` veya `/usr/bin` dizinlerine değil, `/usr/local` hiyerarşisine yerleştirilmelidir. Bu dizinler, Linux Dosya Sistemi Hiyerarşi Standardı tarafından, yalnızca Linux dağıtıcısı tarafından sağlanan ve bakımı yapılan dosyaları içerecek şekilde tanımlanmıştır.

Daha Fazla Biçimlendirme Püf Noktası

Ciddi komut dosyası yazımının temel amaçlarından biri, *bakım* kolaylığı, yani komut dosyasının yazarı veya başkaları tarafından değişen ihtiyaçlara uyum sağlamak için kolayca değiştirilebilmesidir. Komut dosyasını okunması ve anlaşılması kolay hale getirmek, bakımı kolaylaştırmanın bir yoludur.

Uzun Seçenek Adları

Çalıştığımız komutların çoğu hem kısa hem de uzun seçenek adlarına sahiptir. Örneğin, `ls` komutu kısa veya uzun biçimde ifade edilebilen birçok seçeneğe sahiptir. Örneğin, aşağıdaki gibi:

```
[me@linuxbox ~]$ ls -ad
```

şuna eşdeğerdir:

```
[me@linuxbox ~]$ ls --all --directory
```

Yazma işini azaltmak için, komut satırına seçenekler girerken kısa seçenekler tercih edilir, ancak komut dosyaları yazarken uzun seçenekler okunabilirliği artırabilir.

Girinti ve Satır Devamı

Uzun komutlar kullanıldığında, komutu birkaç satıra yayarak okunabilirliği artırabilirsiniz. 17. bölümde, find komutunun özellikle uzun bir örneğini incelemiştik.

```
[me@linuxbox ~]$ find playground \( -type f -not -perm 0600 -exec
```

```
chmod 0600 '{}' ';' \) -or \( -type d -not -perm 0700 -exec chmod 0700
'{}' ';' \)
```

Açıkçası, bu komut ilk bakışta biraz anlaşılması zor. Bir komut dosyasında, bu komut şu şekilde yazılırsa daha kolay anlaşılabilir:

```
find playground \
    \( \
         -type f \
         -not -perm 0600 \
         -exec chmod 0600 '{}' ';' \
    \) \
    -veya \
    \( \
         -type d \
         -not -perm 0700 \
         -exec chmod 0700 '{}' ';' \
    \)
```

Satır devamı (ters eğik çizgi-satır besleme dizileri) ve girinti kullanarak, bu karmaşık komutun mantığı okuyucuya daha açık bir şekilde anlatılır. Bu teknik komut satırında da işe yarar, ancak yazması ve düzenlemesi zor olduğu için nadiren kullanılır. Komut dosyası ile komut satırı arasındaki bir fark, komut dosyasının girinti yapmak için sekme karakterleri kullanabilmesidir, oysa komut satırı bunu yapamaz çünkü sekmeler tamamlama işlemini etkinleştirmek için kullanılır.

`Vim'i` Komut Dosyası Yazmak İçin Yapılandırma

`Vim` metin düzenleyicisinin çok sayıda yapılandırma ayarı vardır. Birkaç yaygın seçenek, komut dosyası yazımını kolaylaştırabilir.

Aşağıdaki komut, sözdizimi vurgulamayı etkinleştirir:

`:syntax on`

Bu ayar ile, bir komut dosyasını görüntülerken kabuk sözdiziminin farklı öğeleri farklı renklerde görüntülenecektir. Bu, belirli türdeki programlama hatalarını tanımlamak için yararlıdır. Ayrıca görsel olarak da hoş görünür. Bu özelliğin çalışması için, tam sürüm `vim'in` yüklü olması ve düzenlediğiniz dosyanın

shebang içermesi gerekir. Önceki komutta sorun yaşıyorsanız, bunun yerine **`:set syntax=sh`** komutunu deneyin.

Aşağıdaki komut, arama sonuçlarını vurgulamak için seçeneği etkinleştirir.

```
:set hlsearch
```

`echo` kelimesini aradığımızı varsayalım. Bu seçenek etkinleştirildiğinde, kelimenin her bir örneği vurgulanacaktır.

Aşağıdaki komut, sekme karakterinin kapladığı sütun sayısını ayarlar:

```
:set tabstop=4
```

Varsayılan değer sekiz sütundur. Değeri 4 olarak ayarlamak (yaygın bir uygulamadır), uzun satırların ekrana daha kolay sığmasını sağlar.

Aşağıdaki komut "otomatik girinti" özelliğini etkinleştirir:

```
:set autoindent
```

Bu, `vim'in` yeni satırı, yazılan satırla aynı miktarda girintilemesini sağlar. Bu, birçok programlama yapısında yazma hızını artırır. Girintilemeyi durdurmak için `Ctrl-d tuşlarına` basın.

Bu değişiklikleri, `~/.vimrc` dosyanıza bu komutları (başındaki iki nokta üst üste karakterleri olmadan) ekleyerek kalıcı hale getirebilirsiniz.

Özet

Senaryo yazımının bu ilk bölümünde, senaryoların nasıl yazıldığını ve sistemimizde kolayca çalıştırılabilmesini sağladığımızı gördük. Ayrıca, senaryolarımızın okunabilirliğini (ve dolayısıyla bakım kolaylığını) artırmak için çeşitli biçimlendirme tekniklerini nasıl kullanabileceğimizi de gördük. İlerleyen bölümlerde, bakım kolaylığı, iyi senaryo yazımının temel ilkelerinden biri olarak tekrar tekrar gündeme gelecektir.

Daha Fazla Okuma

- Çeşitli programlama dillerinde "Hello World" programları ve örnekleri için bkz. https://en.wikipedia.org/wiki/Hello_world
- Bu Wikipedia makalesi, shebang mekanizması hakkında daha fazla bilgi vermektedir: https://en.wikipedia.org/wiki/Shebang_(Unix)

25 – Bir Projeye Başlamak

Bu bölümden itibaren bir program oluşturmaya başlayacağız. Bu projenin amacı, çeşitli kabuk özelliklerinin programları oluşturmak ve daha da önemlisi, *iyi* programlar oluşturmak için nasıl kullanıldığını görmektir.

Yazacağımız program bir *rapor oluşturucu*. Sistemimiz ve durumu hakkında çeşitli istatistikler sunacak ve bu raporu HTML formatında oluşturacak, böylece Firefox veya Chrome gibi bir web tarayıcısıyla görüntüleyebileceğiz.

Programlar genellikle bir dizi aşamada oluşturulur ve her aşamada yeni özellikler ve yetenekler eklenir. Programımızın ilk aşaması, sistem bilgisi içermeyen minimal bir HTML belgesi üretecektir. Sistem bilgileri daha sonra eklenecektir.

İlk Aşama: Minimal Belge

İlk olarak, iyi biçimlendirilmiş bir HTML belgesinin formatını bilmemiz gerekir. Bu format şu şekildedir:

```
<html>
    <head>
         <title>Sayfa Başlığı</title>
    </head>
    <body>
         Sayfa gövdesi.
    </body>
</html>
```

Bunu metin düzenleyicimize girip dosyayı `foo.html` olarak kaydettikten sonra, dosyayı görüntülemek için Firefox'ta aşağıdaki URL'yi kullanabiliriz:

`file:///home/username/foo.html`

Programımızın ilk aşaması, bu HTML dosyasını standart çıktıya aktarabilecektir. Bunu yapmak için oldukça kolay bir program yazabiliriz. Metin düzenleyicimizi başlatıp `~/bin/sys_info_page` adlı yeni bir dosya oluşturalım.

```
[me@linuxbox ~]$ vim ~/bin/sys_info_page
```

Aşağıdaki programı girin:

```
#!/bin/bash

# Sistem bilgisi sayfasını görüntüleyen program

echo "<html>"
echo "   <head>"
echo "        <title>Sayfa
Başlığı</title>" echo " </head>"
echo "   <body>"
echo "        Sayfa
gövdesi." echo " </body>"
echo "</html>"
```

Bu soruna ilk denememizde bir shebang, bir yorum (her zaman iyi bir fikirdir) ve her çıktı satırı için bir `echo` komutu dizisi bulunmaktadır. Dosyayı kaydettikten sonra, onu çalıştırılabilir hale getireceğiz ve çalıştırmayı deneyeceğiz.

```
[me@linuxbox ~]$ chmod 755 ~/bin/sys_info_page
[me@linuxbox ~]$ sys_info_page
```

Program çalıştırıldığında, komut dosyasında bulunan `echo` komutları çıktısını standart çıktıya gönderdiğinden, HTML belgesinin metni ekranda görüntülenmelidir. Programı tekrar çalıştırıp, programın çıktısını `sys_info_page.html` dosyasına yönlendireceğiz, böylece sonucu bir web tarayıcısıyla görüntüleyebileceğiz.

```
[me@linuxbox ~]$ sys_info_page > sys_info_page.html
[me@linuxbox ~]$ firefox sys_info_page.html
```

Şimdiye kadar her şey yolunda.

Program yazarken, sadelik ve netlik için çaba sarf etmek her zaman iyi bir fikirdir. Program okunması ve anlaşılması kolay olduğunda bakımı daha kolaydır, ayrıca yazma miktarını azaltarak programı yazmayı da kolaylaştırır. Programın mevcut sürümü iyi çalışıyor, ancak daha basit olabilir. Aslında tüm echo komutlarını tek bir komutta birleştirebiliriz, bu da programa daha fazla satır eklemeyi kesinlikle kolaylaştıracaktır.

gram'ın çıktısı. Öyleyse, programımızı şu şekilde değiştirelim:

```
#!/bin/bash

# Sistem bilgileri sayfasını çıktılamak için program

echo "<html>
    <head>
        <title>Sayfa Başlığı</title>
    </head>
    <body>
        Sayfa gövdesi.
    </body>
</html>"
```

Tırnak içine alınan bir dize satır sonları içerebilir ve bu nedenle birden fazla satır metin içerebilir. Kabuk, kapanış tırnak işaretine rastlayana kadar metni okumaya devam eder. Bu, komut satırında da aynı şekilde çalışır:

```
[me@linuxbox ~]$ echo "<html>
>         <head>
>                 <title>Sayfa Başlığı</title>
>         </head>
>         <body>
>                 Sayfa
>         </body> gövdesi.
> </html>"
```

Baştaki “>” karakteri, `PS2` kabuk değişkeninde bulunan kabuk istemidir. Kabuğa çok satırlı bir ifade yazdığımızda her zaman görünür. Bu özellik şu anda biraz belirsizdir, ancak daha sonra çok satırlı programlama ifadeleri konusunu ele aldığımızda oldukça kullanışlı olduğu ortaya çıkacaktır.

İkinci Aşama: Biraz Veri Ekleme

Artık programımız minimal bir belge oluşturabiliyor, raporumuza biraz veri ekleyelim. Bunu yapmak için aşağıdaki değişiklikleri yapacağız:

```
#!/bin/bash
```

```
# Sistem bilgisi sayfasını çıktılamak için program

echo "<html>
        <head>
                <title>Sistem Bilgisi Raporu</title>
        </head>
        <body>
                <h1>Sistem Bilgisi Raporu</h1>
        </body>
</html>"
```

Raporun gövdesine bir sayfa başlığı ve bir başlık ekledik.

Değişkenler ve Sabitler

Ancak, komut dosyamızda bir sorun var. "Sistem Bilgisi Raporu" dizesinin nasıl tekrarlandığını fark ettiniz mi? Küçük komut dosyamızda bu bir sorun değil, ancak komut dosyamızın çok uzun olduğunu ve bu dizenin birden fazla örneğinin olduğunu düşünelim. Başlığı başka bir şeyle değiştirmek istersek, bunu birden fazla yerde değiştirmemiz gerekir ve bu çok fazla iş gerektirebilir. Dizgiyi birden fazla kez değil, yalnızca bir kez görünecek şekilde komut dosyasını düzenleyebilsek ne olurdu? Bu, komut dosyasının gelecekteki bakımını çok daha kolay hale getirirdi. Bunu şu şekilde yapabiliriz:

```
#!/bin/bash

# Sistem bilgisi sayfasını görüntülemek için program

title="Sistem Bilgisi Raporu"

echo "<html>
        <head>
                <title>$title</title>
        </head>
        <body>
                <h1>$title</h1>
        </body>
</html>"
```

`title` adlı bir *değişken* oluşturup ona `System Information Report` değerini atayarak, parametre genişletme özelliğinden yararlanabilir ve bu dizeyi birden fazla yere yerleştirebiliriz.

Peki, değişkeni nasıl oluştururuz? Çok basit, sadece kullanırız. Kabuk bir değişkenle karşılaştığında, onu otomatik olarak oluşturur. Bu, değişkenlerin kullanılmadan önce açıkça *bildirilmesi* veya tanımlanması gereken birçok programlama dilinden farklıdır. Kabuk bu konuda çok esnektir, bu da bazı sorunlara yol açabilir. Örneğin, komut satırında gerçekleşen şu senaryoyu ele alalım:

```
[me@linuxbox  ~]$  foo="yes"
[me@linuxbox  ~]$  echo  $foo
yes
[me@linuxbox ~]$ echo $fool

[me@linuxbox ~]$
```

Önce `foo` değişkenine `yes` değerini atıyoruz, ardından `echo` ile değerini görüntülüyoruz. Sonra, yanlış yazılmış `fool` değişkeninin değerini görüntülüyoruz ve boş bir sonuç alıyoruz. Bunun nedeni, kabuk, `fool` değişkeniyle karşılaştığında onu memnuniyetle oluşturmuş ve ona varsayılan olarak hiçbir şey veya boş değer vermiş olmasıdır. Bundan, yazımımıza çok dikkat etmemiz gerektiğini öğreniyoruz! Bu örnekte gerçekte ne olduğunu anlamak da önemlidir. Kabuğun genişletmeleri nasıl gerçekleştirdiğini önceki bölümde incelediğimizden, aşağıdaki komutun

```
[me@linuxbox ~]$ echo $foo
```

parametre genişletmesinden geçer ve şu sonucu verir:

```
[me@linuxbox ~]$ echo yes
```

Buna karşılık, aşağıdaki komut:

```
[me@linuxbox ~]$ echo $fool
```

şöyle genişler:

```
[me@linuxbox ~]$ echo
```

Boş değişken hiçbir şeye genişlemez! Bu, argüman gerektiren komutlarda büyük sorunlara yol açabilir. İşte bir örnek:

```
[me@linuxbox ~]$ foo=foo.txt
[me@linuxbox ~]$ foo1=foo1.txt
[me@linuxbox ~]$ cp $foo $foo1
cp: `foo.txt'den sonra hedef dosya işleneni eksik Daha
fazla bilgi için `cp --help' komutunu deneyin.
```

İki değişkene, `foo` ve `foo1`, değerler atıyoruz. Ardından `cp` komutunu çalıştırıyoruz, ancak ikinci argümanın adını yanlış yazıyoruz (rakam 1 yerine küçük harf L kullanıyoruz). Genişletmeden sonra, `cp` komutuna iki argüman gerekmesine rağmen yalnızca bir argüman gönderiliyor.

Değişken adları ile ilgili bazı kurallar vardır:

1. Değişken adları alfasayısal karakterlerden (harfler ve sayılar) ve alt çizgi karakterlerinden oluşabilir.
2. Değişken adının ilk karakteri bir harf veya alt çizgi olmalıdır.
3. Boşluklar ve noktalama işaretleri kullanılamaz.

"Değişken" kelimesi, değişen bir değeri ifade eder ve birçok uygulamada değişkenler bu şekilde kullanılır. Ancak, uygulamamızdaki değişken olan `başlık`, *sabit* olarak kullanılır. Sabit, bir adı olması ve bir değer içermesi bakımından değişkenlere benzer. Aradaki fark, sabitin değerinin değişmemesidir. Geometrik hesaplamalar yapan bir uygulamada, `PI'yı` bir sabit olarak tanımlayabilir ve ona `3,1415` değerini atayabiliriz, bunun yerine programımız boyunca sayıyı olduğu gibi kullanmak yerine. Kabuk, değişkenler ve sabitler arasında hiçbir ayrım yapmaz; bunlar çoğunlukla programcının rahatlığı içindir. Yaygın bir kural, sabitleri belirtmek için büyük harfleri, gerçek değişkenleri belirtmek için ise küçük harfleri kullanmaktır. Bu kurala uymak için betiğimizi değiştireceğiz:

```
#!/bin/bash

# Sistem bilgisi sayfasını görüntülemek için program

TITLE="$HOSTNAME için Sistem Bilgisi Raporu" echo

"<html>
        <head>
                <title>$TITLE</title>
        </head>
        <body>
                <h1>$TITLE</h1>
        </body>
</html>"
```

Ayrıca, kabuk değişkeni
`HOSTNAME değişkeninin değerini` ekleyerek başlığımızı renklendirdik. Bu, makinenin ağ adıdır.

Not: Kabuk, `-r` (salt okunur) seçeneği ile `declare` yerleşik komutunu kullanarak sabitlerin değişmezliğini zorlamak için bir yol sağlar. `TITLE'ı` şu şekilde atamış olsaydık:

```
declare -r TITLE="Sayfa Başlığı"
```

kabuk, `TITLE`'a sonraki herhangi bir atamayı engeller. Bu özellik nadiren kullanılır, ancak çok resmi komut dosyaları için mevcuttur.

Değişkenlere ve Sabitlere Değer Atama

Burada, genişletme konusundaki bilgilerimiz gerçekten işe yaramaya başlıyor. Gördüğümüz gibi, değişkenlere bu şekilde değerler atanır; burada *değişken*, değişkenin adı ve *değer* ise bir dizedir:

değişken=değer

Diğer bazı programlama dillerinden farklı olarak, kabuk bir değişkene atanan verinin türünü önemsemez; hepsini dize olarak ele alır. `-i` seçeneği ile `declare` komutunu kullanarak kabuğu atamayı tamsayılara sınırlamaya zorlayabilirsiniz, ancak değişkenleri salt okunur olarak ayarlamak gibi, bu da nadiren yapılır.

Atamada, değişken adı, eşittir işareti ve değer arasında boşluk olmaması gerektiğini unutmayın. Peki değer neyden oluşabilir? Dizeye genişletebileceğimiz her şey olabilir.

```
a=z                     # "z" dizesini a değişkenine atayın. b="a
string"                 # Yerleşik boşluklar tırnak işaretleri
içinde olmalıdır.
c="bir dize ve $b"      # Değişkenler gibi diğer genişletmeler olabilir.
                        # ödevde genişletildi. d="$(ls -l
foo.txt)"               # Komutun sonuçları.
e=$((5 * 7))            # Aritmetik genişletme.
f="\t\ta string\n"      # Sekme ve satır sonu gibi kaçış dizileri.
```

Tek bir satırda birden fazla değişken ataması yapılabilir.

```
a=5 b="bir dize"
```

Genişletme sırasında, değişken adları isteğe bağlı olarak küme parantezleri (köşeli parantezler) ile çevrelenebilir
`{}`. Bu, değişken adının çevresindeki bağlam nedeniyle belirsiz hale geldiği durumlarda yararlıdır. Burada, bir değişken kullanarak bir dosyanın adını `myfile'dan myfile1'e` değiştirmeye çalışıyoruz:

```
[me@linuxbox ~]$ filename="myfile" [me@linuxbox
~]$ touch "$filename" [me@linuxbox ~]$ mv
"$filename" "$filename1"
mv: `myfile'den sonra hedef dosya işleneni eksik Daha
fazla bilgi için `mv --help' komutunu deneyin.
```

Bu deneme, kabuk mv komutunun ikinci argümanını yeni (ve boş) bir değişken olarak yorumladığı için başarısız olur. Sorun şu şekilde çözülebilir:

```
[me@linuxbox ~]$ mv "$filename" "${filename}1"
```

Çevreleyen parantezleri ekleyerek, kabuk artık sondaki `1'i` değişken adının bir parçası olarak yorumlamaz.

Not: Kabuğun kelime bölme etkisini sınırlamak için değişkenleri ve komut ikamelerini çift tırnak içine almak iyi bir uygulamadır. Bir değişken dosya adı içerebileceğinden, tırnak kullanımı özellikle önemlidir.

Bu fırsatı değerlendirerek raporumuza bazı veriler ekleyeceğiz, yani raporun oluşturulduğu tarih ve saat ile oluşturucunun kullanıcı adı.

```
#!/bin/bash

# Sistem bilgisi sayfasını çıktılamak için program

TITLE="$HOSTNAME için Sistem Bilgisi Raporu"
CURRENT_TIME="$(date +"%x %r %Z")"
TIMESTAMP="$CURRENT_TIME tarihinde $USER tarafından oluşturuldu"

echo "<html>
        <head>
                <title>$TITLE</title>
        </head>
        <body>
```

```
                <h1>$TITLE</h1>
                <p>$TIMESTAMP</p>
        </body>
</html>"
```

Burada Belgeler

Metnimizi çıktılamak için `echo` komutunu kullanan iki farklı yöntem inceledik. Üçüncü bir yöntem ise *burada belge* veya *burada komut dosyası* olarak adlandırılır. Burada belge, metnin bir bölümünü komut dosyasına gömüp komutun standart girişine beslediğimiz ek bir I/O yönlendirme biçimidir. Şu şekilde çalışır:

komut << token

metin

token

Bu örnekte, *command* standart girişi kabul eden komutun adıdır ve *token* gömülü metnin sonunu belirtmek için kullanılan bir dizedir. Burada, burada belgeyi kullanmak için komut dosyasını değiştireceğiz:

```
#!/bin/bash

# Sistem bilgisi sayfasını görüntüleyen program

TITLE="$HOSTNAME için Sistem Bilgisi Raporu" CURRENT_TIME="$(date
+"%x %r %Z")"
TIMESTAMP="$CURRENT_TIME tarihinde $USER tarafından oluşturuldu"

cat << _EOF_
<html>
        <head>
                <title>$TITLE</title>
        </head>
        <body>
                <h1>$TITLE</h1>
                <p>$TIMESTAMP</p>
        </body>
</html>
_EOF_
```

`Echo` kullanmak yerine, betiğimiz artık `cat` ve bir here document kullanıyor. `_EOF_` dizesi

(*dosya sonu anlamına gelir*, yaygın bir kuraldır) belirteç olarak seçilmiştir ve gömülü metnin sonunu işaret eder. Belirteç tek başına görünmeli ve satırda sonda boşluk olmamalıdır.

Peki, burada belge kullanmanın avantajı nedir? Varsayılan olarak, burada belgeler içindeki tek ve çift tırnak işaretlerinin kabuk için özel anlamlarını kaybetmeleri dışında, çoğunlukla `echo` ile aynıdır. İşte bir komut satırı örneği:

```
[me@linuxbox ~]$ foo="bazı
metinler" [me@linuxbox ~]$ cat <<
_EOF_
> $foo
> "$foo"
> '$foo'
> \$foo
> _EOF_ bir
metin "bir
metin" 'bir
metin'
$foo
```

Gördüğümüz gibi, kabuk tırnak işaretlerine dikkat etmez. Onları sıradan karakterler olarak değerlendirir. Bu, burada belgeye tırnak işaretlerini serbestçe yerleştirmemizi sağlar. Bu, rapor programımız için kullanışlı olabilir.

Burada belge içindeki metin parametre genişletme, komut ikamesi, aritmetik genişletme işlemlerinden geçer ve `$` ve `\` karakterleri `\` ile kaçırılmalıdır.

Ancak, başlangıç belirtecini tırnak içine aldığımızda,

```
cat << '_EOF_'
```

tırnak işaretleri belirginden kaldırılır ve metin üzerinde hiçbir genişletme yapılmaz:

```
[me@linuxbox ~]$ foo="bazı
metinler" [me@linuxbox ~]$ cat <<
'_EOF_'
> $foo
> "$foo"
> '$foo'
> \$foo
> _EOF_
```

```
$foo
"$foo"
'$foo'
\$foo
```

Burada belgeler, standart girişi kabul eden herhangi bir komutla kullanılabilir. Bu örnekte, uzak bir FTP sunucusundan bir dosyayı almak için `ftp` programına bir dizi komut iletmek üzere burada belge kullanıyoruz:

```
#!/bin/bash

# FTP yoluyla bir dosyayı almak için komut dosyası

FTP_SERVER="ftp.nl.debian.org"
FTP_PATH="/debian/dists/bookworm/main/installer-amd64/current/images/
cdrom"
REMOTE_FILE="debian-cd_info.tar.gz"

ftp -n << _EOF_
open $FTP_SERVER
kullanıcı anonim me@linuxbox
cd $FTP_PATH
hash
get $REMOTE_FILE
bye
_EOF_
ls -l "$REMOTE_FILE"
```

Yönlendirme operatörünü `<<` yerine `<<-` olarak değiştirirsek, kabuk burada belgedeki baştaki sekme karakterlerini (ama boşlukları değil) yok sayacaktır. Bu, burada belgenin girintilendirilmesine olanak tanır ve okunabilirliği artırabilir.

```
#!/bin/bash

# FTP aracılığıyla bir dosyayı almak için komut dosyası

FTP_SERVER="ftp.nl.debian.org"
FTP_PATH="/debian/dists/bookworm/main/installer-amd64/current/images/
cdrom"
REMOTE_FILE="debian-cd_info.tar.gz"
```

```
ftp -n <<- _EOF_
    open $FTP_SERVER
    kullanıcı anonim me@linuxbox
    cd $FTP_PATH
    hash
    get $REMOTE_FILE
    bye
    _EOF_

ls -l "$REMOTE_FILE"
```

Bu özelliğin biraz sorunlu olabileceğini unutmayın, çünkü birçok metin editörü (ve programcılar) komut dosyalarında girinti yapmak için sekme yerine boşluk kullanmayı tercih ederler.

Özet

Bu bölümde, başarılı bir komut dosyası oluşturma sürecini ele alan bir projeye başladık. Değişkenler ve sabitler kavramını ve bunların nasıl kullanılabileceğini tanıttık. Bunlar, parametre genişletme için bulacağımız birçok uygulamanın ilkidir. Ayrıca, komut dosyamızdan çıktı üretme yöntemlerini ve metin bloklarını gömme için çeşitli yöntemleri inceledik.

Daha Fazla Okuma

- HTML hakkında daha fazla bilgi için aşağıdaki makalelere ve öğreticilere bakın: https://en.wikipedia.org/wiki/Html https://en.wikibooks.org/wiki/HTML_Programming
- `Bash` man sayfasında, bu özelliğin tam açıklaması bulunan "Here Documents" başlıklı bir bölüm bulunmaktadır.

26 – Yukarıdan Aşağıya Tasarım

Programlar büyüdükçe ve karmaşıklaştıkça, tasarım, kodlama ve bakımları da zorlaşır. Her büyük projede olduğu gibi, büyük ve karmaşık görevleri bir dizi küçük ve basit göreve bölmek genellikle iyi bir fikirdir. Mars'tan gelen bir kişiye, pazara gidip yiyecek almak gibi sıradan, günlük bir görevi anlatmaya çalıştığımızı hayal edelim. Genel süreci aşağıdaki adımlar dizisiyle açıklayabiliriz:

1. Arabaya bin.
2. Markete gitmek için araba sürmek.
3. Arabayı park edin.
4. Markete girin.
5. Yiyecek satın alın.
6. Arabaya dön.
7. Eve gitmek için yola çıkmak.
8. Arabayı park edin.
9. Eve girin.

Ancak, Mars'tan gelen bir kişi muhtemelen daha fazla ayrıntıya ihtiyaç duyacaktır. "Arabayı park et" alt görevini şu adımlara ayırabiliriz:

1. Park yeri bulmak.
2. Arabayı park yerine sür.
3. Motoru kapatın.
4. El frenini çek.
5. Arabadan in.
6. Arabayı kilitleyin.

"Motoru kapat" alt görevi, "kontak anahtarını kapat", "kontak anahtarını çıkar" vb. adımlara bölünebilir, ta ki markete gitme sürecinin her adımı tam olarak tanımlanana kadar.

En üst düzey adımları belirleme ve bu adımların giderek daha ayrıntılı görünümlerini geliştirme süreci, *yukarıdan aşağıya tasarım* olarak adlandırılır. Bu teknik, büyük ve karmaşık görevleri birçok küçük ve basit göreve bölmemizi sağlar. Yukarıdan aşağıya tasarım, tasarımda yaygın olarak kullanılan bir yöntemdir.

programları ve özellikle kabuk programlamaya çok uygun olan bir program.

Bu bölümde, rapor oluşturucu betiğimizi daha da geliştirmek için yukarıdan aşağıya tasarım kullanacağız.

Kabuk İşlevleri

Komut dosyamız şu anda HTML belgesini oluşturmak için aşağıdaki adımları gerçekleştiriyor:

1. Sayfayı aç.
2. Sayfa başlığını aç.
3. Sayfa başlığını ayarlayın.
4. Sayfa başlığını kapat.
5. Sayfa gövdesini aç.
6. Sayfa başlığını çıktıya yazdır.
7. Zaman damgasını çıktı.
8. Sayfa gövdesini kapat.
9. Sayfayı kapat.

Geliştirmenin bir sonraki aşamasında, 7. ve 8. adımlar arasına bazı görevler ekleyeceğiz. Bunlar arasında şunlar yer alacak:

- Sistem çalışma süresi ve yükü. Bu, son kapatma veya yeniden başlatma işleminden bu yana geçen süre ve işlemcide çeşitli zaman aralıklarında şu anda çalışmakta olan ortalama görev sayısıdır.
- Disk alanı. Bu, sistemin depolama aygıtlarındaki toplam alan kullanımıdır.
- Ana alan. Bu, her kullanıcı tarafından kullanılan depolama alanı miktarıdır.

Bu görevlerin her biri için bir komutumuz olsaydı, bunları komut ikamesi yoluyla komut dosyamıza ekleyebilirdik.

```
#!/bin/bash

# Sistem bilgisi sayfasını görüntülemek için program

TITLE="$HOSTNAME için Sistem Bilgisi Raporu"
CURRENT_TIME="$(date +"%x %r %Z")"
TIMESTAMP="$CURRENT_TIME tarihinde $USER tarafından oluşturuldu"

cat << _EOF_
<html>
```

```
        <head>
                <title>$TITLE</title>
        </head>
        <body>
                <h1>$TITLE</h1>
                <p>$TIMESTAMP</p>
                $(report_uptime)
                $(report_disk_space)
                $(report_home_space)
        </body>
</html>
_EOF_
```

Bu ek komutları iki şekilde oluşturabiliriz. Üç ayrı komut dosyası yazıp bunları PATH'imizde listelenen bir dizine yerleştirebiliriz veya komut dosyalarını programımıza *kabuk işlevleri* olarak gömebiliriz. Daha önce de belirttiğimiz gibi, kabuk işlevleri diğer komut dosyalarının içinde bulunan ve bağımsız programlar gibi davranabilen "mini komut dosyaları"dır. Kabuk işlevlerinin iki yaygın sözdizimsel biçimi vardır. İlk olarak, daha resmi olan biçim şöyledir:

```
fonksiyon adı {
      komutlar
      dönüş
}
```

İşte daha basit (ve genellikle tercih edilen) biçim:

```
ad () {
      komutlar
      dön
}
```

Bu örnekte, *name* işlevin adıdır ve *commands* işlev içinde bulunan bir dizi komuttur. Her iki biçim de eşdeğerdir ve birbirinin yerine kullanılabilir. Aşağıda, kabuk işlevinin kullanımını gösteren bir komut dosyası bulunmaktadır:

```
 1  #!/bin/bash
 2
 3  # Kabuk işlevi demosu
 4
 5  fonksiyon adım2 {
 6       echo "Adım 2"
 7       return
 8  }
 9
```

```
10  # Ana program burada başlar
11
12  echo "Adım 1"
13  adım2
14  echo "Adım 3"
```

Kabuk komut dosyasını okurken, 1 ila 11. satırları atlar çünkü bu satırlar yorumlardan ve işlev tanımından oluşur. Yürütme, 12. satırda bir `echo` komutuyla başlar. 13. satır, kabuk işlevi `step2'yi` *çağırır* ve kabuk, bu işlevi diğer komutlar gibi yürütür. Program kontrolü daha sonra 6. satıra geçer ve ikinci `echo` komutu yürütülür. Ardından 7. satır yürütülür. Bu satırdaki `return` komutu işlevi sonlandırır ve kontrolü işlev çağrısının ardından gelen satıra (14. satır) geri döndürür ve son `echo` komutu yürütülür. İşlev çağrılarının kabuk işlevleri olarak tanınması ve harici programların adları olarak yorumlanmaması için, kabuk işlev tanımlarının çağrılmadan önce komut dosyasında görünmesi gerektiğine dikkat edin.

Burada gösterilen minimal kabuk fonksiyon tanımlarını betiğimize ekleyeceğiz:

```
#!/bin/bash

# Sistem bilgisi sayfasını görüntüleyen program

TITLE="$HOSTNAME için Sistem Bilgisi Raporu"
CURRENT_TIME="$(date +"%x %r %Z")"
TIMESTAMP="$CURRENT_TIME tarihinde, $USER tarafından oluşturuldu"

report_uptime () {
    return
}

report_disk_space () {
    return
}

report_home_space () {
    return
}

cat << _EOF_
<html>
    <head>
         <title>$TITLE</title>
```

```
    </head>
    <body>
         <h1>$TITLE</h1>
         <p>$TIMESTAMP</p>
         $(report_uptime)
         $(report_disk_space)
         $(report_home_space)
    </body>
</html>
_EOF_
```

Kabuk işlev adları, değişkenlerle aynı kurallara uyar. Bir işlev en az bir komut içermelidir. `Return` komutu (isteğe bağlıdır) bu gereksinimi karşılar.

Yerel Değişkenler

Şimdiye kadar yazdığımız komut dosyalarında, tüm değişkenler (sabitler dahil) *küresel değişkenlerdi*. Küresel değişkenler, program boyunca varlıklarını sürdürürler. Bu, birçok durumda sorun oluşturmaz, ancak bazen kabuk işlevlerinin kullanımını karmaşıklaştırabilir. Kabuk işlevlerinin içinde, genellikle *yerel değişkenlerin* kullanılması tercih edilir. Yerel değişkenlere, yalnızca tanımlandıkları kabuk işlevi ve kabuk işlevinin çağırabileceği diğer işlevler içinde erişilebilir. Kabuk işlevi sona erdiğinde, yerel değişkenler de ortadan kalkar.

Yerel değişkenlerin olması, programcının potansiyel isim çakışmaları konusunda endişelenmeden, komut dosyasında genel olarak veya diğer kabuk işlevlerinde zaten mevcut olan isimlere sahip değişkenleri kullanmasına olanak tanır.

Aşağıda, yerel değişkenlerin nasıl tanımlandığını ve kullanıldığını gösteren bir komut dosyası örneği verilmiştir:

```
#!/bin/bash

# local-vars: yerel değişkenleri gösteren komut dosyası

foo=0    # global değişken foo

funct_1 () {

    local foo # funct_1 için yerel değişken

    foo=1
    echo "funct_1: foo = $foo"
}
```

```
funct_2 () {

    yerel foo # funct_2 için yerel değişken

    foo=2
    echo "funct_2: foo = $foo"
}

echo "global:   foo = $foo"
funct_1
echo "global:   foo = $foo"
funct_2
echo "global:   foo = $foo"
```

Gördüğümüz gibi, yerel değişkenler değişken adının önüne `local` kelimesini ekleyerek tanımlanır. Bu, tanımlandığı kabuk işlevine yerel bir değişken oluşturur. Kabuk işlevinin dışına çıktığında, değişken artık mevcut değildir. Bu komut dosyasını çalıştırdığımızda şu sonuçları görürüz:

```
[me@linuxbox ~]$ local-vars
global: foo = 0
funct_1: foo = 1
global: foo = 0
funct_2: foo = 2
global: foo = 0
```

Her iki kabuk işlevinde yerel değişken `foo'ya` değer atamanın, işlevlerin dışında tanımlanan `foo` değerine hiçbir etkisi olmadığını görüyoruz.

Bu özellik, kabuk işlevlerinin birbirlerinden ve içinde bulundukları betikten bağımsız kalacak şekilde yazılmasını sağlar. Bu, bir programın bir bölümünün diğerini etkilemesini önlemeye yardımcı olduğu için çok değerlidir. Ayrıca, kabuk işlevlerinin taşınabilir olacak şekilde yazılmasını da sağlar. Yani, gerektiğinde betikten betiğe kesilip yapıştırılabilirler.

Kabuk İşlevleri ve Yeniden Yönlendirme

Kabuk işlevlerinin nasıl yazıldığına daha yakından bakarsak, Bölüm 6'da değindiğimiz bir şeyi fark edebiliriz:

```
my_funct () {
```

```
    komut1
    komut2
    komut3
}
```

Küme parantezleri içindeki üç komut bir *grup komutu* oluşturur. 6. bölümden hatırlayacağımız gibi, grup komutları yeniden yönlendirme söz konusu olduğunda birden fazla komutu tek bir varlıkta birleştirir. Grup komutlarıyla her ikisini de yapabiliriz:

```
{ komut1; komut2; komut3; } > some_output.txt
```

ve:

```
{ komut1; komut2; komut3; } < some_input.txt
```

Aynı durum kabuk işlevleri için de geçerlidir. Aşağıdaki kodu ele alalım:

```
my_funct () {
    echo "Belgelerim" ls
    ~/Belgeler
    echo "Müziğim" ls
    ~/Müzik
    echo "Videolarım"
    ls ~/Videos
    return
}
```

Bu fonksiyonun ne yaptığını anlamak kolaydır, ancak çıktısı nereye gider? Biz yönlendirdiğimiz yere gider. Bu fonksiyonu çağırdığımızda, birleştirilmiş çıktısını standart çıktıya gönderir ve istersek bir dosyaya yönlendirebiliriz:

```
my_funct > my_directories.txt
```

Veya bir boru hattına:

```
my_funct | sort
```

Komut ikamesi kullanarak çıktıyı bir değişkende bile saklayabiliriz:

```
my_var="$(my_funct)"
```

Yönlendirme, standart girişe de uygulanır. İşlev, standart girişi kabul eden bir komut içeriyorsa (örneğin, argüman içermeyen `cat`), bunu kolayca yapabiliriz:

```
my_funct < input.txt
```

Komut Dosyalarını Çalışır Halde Tutma

Programımızı geliştirirken, programı çalıştırılabilir durumda tutmak yararlıdır. Bunu yaparak ve sık sık test ederek, geliştirme sürecinin erken aşamalarında hataları tespit edebiliriz. Bu, hata ayıklama sorunlarını çok daha kolay hale getirecektir. Örneğin, programı çalıştırırsak, küçük bir değişiklik yaparsak ve ardından programı tekrar çalıştırıp bir sorun bulursak, en son yapılan değişikliğin sorunun kaynağı olması muhtemeldir. Programcıların dilinde *"stub"* olarak adlandırılan boş işlevler ekleyerek, programımızın mantıksal akışını erken bir aşamada doğrulayabiliriz. Bir stub oluştururken, mantıksal akışın gerçekleştirildiğini gösteren ve programcıya geri bildirim sağlayan bir şey eklemek iyi bir fikirdir. Şimdi komut dosyasının çıktısına bakarsak, zaman damgasından sonra çıktımızda bazı boş satırlar olduğunu görürüz, ancak bunun nedenini kesin olarak bilemeyiz.

```
[me@linuxbox ~]$ sys_info_page
<html>
    <head>
         <title>twin2 için Sistem Bilgisi Raporu</title>
    </head>
    <body>
         <h1>linuxbox için Sistem Bilgisi Raporu</h1>
         <p>19.03.2009 16:02:10 EDT tarihinde tarafımdan oluşturulmuştur</p>



    </body>
</html>
```

Bazı geri bildirimleri içerecek şekilde işlevleri değiştirirsek

```
report_uptime () {
    echo "Fonksiyon report_uptime çalıştırıldı."
    return
}

report_disk_space () {
    echo "report_disk_space işlevi çalıştırıldı."
    dön
}

report_home_space () {
    echo "report_home_space işlevi yürütüldü."
    dön
}
```

ve komut dosyasını tekrar çalıştırın

```
[me@linuxbox ~]$ sys_info_page
<html>
    <head>
        <title>linuxbox için Sistem Bilgisi Raporu</title>
    </head>
    <body>
        <h1>linuxbox için Sistem Bilgisi Raporu</h1>
        <p>03/20/2009 05:17:26 AM EDT tarihinde, tarafımdan
        oluşturulmuştur</p> report_uptime işlevi çalıştırıldı.
        report_disk_space işlevi yürütüldü.
        report_home_space işlevi yürütüldü.
    </body>
</html>
```

Şimdi, aslında üç fonksiyonumuzun da çalıştırıldığını görüyoruz.

İşlev çerçevemiz hazır ve çalışır durumda olduğuna göre, şimdi bazı işlev kodlarını detaylandırma zamanı geldi. İlk olarak, `report_uptime` işlevini inceleyelim:

```
report_uptime () {
    cat << _EOF_
        <h2>Sistem Çalışma Süresi</h2>
        <pre>$(uptime)</pre>
_EOF_
```

```
    return
}
```

Oldukça basit. Burada bir belge kullanarak bir bölüm başlığı ve `uptime` komutunun çıktısını, komutun biçimini korumak için `<pre>` etiketleriyle çevreleyerek görüntülüyoruz. `report_disk_space` işlevi de benzerdir.

```
report_disk_space () {
    cat << _EOF_
         <h2>Disk Alanı Kullanımı</h2>
         <pre>$(df -h)</pre>
_EOF_
    return
}
```

Bu işlev, disk alanının miktarını belirlemek için `df -h` komutunu kullanır. Son olarak, `report_home_space` işlevini oluşturacağız.

```
report_home_space () {
    cat << _EOF_
         <h2>Ana Sayfa Alanı Kullanımı</h2>
         <pre>$(du -sh /home/*)</pre>
_EOF_
    dön
}
```

Bu görevi gerçekleştirmek için -sh seçenekleriyle `du` komutunu kullanıyoruz. Ancak bu, soruna tam bir çözüm değildir. Bazı sistemlerde (örneğin Ubuntu) çalışsa da, diğerlerinde çalışmayacaktır. Bunun nedeni, birçok sistemin ev dizinlerinin izinlerini herkesin okuyabileceği şekilde ayarlamasıdır, ki bu makul bir güvenlik önlemidir. Bu sistemlerde, yazıldığı şekliyle `report_home_space` işlevi, yalnızca komut dosyamız süper kullanıcı ayrıcalıklarıyla çalıştırıldığında çalışacaktır. Daha iyi bir çözüm, komut dosyasının davranışını kullanıcının ayrıcalıklarına göre ayarlaması olacaktır. Bu konuyu bir sonraki bölümde ele alacağız.

`.bashrc` Dosyanızdaki Kabuk İşlevleri

Kabuk işlevleri, takma adların yerine mükemmel birer alternatif oluşturur ve aslında kişisel kullanım için küçük komutlar oluşturmanın tercih edilen yöntemidir. Takma adlar, destekledikleri komut ve kabuk özellikleri açısından sınırlıdır, oysa kabuk işlevleri komut dosyası ile yazılabilen her şeyi destekler. Örneğin, komut dosyamız için geliştirdiğimiz `report_disk_space` kabuk işlevini beğendiysem, `.bashrc` dosyamız için `ds` adında benzer bir işlev oluşturabiliriz:

```
ds () {
   echo "$HOSTNAME için Disk Alanı Kullanımı" df
   -h
}
```

Özet

Bu bölümde, yukarıdan aşağıya tasarım adı verilen yaygın bir program tasarım yöntemini tanıttık ve bunun gerektirdiği adım adım iyileştirmeyi oluşturmak için kabuk işlevlerinin nasıl kullanıldığını gördük. Ayrıca, yerel değişkenlerin kabuk işlevlerini birbirinden ve yerleştirildikleri programdan bağımsız hale getirmek için nasıl kullanılabileceğini gördük. Bu, kabuk işlevlerinin taşınabilir bir şekilde yazılmasını ve birden fazla programa yerleştirilebilmesini sağlayarak *yeniden kullanılabilir* olmasını mümkün kılar; bu da büyük bir zaman tasarrufu sağlar.

Daha Fazla Okuma

- Wikipedia'da yazılım tasarım felsefesi hakkında birçok makale bulunmaktadır. İşte bunlardan birkaç tanesi:
  https://en.wikipedia.org/wiki/Top-down_design
  https://en.wikipedia.org/wiki/Function_(computer_programming)

27 – Akış Kontrolü: `if` ile Dallanma

Önceki bölümde bir sorunla karşılaştık. Rapor oluşturma betiğimizi, betiği çalıştıran kullanıcının ayrıcalıklarına nasıl uyarlayabiliriz? Bu sorunun çözümü, bir testin sonuçlarına göre betiğimizde "yön değiştirme" yolunu bulmamızı gerektirecektir. Programlama terimleriyle, programın *dallanması* gerekir.

İnsanların kullanımı için tasarlanmış bir bilgisayar dilinin simülasyonu olan *sözde kod*da ifade edilen basit bir mantık örneğini ele alalım.

X = 5

X = 5 ise, o zaman:

"X, 5'e eşittir" deyin. Aksi

takdirde:

"X 5'e eşit değildir" deyin.

Bu, bir dallanma örneğidir. X'in değerini değerlendiririz ve X = 5 ise "X eşittir 5" deriz, aksi takdirde "X eşittir 5 değildir" deriz.

`if`

Kabuğu kullanarak, önceki mantığı şu şekilde kodlayabiliriz:

```
x=5

if [ "$x" -eq 5 ]; then
    echo "x 5'e eşittir."
else
    echo "x, 5'e eşit değildir."
fi
```

Ya da komut satırına doğrudan girebiliriz (biraz kısaltılmış olarak):

```
[me@linuxbox ~]$ x=5
[me@linuxbox ~]$ if [ "$x" -eq 5 ]; then echo "5'e eşittir"; else echo
"5'e eşit değildir"; fi
5'e eşittir
[me@linuxbox ~]$ x=0
[me@linuxbox ~]$ if [ "$x" -eq 5 ]; then echo "5'e eşittir"; else echo
"5'e eşit değildir"; fi
5'e eşit değil
```

Bu örnekte, komutu iki kez çalıştırıyoruz. İlkinde `x` değeri 5 olarak ayarlanmış ve sonuç olarak "5'e eşittir" dizesi çıktısı alınmış, ikinci kez ise `x` değeri 0 olarak ayarlanmış ve sonuç olarak "5'e eşit değildir" dizesi çıktısı alınmış.

`if` bileşik komutunun sözdizimi aşağıdaki gibidir; burada *commands* komutların listesidir:

```
if komutlar; then
     komutlar
[elif komutlar; then
     komutlar...]
[else
     komutlar]
fi
```

İlk bakışta bu biraz kafa karıştırıcı olabilir. Ancak bunu açıklığa kavuşturmadan önce, kabuğun bir komutun başarısını veya başarısızlığını nasıl değerlendirdiğine bakmamız gerekir.

Çıkış Durumu

Komutlar (yazdığımız betikler ve kabuk işlevleri dahil), sonlandıklarında sisteme bir değer gönderir; buna *çıkış durumu* denir. 0 ile 255 arasında bir tamsayı olan bu değer, komutun yürütülmesinin başarılı olup olmadığını gösterir. Geleneksel olarak, sıfır değeri başarıyı, diğer değerler ise başarısızlığı gösterir. Kabuk, bir komutun çıkış durumunu incelemek için kullanabileceğimiz bir parametre sağlar. İşte bunun nasıl çalıştığını görelim:

```
[me@linuxbox ~]$ ls -d /usr/bin
/usr/bin
[me@linuxbox ~]$ echo $?
0
[me@linuxbox ~]$ ls -d /bin/usr
ls: /bin/usr'ye erişilemiyor: Böyle bir dosya veya dizin yok
[me@linuxbox ~]$ echo $?
2
```

Bu örnekte, `ls` komutunu iki kez çalıştırıyoruz. İlk seferinde komut başarıyla çalışıyor. `$?` parametresinin değerini görüntülersek, bunun sıfır olduğunu görürüz. `Ls` komutunu ikinci kez çalıştırıyoruz (var olmayan bir dizin belirtiyoruz), bir hata üretiyoruz ve `$?` parametresini tekrar inceliyoruz. Bu sefer parametre 2 değerini içeriyor, bu da komutun bir hatayla karşılaştığını gösteriyor. Bazı komutlar, hataların tanılama için farklı çıkış durumu değerleri kullanırken, birçok komut başarısız olduğunda basitçe 1 değeriyle çıkar. Man sayfaları genellikle hangi kodların kullanıldığını açıklayan "Çıkış Durumu" başlıklı bir bölüm içerir. Ancak, sıfır her zaman başarıyı gösterir.

Kabuk, 0 veya 1 çıkış durumu ile sonlanmaktan başka hiçbir şey yapmayan iki son derece basit yerleşik komut sağlar. `true` komutu her zaman başarıyla yürütülür ve `false` komutu her zaman başarısız olarak yürütülür.

```
[me@linuxbox ~]$ true
[me@linuxbox ~]$ echo $?
0
[me@linuxbox ~]$ false
[me@linuxbox ~]$ echo $?
1
```

Bu komutları kullanarak `if` ifadesinin nasıl çalıştığını görebiliriz. `if` ifadesinin gerçekte yaptığı şey, komutların başarısını veya başarısızlığını değerlendirmektir.

```
[me@linuxbox ~]$ if true; then echo "Doğru."; fi Doğru.
[me@linuxbox ~]$ if false; then echo "Doğru."; fi [me@linuxbox
~]$
```

`if komutunu` izleyen komut başarıyla yürütüldüğünde `echo "Doğru."` komutu yürütülür, `if komutunu` izleyen komut başarıyla yürütülemediğinde ise yürütülmez. `if komutunu` bir komut listesi izliyorsa, listedeki son komut değerlendirilir:

```
[me@linuxbox ~]$ if false; true; then echo "Doğru."; fi Doğru.
[me@linuxbox ~]$ if true; false; then echo "Doğru."; fi
[me@linuxbox ~]$
```

`Test` kullanımı

Şimdiye kadar, `if` ile en sık kullanılan komut `test` komutudur. `Test` komutu çeşitli kontroller ve karşılaştırmalar gerçekleştirir. İki eşdeğer biçimi vardır. İlk biçim şöyledir:

`test` *`expression`*

İkinci ve daha popüler olan form ise şöyledir:

`[` *`ifade`* `]`

ifade, doğru veya yanlış olarak değerlendirilen bir ifadedir. `test` komutu, ifade doğru olduğunda 0 çıkış durumu, ifade yanlış olduğunda ise 1 çıkış durumu döndürür.

Hem `test` hem de `[` komutlarının aslında komutlar olması ilginçtir. `Bash'ta` bunlar yerleşik komutlardır, ancak diğer kabuklarla kullanılmak üzere `/usr/bin` dizininde programlar olarak da bulunurlar. İfade aslında sadece argümanlarından oluşur ve `[` komutu, `]` karakterinin son argüman olarak sağlanmasını gerektirir.

`test` ve `[` komutları çok çeşitli yararlı ifadeler ve testleri destekler.

Dosya İfadeleri

Tablo 27-1, dosyaların durumunu değerlendirmek için kullanılan ifadeleri listeler.

Tablo 27-1: `Test` Dosyası İfadeleri

İfade	Aşağıdaki durumlarda doğrudur...
`file1` `-ef` *`file2`*	*file1* ve *file2* aynı inode numaralarına sahipse (iki dosya adı sabit bağlantı ile aynı dosyayı ifade ediyorsa).
`file1` `-nt` *`file2`*	*file1, file2*'den daha yenidir.
`file1` `-ot` *`file2`*	*file1, file2*'den daha eskidir.
`-b` *`dosya`*	*dosya* mevcuttur ve blok özel (aygıt) dosyasıdır.
`-c` *`dosya`*	*dosya* var ve karakter özel (aygıt) dosyasıdır.
`-d` *`dosya`*	*dosya* var ve bir dizindir.
`-e` *`dosya`*	*dosya* mevcuttur.
`-f` *`dosya`*	*dos*ya var ve normal bir dosyadır.
`-g` *`dosya`*	*dosya* mevcut ve grup kimliği ayarlanmış.
`-G` *`dosya`*	*dosya* mevcut ve etkin grup kimliğine ait.
`-k` *`dosya`*	*dos*ya mevcut ve "sticky bit" ayarlı.

`-L` *dosya*	*dos*ya var ve sembolik bir bağlantı.
`-` *dosya*	*dosya* var ve etkin kullanıcı kimliğine ait.
`-p` *dosya*	*dosya* mevcuttur ve adlandırılmış bir borudur.
`-` *dosya*	*dosya* var ve okunabilir (etkin kullanıcı için okunabilir izni var).
`-` *dosya*	*dos*ya mevcut ve uzunluğu sıfırdan büyük.
`-S` *dosya*	*dosyası* var ve bir ağ soketi.
`-t` *fd*	*fd* terminale yönlendirilen/terminalden gelen bir dosya tanımlayıcısıdır. Bu, standart giriş/çıkış/hatanın yeniden yönlendirilip yönlendirilmediğini belirlemek için kullanılabilir.
`-` *dosya*	*dosya* mevcuttur ve setuid ayarlıdır.
`-w` *dosya*	*dosya* mevcut ve yazılabilir (etkin kullanıcı için yazma izni var).
`-` *dosya*	*dosya* mevcut ve çalıştırılabilir (etkin kullanıcı için çalıştırma/arama izni var).

Burada, bazı dosya ifadelerini gösteren bir komut dosyası bulunmaktadır:

```
#!/bin/bash

# test-file: Bir dosyanın durumunu değerlendir

FILE=~/.bashrc

if [ -e "$FILE" ]; then
    if [ -f "$FILE" ]; then
         echo "$FILE normal bir dosyadır."
    fi
    if [ -d "$FILE" ]; then
         echo "$FILE bir dizindir."
    fi
    if [ -r "$FILE" ]; then
         echo "$FILE okunabilir."
    fi
    if [ -w "$FILE" ]; then
```

```
          echo "$FILE yazılabilir."
    fi
    if [ -x "$FILE" ]; then
          echo "$FILE çalıştırılabilir/arama yapılabilir."
    fi
else
    echo "$FILE mevcut değil"
    exit 1
fi

çık
```

Komut dosyası, `FILE` sabitine atanan dosyayı değerlendirir ve değerlendirme gerçekleştirilirken sonuçları görüntüler. Bu komut dosyası hakkında dikkat edilmesi gereken iki ilginç nokta vardır. İlk olarak, `$FILE` parametresinin ifadeler içinde nasıl tırnak içine alındığına dikkat edin. Bu, ifadenin sözdizimsel olarak tamamlanması için gerekli değildir; daha çok, parametrenin boş olması veya yalnızca boşluk içermesi durumuna karşı bir önlemdir. `$FILE` parametresinin genişletilmesi boş bir değerle sonuçlanırsa, bir hata oluşur (işlemciler, işlemciler olarak değil, boş olmayan dizeler olarak yorumlanır). Parametrenin etrafındaki tırnak işaretleri, dize boş olsa bile operatörün her zaman bir dize ile takip edilmesini sağlar. İkincisi, komut dosyasının sonuna yakın `exit` komutunun varlığına dikkat edin. `exit` komutu, komut dosyasının çıkış durumu haline gelen tek bir isteğe bağlı argüman kabul eder. Hiçbir argüman iletilmediğinde, çıkış durumu varsayılan olarak son yürütülen komutun çıkış durumu olur. `exit komutunu` bu şekilde kullanmak, `$FILE` mevcut olmayan bir dosyanın adına genişlerse komut dosyasının hatayı göstermesini sağlar. Komut dosyasının son satırında görünen `exit` komutu formalite icabı oradadır. Bir komut dosyası "sonuna ulaştığında" (dosyanın sonuna ulaştığında), son yürütülen komutun çıkış durumu ile sonlanır.

Benzer şekilde, kabuk işlevleri de return komutuna bir tamsayı argümanı ekleyerek bir çıkış durumu döndürebilir. Önceki betiği daha büyük bir programa dahil etmek için bir kabuk işlevine dönüştürmek istersek, `exit` komutlarını `return` ifadeleriyle değiştirebilir ve istenen davranışı elde edebiliriz.

```
test_file () {

    # test-file: Bir dosyanın durumunu değerlendir

    FILE=~/.bashrc

    if [ -e "$FILE" ]; then
          if [ -f "$FILE" ]; then
```

```
                echo "$FILE normal bir dosyadır."
        fi
        if [ -d "$FILE" ]; then
                echo "$FILE bir dizindir."
        fi
        if [ -r "$FILE" ]; then
                echo "$FILE okunabilir."
        fi
        if [ -w "$FILE" ]; then
                echo "$FILE yazılabilir."
        fi
        if [ -x "$FILE" ]; then
                echo "$FILE çalıştırılabilir/arama yapılabilir."
        fi
    else
        echo "$FILE mevcut değil" return
        1
    fi

}
```

Dize İfadeleri

Tablo 27-2, dizeleri değerlendirmek için kullanılan ifadeleri listeler:

Tablo 27-2: `test` Dize İfadeleri

İfade	Aşağıdaki durumlarda doğrudur...
`dize`	*dize* null değilse.
`-n` *`string`*	*string'in* uzunluğu sıfırdan büyükse.
`-z` *`string`*	*Dizenin* uzunluğu sıfırdır.
`string1 = string2` *`string1 == string2`*	*string1* ve *string2* eşittir. Tek veya çift eşittir işareti kullanılabilir. Çift eşittir işaretinin kullanımı `bash` tarafından desteklenir ve genellikle tercih edilir, ancak POSIX uyumlu değildir.
`string1` `!=` *`string2`*	*string1* ve *string2* eşit değildir.
`string1` `>` *`string2`*	*string1, string2'den* sonra gelir.
`string1` `<` *`string2`*	*string1, string2'den* önce gelir.

Uyarı: > ve < ifade operatörleri `test` ile kullanıldığında tırnak içine alınmalı (veya ters eğik çizgi ile kaçış karakteri eklenmelidir). Aksi takdirde, kabuk tarafından yeniden yönlendirme operatörleri olarak yorumlanacak ve potansiyel olarak yıkıcı sonuçlar doğuracaktır. Ayrıca, `bash` belgelerinde sıralama sırasının geçerli yerel ayarın sıralama sırasına uygun olduğu belirtilse de, bu her zaman geçerli olmayabilir. ASCII (POSIX) sırası, 4.0 dahil olmak üzere `bash` sürümlerinde kullanılır. Bu sorun 4.1 sürümünde düzeltilmiştir.

Aşağıda, dize ifadeleri içeren bir komut dosyası bulunmaktadır:

```
#!/bin/bash

# test-string: bir dizgenin değerini değerlendir

ANSWER=belki

if [ -z "$ANSWER" ]; then
    echo "Cevap yok." >&2 exit 1
fi

if [ "$ANSWER" == "yes" ]; then
    echo "Cevap EVET."
elif [ "$ANSWER" == "hayır" ]; then
    echo "Cevap HAYIR."
elif [ "$ANSWER" == "belki" ]; then
    echo "Cevap BELKİ."
else
    echo "Cevap BİLİNMİYOR."
fi
```

Bu komut dosyasında, `ANSWER` sabitini değerlendiriyoruz. Önce dizgenin boş olup olmadığını belirliyoruz. Boşsa, komut dosyasını sonlandırıyor ve çıkış durumunu 1 olarak ayarlıyoruz. Echo komutuna uygulanan yönlendirmeye dikkat edin. Bu, "Cevap yok." hata mesajını standart hataya yönlendirir, ki bu hata mesajları için doğru olan şeydir. Dize boş değilse, dizenin değerini değerlendirerek "yes", "no" veya "maybe" ile eşit olup olmadığını kontrol ederiz. Bunu, "else if" ifadesinin kısaltması olan `elif` kullanarak yaparız. `Elif` kullanarak daha karmaşık bir mantıksal test oluşturabiliriz.

Tamsayı İfadeleri

Değerleri dize olarak değil tamsayı olarak karşılaştırmak için,

Tablo 27-3'te listelenen ifadeleri kullanabiliriz.

Tablo 27-3: `test` Tamsayı İfadeleri

İfade	Aşağıdaki durumlarda doğrudur...
integer1 `-eq` *integer2*	*tamsayı1, tamsayı2'ye* eşittir. *tamsayı1*
`-ne` *tamsayı2*	*tamsayı1, tamsayı2'*ye eşit değildir.
tamsayı1 `-le` *tamsayı2*	*tamsayı1, tamsayı2*'den küçük veya ona
eşittir. *tamsayı1* `-lt` *tamsayı2*	*tamsayı1, tamsayı2*'den küçüktür.
tamsayı1 `-ge` *tamsayı2*	*integer1*, *integer2'den* büyük veya ona eşittir.
tamsayı1 `-gt` *tamsayı2*	*integer1, integer2*'den büyüktür.

Bunları gösteren bir komut dosyası aşağıda verilmiştir:

```
#!/bin/bash

# test-integer: bir tamsayının değerini değerlendirir.

INT=-5

if [ -z "$INT" ]; then
    echo "INT boş." >&2 exit 1
fi

if [ "$INT" -eq 0 ]; then
    echo "INT sıfırdır."
else
    if [ "$INT" -lt 0 ]; then echo
         "INT negatif."
    else
         echo "INT pozitif."
    fi
    if [ $((INT % 2)) -eq 0 ]; then
         echo "INT çift sayıdır."
    else
         echo "INT tek sayıdır."
    fi
fi
```

Bu komut dosyasının ilginç yanı, bir tam sayının çift mi tek mi olduğunu nasıl belirlediğidir. Sayı üzerinde 2 modülü işlemi gerçekleştirerek, sayıyı 2'ye böler ve kalanı döndürür, böylece sayının tek mi çift mi olduğunu belirleyebilir.

`Testin` Daha Modern Bir Versiyonu

Modern `bash` sürümleri, `test` komutunun gelişmiş bir yerine geçen bileşik bir komut içerir. Bu komut, `test komutunda` olduğu gibi, *ifade'nin* doğru veya yanlış bir sonuç veren bir ifade olduğu aşağıdaki sözdizimini kullanır. :

`[[ ifade ]]`

`[[ ]]` komutu `test` komutuna benzer (tüm ifadeleri destekler), ancak önemli bir yeni dize ifadesi ekler.

`string1 =~ regex`

Bu, *string1* genişletilmiş düzenli ifade *regex* ile eşleşirse doğru sonucunu döndürür. Bu, veri doğrulama gibi görevleri gerçekleştirmek için birçok olasılık sunar. Daha önceki tamsayı ifadeleri örneğimizde, `INT` sabiti tamsayı dışında herhangi bir şey içeriyorsa komut dosyası başarısız olurdu. Komut dosyasının sabitin tamsayı içerdiğini doğrulamak için bir yola ihtiyacı vardır. `=~` dize ifadesi operatörü ile `[[ ]]` kullanarak komut dosyasını şu şekilde iyileştirebiliriz:

```
#!/bin/bash

# test-integer2: bir tamsayının değerini değerlendirir.

INT=-5

if [[ "$INT" =~ ^-?[0-9]+$ ]]; then
    if [ "$INT" -eq 0 ]; then
         echo "INT sıfırdır."
    else
         if [ "$INT" -lt 0 ]; then echo
              "INT negatif."
         else
              echo "INT pozitif."
         fi
         if [ $((INT % 2)) -eq 0 ]; then
              echo "INT çift sayıdır."
         else
              echo "INT tek sayıdır."
         fi
```

```
    fi
else
    echo "INT bir tamsayı değildir." >&2
    exit 1
fi
```

Düzenli ifadeyi uygulayarak, `INT` değerini isteğe bağlı bir eksi işaretiyle başlayan ve ardından bir veya daha fazla rakam gelen dizelerle sınırlayabiliriz. Bu ifade aynı zamanda boş değerlerin olasılığını da ortadan kaldırır.

`[[ ]]`'nin bir başka ek özelliği de == operatörünün, yol adı genişletmeyle aynı şekilde desen eşleştirmeyi desteklemesidir. İşte bir örnek:

```
[me@linuxbox ~]$ FILE=foo.bar
[me@linuxbox ~]$ if [[ $FILE == foo.* ]]; then
> echo "$FILE 'foo.*' deseniyle eşleşiyor"
> fi
foo.bar 'foo.*' deseniyle eşleşiyor
```

Bu, `[[ ]]'yi` dosya ve yol adlarını değerlendirmek için kullanışlı hale getirir.

`(( ))` - Tamsayılar için tasarlanmıştır

`[[ ]]` bileşik komutuna ek olarak, `bash` ayrıca tamsayılar üzerinde işlem yapmak için yararlı olan `(( ))` bileşik komutunu da sağlar. Bu komut, aritmetik değerlendirmelerin tam setini destekler; bu konuyu 34. bölümde ayrıntılı olarak ele alacağız.

`(( ))` *aritmetik doğruluk testleri* yapmak için kullanılır. Aritmetik doğruluk testi, aritmetik değerlendirmenin sonucu sıfırdan farklı ise doğru sonucunu verir.

```
[me@linuxbox ~]$ if ((1)); then echo "Doğrudur."; fi
Doğrudur.
[me@linuxbox ~]$ if ((0)); then echo "Doğru."; fi [me@linuxbox
~]$
```

`(( ))` kullanarak, `test-integer2` komut dosyasını şu şekilde biraz basitleştirebiliriz:

```
#!/bin/bash
```

```
# test-integer2a: bir tamsayının değerini değerlendirir.

INT=-5

if [[ "$INT" =~ ^-?[0-9]+$ ]]; then
    if ((INT == 0)); then
         echo "INT sıfırdır."
    else
         if ((INT < 0)); then
              echo "INT negatif."
         else
              echo "INT pozitif."
         fi
         if (( ((INT % 2)) == 0)); then
              echo "INT çift sayıdır."
         else
              echo "INT tek sayıdır."
         fi
    fi
else
    echo "INT tamsayı değildir." >&2
    exit 1
fi
```

Küçük ve büyük işaretleri kullandığımızı ve eşdeğerliği test etmek için == kullanıldığını unutmayın. Bu, tamsayılarla çalışmak için daha doğal görünen bir sözdizimidir. Ayrıca, bileşik komut `(( ))` sıradan bir komut değil, kabuk sözdiziminin bir parçası olduğu ve yalnızca tamsayılarla çalıştığı için, değişkenleri adlarına göre tanıyabildiğini ve genişletme işlemi gerektirmediğini unutmayın. `(( ))` ve ilgili aritmetik genişletmeyi 34. bölümde daha ayrıntılı olarak ele alacağız.

İfadelerin Birleştirilmesi

Daha karmaşık değerlendirmeler oluşturmak için ifadeleri birleştirmek de mümkündür. İfadeler, mantıksal operatörler kullanılarak birleştirilir. Bunları, `find` komutunu öğrendiğimiz 17. bölümde görmüştük. `test` ve `[[ ]]` için üç mantıksal işlem vardır. Bunlar AND, OR ve NOT'tur. `test` ve `[[ ]]` bu işlemleri temsil etmek için farklı operatörler kullanır:

Tablo 27-4: Mantıksal Operatörler

İşlem	test	[[ ]] ve (( ))

AND	`-a`	`&&`
VEYA	`-o`	`||`
NOT	!	!

İşte bir AND işleminin örneği. Aşağıdaki komut dosyası, bir tamsayının bir değer aralığı içinde olup olmadığını belirler:

```
#!/bin/bash

# test-integer3: bir tamsayının # belirtilen değer
aralığı içinde olup olmadığını belirler.

MIN_VAL=1
MAX_VAL=100

INT=50

if [[ "$INT" =~ ^-?[0-9]+$ ]]; then
    if [[ "$INT" -ge "$MIN_VAL" && "$INT" -le "$MAX_VAL" ]]; then
         echo "$INT, $MIN_VAL ile $MAX_VAL arasındadır."
    else
         echo "$INT aralık dışı."
    fi
else
   echo "INT bir tamsayı değil." >&2 exit
   1
fi
```

Bu komut dosyasında, `INT` tamsayısının değerinin `MIN_VAL` ve `MAX_VAL` değerleri arasında olup olmadığını belirliyoruz. Bu, `&&` operatörü ile ayrılmış iki ifadeyi içeren `[[ ]]` ifadesinin tek bir kullanımıyla gerçekleştirilir. Bunu `test` kullanarak da kodlayabilirdik:

```
    if [ "$INT" -ge "$MIN_VAL" -a "$INT" -le "$MAX_VAL" ]; then echo
         "$INT, $MIN_VAL ile $MAX_VAL aralığındadır."
    else
         echo "$INT aralık dışındadır."
    fi
```

`!` olumsuzlama operatörü, bir ifadenin sonucunu tersine çevirir. Bir ifade yanlışsa doğru, ifade doğruysa yanlış sonucunu döndürür. Aşağıdaki komut dosyasında, belirtilen aralığın dışında kalan `INT` değerlerini bulmak için değerlendirme mantığımızı değiştiriyoruz:

```
#!/bin/bash

# test-integer4: bir tamsayının belirtilen değer aralığı
dışında olup olmadığını belirler.

MIN_VAL=1
MAX_VAL=100

INT=50

if [[ "$INT" =~ ^-?[0-9]+$ ]]; then
    if [[ ! ("$INT" -ge "$MIN_VAL" && "$INT" -le "$MAX_VAL") ]]; then echo
         "$INT, $MIN_VAL ile $MAX_VAL aralığının dışındadır."
    else
         echo "$INT aralık içindedir."
    fi
else
    echo "INT bir tamsayı değil." >&2
    exit 1
fi
```

Gruplama için ifadenin etrafına parantezler de ekliyoruz. Bunlar eklenmezse, olumsuzlama sadece ilk ifadeye uygulanır, ikisinin birleşimine uygulanmaz. Bunu `test` ile kodlamak şu şekilde yapılır:

```
    if [ ! \( "$INT" -ge "$MIN_VAL" -a "$INT" -le "$MAX_VAL" \) ]; then
         echo "$INT, $MIN_VAL ile $MAX_VAL aralığının dışındadır."
    else
         echo "$INT aralık içindedir."
    fi
```

`Test` tarafından kullanılan tüm ifadeler ve operatörler kabuk tarafından komut argümanları olarak değerlendirildiğinden (`[[ ]]` ve `(( ))` 'den farklı olarak), <, >, `(` ve `)` gibi `bash` için özel anlamı olan karakterler tırnak içine alınmalı veya kaçış karakteri ile belirtilmelidir.

`test` ve `[[ ]]` kabaca aynı şeyi yaptığını gördüğümüzden, hangisi tercih edilebilir? `test` gelenekseldir (ve genellikle sistem başlangıç komut dosyalarını çalıştırmak için kullanılan standart kabuklar için POSIX spesifikasyonunun bir parçasıdır), oysa `[[ ]]` `bash'a` (ve birkaç diğer modern kabuğa) özgüdür. `test` yaygın olarak kullanıldığı için nasıl kullanıldığını bilmek önemlidir, ancak `[[ ]]` açıkça daha kullanışlıdır ve kodlaması daha kolaydır, bu nedenle modern komut dosyaları için tercih edilir.

Taşınabilirlik, küçük zihinlerin hobgoblinidir

"Gerçek" Unix kullanıcılarıyla konuşursanız, çoğunun Linux'u pek sevmediğini hemen fark edersiniz. Onlar Linux'u saf olmayan ve kirli olarak görürler. Unix kullanıcılarının bir ilkesi, her şeyin "taşınabilir" olması gerektiğidir. Bu, yazdığınız herhangi bir komut dosyasının, herhangi bir Unix benzeri sistemde değiştirilmeden çalışabilmesi gerektiği anlamına gelir.

Unix kullanıcılarının buna inanmak için iyi nedenleri vardır. POSIX'ten önce komutlara ve kabuklara yapılan özel uzantıların Unix dünyasına ne yaptığını gördükten sonra, sevdikleri işletim sistemi üzerinde Linux'un etkisinden doğal olarak çekinirler.

Ancak taşınabilirliğin ciddi bir dezavantajı vardır. İlerlemeyi engeller. Her şeyin her zaman "en küçük ortak payda" teknikleri kullanılarak yapılmasını gerektirir. Kabuk programlama durumunda, bu, her şeyi orijinal Bourne kabuğu olan `sh` ile uyumlu hale getirmek anlamına gelir.

Bu dezavantaj, özel yazılım satıcılarının özel uzantılarını haklı çıkarmak için kullandıkları bahanedir, ancak onlar buna "yenilik" diyorlar. Ancak bunlar aslında müşterileri için birer kilitlenme aracıdır.

`Bash` gibi GNU araçlarında böyle bir kısıtlama yoktur. Standartları destekleyerek ve evrensel olarak kullanılabilir olmalarıyla taşınabilirliği teşvik ederler. `Bash` ve diğer GNU araçlarını, Windows dahil olmak üzere hemen hemen her tür sisteme ücretsiz olarak yükleyebilirsiniz. Bu nedenle `bash'ın` tüm özelliklerini rahatlıkla kullanabilirsiniz. *Gerçekten* taşınabilirdir.

Kontrol Operatörleri: Dallanmanın Başka Bir Yolu

`bash,` dallanma gerçekleştirebilen iki kontrol operatörü sağlar. `&&` (AND) ve `||` (OR) operatörleri, `[[ ]]` bileşik komutundaki mantıksal operatörler gibi çalışır. `&&` için sözdizimi şöyledir:

`komut1 && komut2`

İşte `||`'nin sözdizimi:

`komut1 || komut2`

Bunların davranışını anlamak önemlidir. `&&` operatörü ile, *command1* her zaman

her zaman yürütülür ve *command2*, *yalnızca command1* başarılı olursa yürütülür. `||` operatörü ile *command1* her zaman yürütülür ve *command2, yalnızca command1* başarısız *olursa* yürütülür.

Pratik olarak, bu şu şekilde bir şey yapabileceğimiz anlamına gelir:

```
[me@linuxbox ~]$ mkdir temp && cd temp
```

Bu, `temp` adlı bir dizin oluşturacak ve başarılı olursa, geçerli çalışma dizini `temp` olarak değiştirilecektir. İkinci komut, yalnızca `mkdir` komutu başarılı olursa denenir. Benzer şekilde,

```
[me@linuxbox ~]$ [[ -d temp ]] || mkdir temp
```

`temp` dizinini test eder ve test başarısız olursa dizin oluşturulur. Bu tür bir yapı, komut dosyalarında hataları işlemek için kullanışlıdır. Bu konuyu sonraki bölümlerde daha ayrıntılı olarak ele alacağız. Örneğin, bir komut dosyasında bunu yapabiliriz:

```
[ -d temp ] || exit 1
```

Komut dosyası `temp` dizinini gerektiriyorsa ve bu dizin mevcut değilse, komut dosyası 1 çıkış durumuyla sonlandırılır.

Karmaşık bir şey yapmak istediğimizde, bir komutun grup komutu olabileceğini unutmayın.

```
{ true && echo "true"; } && { false || echo "false"; }
```

Grup komutları, gruptaki son komutun çıkış durumunu döndürür.

Özet

Bu bölüme bir soru ile başladık. sys_info_page betiğimizin, kullanıcının tüm ana dizinleri okuma iznine sahip olup olmadığını nasıl algılamasını sağlayabiliriz? `if` ile ilgili bilgilerimizle, bu kodu report_home_space işlevine ekleyerek sorunu çözebiliriz:

```
report_home_space () {
    if [[ "$(id -u)" -eq 0 ]]; then
         cat << _EOF_
              <h2>Ev Alanı Kullanımı (Tüm Kullanıcılar)</h2>
              <pre>$(du -sh /home/*)</pre>
_EOF_
    else
         cat << _EOF_
              <h2>Ana Sayfa Alanı Kullanımı ($USER)</h2>
              <pre>$(du -sh $HOME)</pre>
_EOF_
    fi
    dönüş
}
```

id komutunun çıktısını değerlendiriyoruz. `-u` seçeneği ile `id`, etkin kullanıcının sayısal kullanıcı kimlik numarasını çıkarır. Süper kullanıcı her zaman kimlik numarası sıfırdır ve diğer tüm kullanıcılar sıfırdan büyük bir sayıdır. Bunu bilerek, biri süper kullanıcı ayrıcalıklarından yararlanan, diğeri ise kullanıcının kendi ana dizinine sınırlı olan iki farklı belge oluşturabiliriz.

sys_info_page programına bir süre ara vereceğiz, ama endişelenmeyin. Geri döneceğiz. Bu arada, çalışmamıza devam ettiğimizde ihtiyacımız olacak bazı konuları ele alacağız.

Daha Fazla Okuma

Bu bölümde ele alınan konularla ilgili daha ayrıntılı bilgi içeren `bash` man sayfasının birkaç bölümü vardır:

- Listeler (kontrol operatörleri `||` ve `&&'yi` kapsar)
- Bileşik Komutlar (`[[ ]]`, `(( ))` ve `if'i` kapsar)
- KOŞULLU İFADELER
- SHELL BUILTIN KOMUTLARI (`test'i` kapsar)

Ayrıca, Wikipedia'da sözde kod kavramı hakkında iyi bir makale

bulunmaktadır: https://en.wikipedia.org/wiki/Pseudocode

28 – Klavye Girişini Okuma

Şimdiye kadar yazdığımız komut dosyaları, çoğu bilgisayar programında bulunan bir özellikten yoksundur: *etkileşim*, yani programın kullanıcıyla etkileşime girme yeteneği. Birçok programın etkileşimli olması gerekmezken, bazı programlar kullanıcıdan doğrudan girdi alabilme özelliğinden yararlanır. Örneğin, önceki bölümdeki şu komut dosyasını ele alalım:

```
#!/bin/bash

# test-integer2: bir tamsayının değerini değerlendirir.

INT=-5

if [[ "$INT" =~ ^-?[0-9]+$ ]]; then
    if [ "$INT" -eq 0 ]; then
         echo "INT sıfırdır."
    else
         if [ "$INT" -lt 0 ]; then echo
              "INT negatif."
         else
              echo "INT pozitif."
         fi
         if [ $((INT % 2)) -eq 0 ]; then
              echo "INT çift sayıdır."
         else
              echo "INT tek sayıdır."
         fi
    fi
else
    echo "INT tamsayı değil." >&2 exit 1
fi
```

`INT` değerini değiştirmek istediğimizde, her seferinde komut dosyasını düzenlememiz gerekiyor. Komut dosyası kullanıcıdan bir değer istese çok daha kullanışlı olurdu. Bu bölümde, programlarımıza nasıl etkileşim ekleyebileceğimizi incelemeye başlayacağız.

programlarımıza etkileşim özelliği eklemeyi inceleyeceğiz.

`read` – Standart Girişten Değerleri Oku

`read` yerleşik komutu, standart girdiden tek bir satırı okumak için kullanılır. Bu komut, klavye girdisini okumak için veya yeniden yönlendirme kullanıldığında bir dosyadan bir satırlık veriyi okumak için kullanılabilir. Komutun sözdizimi aşağıdaki gibidir:

`read [-seçenekler] [değişken...]`

burada *seçenekler*, Tablo 28-1'de daha sonra listelenen kullanılabilir seçeneklerden bir veya daha fazlasıdır ve *değişken*, giriş değerini tutmak için kullanılan bir veya daha fazla değişkenin adıdır. Değişken adı belirtilmezse, kabuk değişkeni `REPLY` veri satırını içerir.

Temel olarak, `read` komutu standart girdiden belirli değişkenlere alanlar atar. Tamsayı değerlendirme betiğimizi `read komutunu` kullanacak şekilde değiştirirsek, şöyle görünebilir:

```
#!/bin/bash

# read-integer: bir tamsayının değerini değerlendirir.

echo -n "Lütfen bir tamsayı girin -> "
read int

if [[ "$int" =~ ^-?[0-9]+$ ]]; then if
    [ "$int" -eq 0 ]; then
         echo "$int sıfırdır."
    else
         if [ "$int" -lt 0 ]; then echo
              "$int negatif."
         else
              echo "$int pozitif."
         fi
         if [ $((int % 2)) -eq 0 ]; then echo
              "$int çift sayıdır."
         else
              echo "$int tek sayıdır."
         fi
    fi
else
    echo "Girilen değer tamsayı değil." >&2 exit 1
fi
```

`Echo` komutunu `-n` seçeneği ile (çıktıda son satır sonunu gizler) kullanarak bir komut istemi görüntüleriz ve ardından `read komutunu` kullanarak `int` değişkeni için bir değer gireriz. Bu komut dosyasını çalıştırdığımızda şu sonuç elde edilir:

```
[me@linuxbox ~]$ read-integer
Lütfen bir tamsayı girin -> 5
5 pozitif bir sayıdır.
5 tek sayıdır.
```

`read`, bu komut dosyasında gösterildiği gibi girişi birden fazla değişkene atayabilir:

```
#!/bin/bash

# read-multiple: klavyeden birden fazla değer oku echo -

n "Bir veya daha fazla değer girin > "
read var1 var2 var3 var4 var5

echo "var1 = '$var1'"
echo "var2 = '$var2'"
echo "var3 = '$var3'"
echo "var4 = '$var4'"
echo "var5 = '$var5'"
```

Bu komut dosyasında, beş adede kadar değer atıyor ve görüntülüyoruz. Farklı sayıda değer verildiğinde `read komutunun` nasıl davrandığına dikkat edin:

```
[me@linuxbox ~]$ read-multiple
Bir veya daha fazla değer girin > a b
c d e var1 = 'a'
var2 = 'b'
var3 = 'c'
var4 = 'd'
var5 = 'e'
[me@linuxbox ~]$ read-multiple
Bir veya daha fazla değer girin >
a var1 = 'a'
var2 = ''
var3 = ''
var4 = ''
```

```
var5 = ''
[me@linuxbox ~]$ read-multiple
Bir veya daha fazla değer girin > a b c d e f
g var1 = 'a'
var2 = 'b'
var3 = 'c'
var4 = 'd'
var5 = 'e f g'
```

`read komutu` beklenen sayıdan daha az değer alırsa, fazladan değişkenler boş kalır; fazla girdi ise son değişkenin tüm fazladan girdileri içermesine neden olur.

`read` komutundan sonra hiçbir değişken listelenmezse, tüm girişler bir kabuk değişkeni olan `REPLY`'ye atanır.

```
#!/bin/bash

# read-single: varsayılan değişkene birden fazla değer okur echo

-n "Bir veya daha fazla değer girin > "
read

echo "REPLY = '$REPLY'"
```

Bu komut dosyasını çalıştırdığınızda şu sonuç elde edilir:

```
[me@linuxbox ~]$ read-single
Bir veya daha fazla değer girin > a b
c d REPLY = 'a b c d'
```

Seçenekler

`read`, Tablo 28-1'de açıklanan seçenekleri destekler.

Tablo 28-1: `read` Seçenekleri

Seçenek	Açıklama
`-a` *`dizisi`*	Girişi, sıfır indeksinden başlayarak *diziye* atayın. Dizileri 35. bölümde ele alacağız.

`-d` *ayırıcı*	Dizgedeki *sınırlayıcının* ilk karakteri, satır sonu karakteri yerine girdinin sonunu belirtmek için kullanılır.
`-e`	Girişi işlemek için Readline kullanın. Bu, komut satırında olduğu gibi giriş düzenlemesine izin verir.
`-` *dize*	Kullanıcı sadece `ENTER` tuşuna basarsa `string'i varsayılan yanıt olarak kullanır.` `-e` seçeneği gereklidir.
`-n` *num*	Satırın tamamı yerine *num* karakterlik girişi okuyun.
`-p` *prompt*	Dize *prompt* kullanarak giriş için bir istem görüntüleyin.
`-r`	Ham mod. Ters eğik çizgi karakterlerini kaçış karakteri olarak yorumlamayın. **Güvenlik için bu seçeneği kullanmanız önerilir**. Örneğin, DOS yol adını girerken, ters eğik çizgilerin gerçek karakterler olarak değerlendirilmesini isteriz.
`-s`	Sessiz mod. Karakterler yazıldıkça ekrana yansıtmayın. Bu, şifreler ve diğer gizli bilgileri girerken kullanışlıdır.
`-` *saniye*	Zaman aşımı. *Saniye* sonra girişi sonlandırır. Giriş zaman aşımına uğradığında `read` sıfır olmayan bir çıkış durumu döndürür.
`-u` *fd*	Standart giriş yerine *fd* dosya tanımlayıcısından girişi kullan.

Çeşitli seçenekleri kullanarak, `read` ile ilginç şeyler yapabiliriz. Örneğin, `-p` seçeneği ile bir komut istemi dizesi sağlayabiliriz.

```
#!/bin/bash

# read-single: varsayılan değişkene birden fazla değer okur

read -r -p "Bir veya daha fazla değer girin > "

echo "REPLY = '$REPLY'"
```

`-t` ve `-s` seçenekleriyle, "gizli" girişi okuyan ve giriş belirli bir süre içinde tamamlanmazsa zaman aşımına uğrayan bir komut dosyası yazabiliriz.

```
#!/bin/bash
```

```
# read-secret: gizli bir parola girin

if read -r -t 10 -sp "Gizli parolayı girin > " secret_pass; then echo -e
    "\nGizli parola = '$secret_pass'"
else
    echo -e "\nGiriş süresi doldu" >&2
    exit 1
fi
```

Komut dosyası kullanıcıdan gizli bir parola girmesini ister ve giriş için 10 saniye bekler. Giriş belirtilen süre içinde tamamlanmazsa, komut dosyası bir hata ile sonlanır. `-s` seçeneği dahil edildiğinden, parola karakterleri yazıldıkça ekrana yansıtılmaz.

`-e` ve `-i` seçeneklerini birlikte kullanarak kullanıcıya varsayılan bir yanıt vermek mümkündür.

```
#!/bin/bash

# read-default: kullanıcı Enter tuşuna basarsa varsayılan bir değer

girin. read -e -p "Kullanıcı adınız nedir? " -i $USER
echo "Cevabınız: '$REPLY'"
```

Bu komut dosyasında, kullanıcıdan bir kullanıcı adı girmesini isteriz ve varsayılan bir değer sağlamak için `USER` ortam değişkenini kullanırız. Komut dosyası çalıştırıldığında, varsayılan dizeyi görüntüler ve kullanıcı sadece `ENTER tuşuna` basarsa, `read` varsayılan dizeyi `REPLY` değişkenine atar.

```
[me@linuxbox ~]$ read-default
Kullanıcı adınız nedir? me
Cevabınız: 'me'
```

IFS

Normalde, kabuk `read'e` sağlanan girişte kelime bölme işlemi gerçekleştirir. Gördüğümüz gibi, bu, bir veya daha fazla boşlukla ayrılmış birden fazla kelimenin giriş satırında ayrı öğeler haline geldiği ve `read` tarafından ayrı değişkenlere atandığı anlamına gelir. Bu davranış, `IFS` ("Internal Field Separator" için) adlı bir kabuk değişkeni tarafından yapılandırılır. Varsayılan

IFS değeri, öğeleri birbirinden ayıran bir boşluk, bir sekme ve bir satır sonu karakteri içerir.

IFS değerini ayarlayarak okunacak alanların ayrılmasını kontrol edebiliriz. Örneğin, /etc/passwd dosyası, alan ayırıcı olarak iki nokta üst üste karakterini kullanan veri satırları içerir. IFS değerini tek bir iki nokta üst üste karakterine değiştirerek, read komutunu kullanarak /etc/passwd dosyasının içeriğini okuyabilir ve alanları farklı değişkenlere başarıyla ayırabiliriz. İşte bunu yapan bir komut dosyası:

```
#!/bin/bash

# read-ifs: bir dosyadan alanları okur

FILE=/etc/passwd

read -r -p "Kullanıcı adı girin > " kullanıcı_adı

dosya_bilgisi="$(grep "^$kullanıcı_adı:" $FILE)"

if [ -n "$file_info" ]; then
    IFS=":" read -r kullanıcı pw uid gid isim ev kabuk <<< "$file_info"
    echo "Kullanıcı =   '$kullanıcı'"
    echo "UID =         '$uid'"
    echo "GID =         '$gid'"
    echo "Tam Ad = '$name'" echo
    "Ana Dizin = '$home'" echo
    "Kabuk =            '$shell'"
else
    echo "Böyle bir kullanıcı yok
    '$user_name'" >&2 exit 1
fi
```

Bu komut dosyası, kullanıcıdan sistemdeki bir hesabın kullanıcı adını girmesini ister ve ardından /etc/passwd dosyasında bulunan kullanıcı kaydındaki farklı alanları görüntüler. Komut dosyası iki ilginç satır içerir. İlki şöyledir:

```
file_info=$(grep "^$user_name:" $FILE)
```

Bu satır, grep komutunun sonuçlarını file_info değişkenine atar. Grep tarafından kullanılan düzenli ifade, kullanıcı adının /etc/passwd dosyasındaki tek bir satırla eşleşmesini sağlar.

İkinci ilginç satır şudur:

```
IFS=":" read user pw uid gid name home shell <<< "$file_info"
```

Bu satır üç bölümden oluşur: bir değişken ataması, argüman olarak değişken adlarının listesini içeren bir `read` komutu ve garip yeni bir yönlendirme operatörü. Önce değişken atamasına bakalım.

Kabuk, bir komuttan hemen önce bir veya daha fazla değişken atamasının yapılmasına izin verir. Bu atamalar, takip eden komutun ortamını değiştirir. Atamanın etkisi, komutun süresi boyunca ortamı geçici olarak değiştirmektir. Bizim durumumuzda, `IFS'nin` değeri iki nokta üst üste karakterine değiştirilir. Alternatif olarak, bunu şu şekilde kodlayabilirdik:

```
OLD_IFS="$IFS"
IFS=":"
read user pw uid gid name home shell <<< "$file_info"
IFS="$OLD_IFS"
```

Burada `IFS` değerini saklıyor, yeni bir değer atıyor, `read` komutunu çalıştırıyor ve ardından `IFS'yi` orijinal değerine geri yüklüyoruz. Açıkçası, değişken atamasını komutun önüne yerleştirmek aynı şeyi yapmak için daha kısa bir yoldur.

`<<<` operatörü bir *here string'i* gösterir. Here string, here document'a benzer, ancak daha kısadır ve tek bir string'den oluşur. Örneğimizde, `/etc/` passwd dosyasındaki veri satırı read komutunun standart girişine beslenir. Neden bu oldukça dolaylı yöntem yerine şunu seçmedik diye merak edebiliriz:

```
echo "$file_info" | IFS=":" read kullanıcı pw uid gid isim ev kabuk
```

Bunun bir nedeni var...

Pipe `read` kullanamazsınız

`read` komutu normalde standart girişten girdi alır, ancak bunu yapamazsınız:

```
echo "foo" | read
```

Bunun çalışmasını bekleriz, ancak çalışmaz. Komut başarılı gibi görünecek, ancak `REPLY` değişkeni her zaman boş olacaktır. Neden böyle olur?

Bunun açıklaması, kabuğun boru hatlarını işleme biçimiyle ilgilidir. `Bash'ta` (ve `sh` gibi diğer kabuklarda), boru hatları *alt kabuklar* oluşturur. Bunlar, boru hattındaki komutu yürütmek için kullanılan kabuğun ve ortamının kopyalarıdır. Önceki örneğimizde, `read` bir alt kabukta yürütülür.

Unix benzeri sistemlerdeki alt kabuklar, işlemlerin yürütülmesi sırasında kullanmak üzere ortamın kopyalarını oluşturur. İşlemler bittiğinde, ortamın kopyası

Bu, *bir alt kabuğun asla ana işleminin ortamını değiştiremeyeceği* anlamına gelir. `read` değişkenleri atar ve bunlar ortamın bir parçası olur. Önceki örnekte, `read` alt kabuğun ortamındaki `REPLY` değişkenine `foo` değerini atar, ancak komut sonlandığında alt kabuk ve ortamı yok olur ve atamanın etkisi kaybolur.

Burada dizileri kullanmak, bu davranışı aşmanın bir yoludur. Başka bir yöntem ise Bölüm 36'da ele alınmaktadır.

Girişleri Doğrulama

Klavye girişi yapabilme özelliğimizle birlikte, girişi doğrulama gibi ek bir programlama zorluğu da ortaya çıkmıştır. Genellikle, iyi yazılmış bir program ile kötü yazılmış bir program arasındaki fark, programın beklenmedik durumlarla başa çıkma yeteneğinde yatmaktadır. Beklenmedik durumlar genellikle hatalı giriş şeklinde ortaya çıkar. Önceki bölümdeki değerlendirme programlarımızda, tamsayıların değerlerini kontrol edip boş değerleri ve sayısal olmayan karakterleri eleyerek bu konuda biraz çalışma yaptık. Geçersiz verilere karşı korunmak için, program her girdi aldığında bu tür programlama kontrollerini gerçekleştirmek önemlidir. Bu, özellikle birden fazla kullanıcı tarafından paylaşılan programlar için önemlidir. Bir programın yalnızca yazar tarafından özel bir görevi gerçekleştirmek için bir kez kullanılacak olması durumunda, ekonomik nedenlerle bu güvenlik önlemlerinin atlanması mazur görülebilir. Ancak bu durumda bile, program dosya silme gibi tehlikeli görevler gerçekleştiriyorsa, her ihtimale karşı veri doğrulama eklemek akıllıca olacaktır.

İşte çeşitli türdeki girdileri doğrulayan bir örnek program:

```
#!/bin/bash

# read-validate: girişi doğrular

invalid_input () {
    echo "Geçersiz giriş '$REPLY'" >&2
    exit 1
}

read -r -p "Tek bir öğe girin > " #

giriş boş (geçersiz)
[[ -z "$REPLY" ]] && invalid_input
```

```
# giriş birden fazla öğe (geçersiz)
(( "$(echo "$REPLY" | wc -w)" > 1 )) && geçersiz_giriş

# girdi geçerli bir dosya adı mı?
if [[ "$REPLY" =~ ^[-[:alnum:]\._]+$ ]]; then echo
    "'$REPLY' geçerli bir dosya adıdır."
    if [[ -e "$REPLY" ]]; then
         echo "Ve '$REPLY' dosyası mevcut."
    else
         echo "Ancak, '$REPLY' dosyası mevcut değil."
    fi

    # giriş bir kayan nokta sayısı mı?
    if [[ "$REPLY" =~ ^-?[[:digit:]]*\.[[:digit:]]+$ ]]; then echo
         "'$REPLY' bir kayan nokta sayıdır."
    else
         echo "'$REPLY' kayan nokta sayısı değil."
    fi

    # girdi bir tamsayı mı?
    if [[ "$REPLY" =~ ^-?[[:digit:]]+$ ]]; then
         echo "'$REPLY' bir tamsayıdır."
    else
         echo "'$REPLY' bir tamsayı değildir."
    fi
else
    echo " '$REPLY' dizesi geçerli bir dosya adı değil."
fi
```

Bu komut dosyası, kullanıcıdan bir öğe girmesini ister. Öğe daha sonra içeriğini belirlemek için analiz edilir. Gördüğümüz gibi, komut dosyası şimdiye kadar ele aldığımız birçok kavramı kullanır; bunlar arasında kabuk işlevleri, `[[ ]]`, `(( ))`, kontrol operatörü `&&` ve `if` ile birlikte düzenli ifadeler de bulunur.

Menüler

Yaygın bir etkileşim türü, *menü odaklı* etkileşimdir. Menü odaklı programlarda, kullanıcıya bir seçenek listesi sunulur ve bunlardan birini seçmesi istenir. Örneğin, aşağıdakileri sunan bir program düşünebiliriz:

```
Lütfen Seçin:
```

```
1. Sistem Bilgilerini Göster
2. Disk Alanını Görüntüle
3. Ev Alanı Kullanımını Görüntüle
0. Çık

Seçim girin [0-3] >
```

sys_info_page programımızı yazarken öğrendiklerimizi kullanarak, önceki menüdeki görevleri gerçekleştirmek için menü tabanlı bir program oluşturabiliriz.

```
#!/bin/bash

# read-menu: menü tabanlı bir sistem bilgisi programı

clear
echo "
Lütfen Seçin:

1. Sistem Bilgilerini Görüntüle
2. Disk Alanını Görüntüle
3. Ana Sayfa Alanı Kullanımını Görüntüle
0. Çıkış
"
read -r -p "Seçimi girin [0-3] > "

if [[ "$REPLY" =~ ^[0-3]$ ]]; then if
    [[ "$REPLY" == 0 ]]; then
         echo "Program sonlandırıldı."
         exit
    fi
    if [[ "$REPLY" == 1 ]]; then echo
         "Ana bilgisayar adı:
         $HOSTNAME" uptime
         çık
    fi
    if [[ "$REPLY" == 2 ]]; then
         df -h
         çık
    fi
    eğer [[ "$REPLY" == 3 ]]; o zaman
         if [[ "$(id -u)" -eq 0 ]]; then
              echo "Ev Alanı Kullanımı (Tüm Kullanıcılar)"
```

```
                du -sh /home/*
            else
                echo "Ana Sayfa Alanı Kullanımı ($USER)"
                du -sh "$HOME"
            fi
            exit
        fi
    else
        echo "Geçersiz giriş." >&2
        exit 1
    fi
```

Bu komut dosyası mantıksal olarak iki bölüme ayrılmıştır. İlk bölüm menüyü görüntüler ve kullanıcının yanıtını girer. İkinci bölüm yanıtı tanımlar ve seçilen eylemi gerçekleştirir. Bu betikte exit komutunun kullanımına dikkat edin. Burada, bir eylem gerçekleştirildikten sonra betiğin gereksiz kodları çalıştırmasını önlemek için kullanılır. Bir programda birden fazla çıkış noktasının bulunması genellikle iyi bir fikir değildir (program mantığını anlamayı zorlaştırır), ancak bu betikte işe yarar.

Özet

Bu bölümde, etkileşimli programlara doğru ilk adımlarımızı attık ve kullanıcıların klavye aracılığıyla programlarımıza veri girmelerine olanak sağladık. Şimdiye kadar sunduğumuz teknikleri kullanarak, özel hesaplama programları ve karmaşık komut satırı araçları için kullanımı kolay ön uçlar gibi birçok yararlı program yazmak mümkündür. Bir sonraki bölümde, menü odaklı program kavramını daha da iyileştirmek için bu kavramı temel alacağız.

Ekstra Puan

Bu bölümdeki programları dikkatlice incelemek ve mantıksal yapılarını tam olarak anlamak önemlidir, çünkü ilerleyen bölümlerde programlar giderek daha karmaşık hale gelecektir. Alıştırma olarak, bu bölümdeki programları `[[ ]]` bileşik komutu yerine `test` komutunu kullanarak yeniden yazın. İpucu: Düzenli ifadeleri değerlendirmek ve çıkış durumunu değerlendirmek için `grep komutunu` kullanın. Bu iyi bir alıştırma olacaktır.

Daha Fazla Okuma

- *Bash Referans Kılavuzu*, `read` komutunu da içeren yerleşik komutlar hakkında bir bölüm içerir: https://www.gnu.org/software/bash/manual/bashref.html#Bash-Builtins

29 – Akış Kontrolü: `while / until`

Önceki bölümde, çeşitli sistem bilgilerini üretmek için menü tabanlı bir program geliştirdik. Program çalışıyor, ancak hala önemli bir kullanılabilirlik sorunu var. Yalnızca tek bir seçeneği yürütüyor ve sonra sonlanıyor. Daha da kötüsü, geçersiz bir seçim yapıldığında, programa yeniden deneme fırsatı vermeden hata ile sonlanıyor. Kullanıcı programı sonlandırmayı seçene kadar menü ekranını ve seçimi tekrar tekrar tekrarlayabilecek şekilde programı bir şekilde oluşturabilirsek daha iyi olur.

Bu bölümde, programların bazı bölümlerini tekrarlamak için kullanılabilen *döngü* adlı bir programlama kavramını inceleyeceğiz. Kabuk, döngü için üç bileşik komut sağlar. Bu bölümde bunlardan ikisini, üçüncü komutu ise sonraki bölümlerde inceleyeceğiz.

Döngü

Günlük yaşam, tekrarlanan faaliyetlerle doludur. Her gün işe gitmek, köpeği gezdirmek ve havuç dilimlemek, bir dizi adımı tekrarlamayı içeren görevlerdir. Havuç dilimlemeyi ele alalım. Bu faaliyeti sözde kodla ifade edersek, şöyle görünebilir:

1. kesme tahtası al
2. bıçak al
3. havucu kesme tahtasına koy
4. bıçağı kaldır
5. havucu ilerlet
6. havucu dilimleyin
7. havuç tamamen dilimlendiyse, işlemi sonlandırın; aksi takdirde 4. adıma geçin

4. ve 7. adımlar bir *döngü* oluşturur. Döngü içindeki eylemler, "havuç tamamen dilimlendi" koşulu sağlanana kadar tekrarlanır.

while

bash da benzer bir fikri ifade edebilir. Diyelim ki 1'den 5'e kadar sıralı olarak beş sayı görüntülemek istiyoruz. Bir bash betiği şu şekilde oluşturulabilir:

```
#!/bin/bash

# while-count: bir dizi sayı göster count=1

while [[ "$count" -le 5 ]]; do
    echo "$count" count=$((count
    + 1))
done
echo "Bitti."
```

Bu komut dosyası çalıştırıldığında aşağıdakileri görüntüler:

```
[me@linuxbox ~]$ while-count 1
2
3
4
5
Bitti.
```

while komutunun sözdizimi aşağıdaki gibidir:

while *komutlar*; do *komutlar*; done

if gibi, while da bir komut listesinin çıkış durumunu değerlendirir. Çıkış durumu sıfır olduğu sürece, döngü içindeki komutları gerçekleştirir. Önceki betikte, count değişkeni oluşturulur ve başlangıç değeri 1 olarak atanır. while komutu, [[]] bileşik komutunun çıkış durumunu değerlendirir. [[]] komutu sıfır çıkış durumu döndürdüğü sürece, döngü içindeki komutlar yürütülür. Her döngünün sonunda [[]] komutu tekrarlanır. Döngünün beş kez yinelenmesinden sonra count değişkeninin değeri 6'ya yükselir, [[]] komutu artık sıfır çıkış durumu döndürmez ve döngü sonlanır. Program, döngünün ardından gelen bir sonraki deyimle devam eder.

Bir önceki bölümdeki menü okuma programını iyileştirmek için *while döngüsü* kullanabiliriz.

```
#!/bin/bash

# while-menu: menü tabanlı bir sistem bilgisi programı

DELAY=3 # Sonuçları görüntülemek için saniye sayısı

while [[ "$REPLY" != 0 ]]; do
    clear
    cat << _EOF_
         Lütfen seçin:

         1. Sistem Bilgilerini Görüntüle
         2. Disk Alanını Görüntüle
         3. Ana Sayfa Alanı Kullanımını Görüntüle
         0. Çıkış

_EOF_
    read -r -p "Seçimi girin [0-3] > "

    if [[ "$REPLY" =~ ^[0-3]$ ]]; then if
         [[ $REPLY == 1 ]]; then
              echo "Ana bilgisayar adı:
              $HOSTNAME" çalışma süresi
              sleep "$DELAY"
         fi
         if [[ "$REPLY" == 2 ]]; then
              df -h
              sleep "$DELAY"
         fi
         if [[ "$REPLY" == 3 ]]; then
              if [[ "$(id -u)" -eq 0 ]]; then
                   echo "Ev Alanı Kullanımı (Tüm Kullanıcılar)" du
                   -sh /home/*
              else
                   echo "Ana Sayfa Alanı Kullanımı ($USER)"
                   du -sh "$HOME"
              fi
              sleep "$DELAY"
         fi
    else
         echo "Geçersiz giriş."
         sleep "$DELAY"
    fi
done
```

```
echo "Program sonlandırıldı."
```

Menüyü bir while döngüsü içine alarak, programın her seçimden sonra menü ekranını tekrarlamasını sağlayabiliriz. `REPLY 0'a` eşit olmadığı sürece döngü devam eder ve menü tekrar görüntülenir, böylece kullanıcı başka bir seçim yapma fırsatı bulur. Her eylemin sonunda, bir `sleep` komutu yürütülür, böylece program birkaç saniye duraklayarak, ekran silinip menü yeniden görüntülenmeden önce seçimin sonuçlarının görülmesini sağlar. `REPLY 0'a` eşit olduğunda, yani "çıkış" seçeneği seçildiğinde, döngü sonlanır ve yürütme `done` satırıyla devam eder.

`break` ve `continue`

`bash`, döngüler içindeki program akışını kontrol etmek için kullanılabilecek iki yerleşik komut sağlar. `break` komutu bir döngüyü hemen sonlandırır ve program kontrolü döngünün ardından gelen bir sonraki deyimle devam eder. `continue` komutu, döngünün geri kalanının atlanmasına neden olur ve program kontrolü döngünün bir sonraki yinelemesiyle devam eder. Burada, hem `break` hem de `continue` komutlarını içeren `while-menu` programının bir sürümünü görüyoruz:

```
#!/bin/bash

# while-menu2: menü tabanlı bir sistem bilgisi programı

DELAY=3 # Sonuçları görüntülemek için saniye sayısı

while true; do
    clear
    cat << _EOF_
         Lütfen Seçin:

         1. Sistem Bilgilerini Görüntüle
         2. Disk Alanını Görüntüle
         3. Ev Alanı Kullanımını Görüntüle
         0. Çıkış

_EOF_
    read -p "Seçimi girin [0-3] > "

    if [[ "$REPLY" =~ ^[0-3]$ ]]; then if
         [[ "$REPLY" == 1 ]]; then
              echo "Ana bilgisayar adı: $HOSTNAME"
```

```
            uptime
            sleep "$DELAY"
            devam
        fi
        if [[ "$REPLY" == 2 ]]; then
            df -h
            sleep "$DELAY"
            devam
        fi
        if [[ "$REPLY" == 3 ]]; then
            if [[ "$(id -u)" -eq 0 ]]; then
                echo "Ev Alanı Kullanımı (Tüm Kullanıcılar)" du
                -sh /home/*
            else
                echo "Ana Sayfa Alanı Kullanımı ($USER)"
                du -sh "$HOME"
            fi
            sleep "$DELAY"
            devam
        fi
        if [[ "$REPLY" == 0 ]]; then
            break
        fi
    else
        geçersiz giriş. uyku
        "$DELAY"
    fi
done
echo "Program sonlandırıldı."
```

Bu komut dosyası sürümünde, `while komutuna` bir çıkış durumu sağlamak için `true` komutunu kullanarak *sonsuz* bir *döngü* (kendi kendine asla sonlanmayan) oluşturduk. `True` komutu her zaman sıfır çıkış durumu ile sonlanacağından, döngü asla sonlanmayacaktır. Bu, şaşırtıcı derecede yaygın bir komut dosyası tekniğidir. Döngü kendi kendine asla sonlanmayacağından, doğru zamanda döngüden çıkmak için bir yol sağlamak programcının sorumluluğundadır. Bu komut dosyasında, `0` seçeneği seçildiğinde döngüden çıkmak için `break` komutu kullanılır. Daha verimli bir yürütme sağlamak için, diğer komut dosyası seçeneklerinin sonuna `continue` komutu eklenmiştir. `Continue komutu` kullanılarak, bir seçim belirlendiğinde komut dosyası gerekli olmayan kodu atlar. Örneğin, `1` seçeneği seçilip belirlendiğinde, diğer seçenekleri test etmek için bir neden yoktur.

select

Bu, döngüsel menüler oluşturmak için kullanılan `select` shell builtin'den bahsetmek için iyi bir fırsat olacaktır. Bu komutun sözdizimi aşağıdaki gibidir; burada var bir değişken, *string* ise menü seçeneğinin metnidir:

`select` *`var`* `in [`*`string`*`.. ;] do` *`commands`*`; done`

`select` komutu çalıştırıldığında, *string*(ler)i ve ardından `PS3` (prompt string 3) değişkeninin içeriğini kullanıcının girişi için bir istem olarak görüntüler. Bir seçim yapıldığında, `REPLY` değişkenini kullanıcının girdisiyle ayarlar (`read` ile olduğu gibi) ve *var* değişkenindeki seçimle ilişkili string'i döndürür. Değerler ayarlandıktan sonra komutlar yürütülür ve başka bir seçim için istem tekrar görüntülenir. Bu biraz kafa karıştırıcı gelebilir, ancak bunu şu küçük komut dosyasıyla gösterebiliriz:

```
#!/bin/bash

# select-demo: select yerleşik demo

PS3="
Seçiminiz: "

select my_choice in First Second Third Fourth Quit; do echo
    "REPLY= $REPLY my_choice= $my_choice"
    [[ "$my_choice" == "Quit" ]] && break
done
```

İlk olarak, PS3 değişkeninin içeriğini istediğimiz komut istemi dizesiyle ayarlıyoruz. Ardından, `select komutunu` çalıştırıyoruz. Bu örnekte beş dize var ve dizeler olarak tek kelimeler kullanmış olsak da, herhangi bir tür tırnak içine alınmış metin kullanabiliriz. *Komutlarımız* için, `select` tarafından yapılan atamaları basitçe yankılıyoruz. Ayrıca, kullanıcının Quit seçeneğini seçip seçmediğini görmek için `my_choice` değişkenimizin içeriğini test ediyoruz ve seçtiyse, döngüden çıkmak için bir `break komutu` uyguluyoruz.

```
[me@linuxbox ~]$ select-demo
1) İlk
2) İkinci
3) Üçüncü
4) Dördüncü
5) Çık
```

```
Seçiminiz:
```

`select` ilk kez çalıştırıldığında, her bir dizeyi bir sayı ve ardından komut istemi dizesiyle birlikte görüntüler. Kullanıcı daha sonra istediği seçeneği temsil eden sayıyı girer. `select` komutu daha sonra `REPLY` değişkenini kullanıcının girdiği değeri ve varsa karşılık gelen dizeyi içerecek şekilde ayarlar.

```
Seçiminiz: 1
REPLY= 1 my_choice= İlk

Seçiminiz: 2
REPLY= 2 my_choice= İkinci
```

Burada kullanıcının 1 girdiğini görüyoruz ve `echo` komutu `RE-PLY` ve my_choice değişkenlerinin değerlerini görüntülüyor. `select`, kullanıcı "5" girene kadar komut istemini tekrar tekrar görüntüler. Kullanıcı geçersiz bir değer girerse, `select my_choice'i` boş bir dizeye ayarlar. Kullanıcı sadece `ENTER tuşuna` basarsa, `select` baştan başlar ve menü seçeneklerinin listesini yeniden görüntüler.

```
Seçiminiz: 6
REPLY= 6 my_choice=

Seçiminiz: abc REPLY=
abc my_choice=
```

`Seçim` döngüsü, bir `break` komutu ile karşılaşılana kadar veya kullanıcı dosya sonunu belirtmek için `Ctrl-d` tuşlarına basana kadar sonsuza kadar devam eder.

```
Seçiminiz: 5
REPLY= 5 my_choice= Çıkış
[me@linuxbox ~]$
```

`select` komutunun ilginç bir özelliği, menü seçeneklerini veya komut istemini standart çıktıya görüntülememesi, bunun yerine standart hatayı kullanmasıdır. Bu, döngü içindeki komutların yaptığı gerçek işin yeniden yönlendirilmesine olanak tanıdığı için aslında kullanışlıdır. İşte bir örnek:

```
[me@linuxbox ~]$ select-demo > choices.txt
```

Bu yeniden yönlendirmeyi yaptığımızda, menü ve komut istemi hala görüntülenir, ancak `echo` komutunun çıktısı yeniden yönlendirilir.

Sistem bilgisi betiğimizin önceki
`while` döngüsünü `select` ile değiştirelim.

```
#!/bin/bash

# select-menu: menü tabanlı bir sistem bilgisi programı

DELAY=3 # Sonuçları görüntülemek için saniye sayısı
PS3="
Seçimi girin [1-4] > "

select str in \
    "Sistem Bilgilerini Göster" \
    "Disk Alanını Göster" \
    "Ev Alanı Kullanımını Göster" \ "Çık"; yap
    if [[ "$REPLY" == "1" ]]; then
        echo "Ana Bilgisayar Adı:
        $HOSTNAME" çalışma süresi
        sleep "$DELAY" devam
    fi
    if [[ "$REPLY" == "2" ]]; then
        df -h
        sleep "$DELAY" devam
    fi
    eğer [[ "$REPLY" == "3" ]]; o zaman
        if [[ $(id -u) -eq 0 ]]; then
            echo "Ev Alanı Kullanımı (Tüm Kullanıcılar)"
            du -sh /home/* 2> /dev/null
        else
            echo "Ana Sayfa Alanı Kullanımı ($USER)"
            du -sh "$HOME" 2> /dev/null
        fi
        sleep "$DELAY" devam
    fi
```

```
    if [[ "$REPLY" == "4" ]]; then
        break
    fi
    if [[ -z "$str" ]]; then
        echo "Geçersiz giriş."
        sleep "$DELAY"
    fi
done
echo "Program sonlandırıldı."
```

Alternatif betiğimizde, `PS3` değişkenini ayarladıktan sonra dört dize ile `select komutunu` çağırıyoruz. Daha sonra `select` tarafından ayarlanan `str` değişkenini test edebiliriz, ancak `REPLY` değişkenini test edip ona göre hareket etmek daha kolaydır. Döngünün sonunda, `str` değişkeninin geçersiz bir değeri gösteren sıfır uzunluğuna sahip olup olmadığını kontrol ediyoruz.

Peki, menü oluşturmak için hangi yöntemi kullanmalıyız? `select` komutu ilginçtir, ancak menü görüntüleme için standart hatayı kullanmasının yanı sıra, kodlama çabasını pek azaltmaz ve menü görüntülemesinin görsel tasarımını büyük ölçüde sınırlar.

until

`until` komutu `while komutuna` çok benzer, ancak sıfır olmayan bir çıkış durumu ile karşılaşıldığında döngüden çıkmak yerine tam tersini yapar. Bir *until döngüsü*, sıfır çıkış durumu alana kadar devam eder. while-count betiğimizde, count değişkeninin değeri 5'ten küçük veya 5'e eşit olduğu sürece döngüyü sürdürdük. Betiği `until` ile kodlayarak da aynı sonucu elde edebiliriz.

```
#!/bin/bash

# until-count: bir dizi sayı görüntüler count=1

[[ "$count" -gt 5 ]]; do echo
    "$count" count=$((count +
    1))
done
echo "Tamamlandı."
```

Test ifadesini `$count -gt 5` olarak değiştirerek, `until` döngüyü doğru zamanda sonlandıracaktır. `While` veya `until` döngüsünün hangisinin kullanılacağına karar vermek genellikle

en net testi yazmaya izin veren döngüyü seçme meselesidir.

Döngülerle Dosya Okuma

`while` ve `until` standart girişi işleyebilir. Bu, dosyaların while ve until döngüleri ile işlenmesine olanak tanır. Aşağıdaki örnekte, önceki bölümlerde kullanılan dis-tros.txt dosyasının içeriğini göstereceğiz:

```
#!/bin/bash

# while-read: bir dosyadan satırları okur

while read -r distro version release; do
    printf "Dağıtım: %s\tSürüm: %s\tYayınlanma Tarihi:
         %s\n" \ "$distro" \
         "$version" \
         "$release"
done < distros.txt
```

Bir dosyayı döngüye yönlendirmek için, yönlendirme operatörünü `done` ifadesinin arkasına yerleştiririz. Döngü, yönlendirilen dosyadan alanları girmek için `read` komutunu kullanır. `read` komutu, her satır okunduktan sonra, dosya sonuna ulaşılana kadar sıfır çıkış durumuyla sonlanır. Bu noktada, sıfır olmayan bir çıkış durumuyla sonlanır ve böylece döngü sonlandırılır. Standart girişi bir döngüye aktarmak da mümkündür.

```
#!/bin/bash

# while-read2: bir dosyadan satırları oku

sort -k 1,1 -k 2n distros.txt | while read -r distro version release; do
    printf "Dağıtım: %s\tSürüm: %s\tYayınlanma Tarihi:
         %s\n" \ "$distro" \
         "$version" \
         "$release"
done
```

Burada sort komutunun çıktısını alıp metin akışını görüntülüyoruz. Ancak, bir boru alt kabukta döngüyü çalıştıracağından, döngü içinde oluşturulan veya atanan tüm değişkenlerin döngü sona erdiğinde kaybolacağını unutmamak önemlidir.

Özet

Döngülerin tanıtımı ve dallanma, alt yordamlar ve dizilerle ilgili önceki bilgilerimizle, programlarda kullanılan başlıca akış kontrol türlerini ele aldık. bash'ın başka püf noktaları da vardır, ancak bunlar temel kavramların daha ileri düzeydeki uygulamalarıdır.

Daha Fazla Okuma

- Linux Dokümantasyon Projesi'nin *Yeni Başlayanlar İçin Bash Kılavuzu'nda* while döngülerine ilişkin birkaç örnek daha bulunmaktadır: https://tldp.org/LDP/Bash-Beginners-Guide/html/sect_09_02.html
- Wikipedia'da, akış kontrolü ile ilgili daha kapsamlı bir makalenin parçası olan döngülerle ilgili bir makale bulunmaktadır: https://en.wikipedia.org/wiki/Control_flow#Loops

30 – Sorun Giderme

Artık komut dosyalarımız daha karmaşık hale geldiğine göre, bir şeyler ters gittiğinde neler olacağına bakmanın zamanı geldi. Bu bölümde, komut dosyalarında sıkça karşılaşılan bazı hata türlerini inceleyecek ve sorunları tespit edip ortadan kaldırmak için kullanılabilecek birkaç yararlı tekniği ele alacağız.

Sözdizimsel Hatalar

Genel bir hata türü, *sözdizimsel* hatalardır. Sözdizimsel hatalar, kabuk sözdiziminin bazı öğelerinin yanlış yazılmasından kaynaklanır. Kabuk, bu tür bir hatayla karşılaştığında komut dosyasının yürütülmesini durdurur.

Aşağıdaki açıklamalarda, yaygın hata türlerini göstermek için bu komut dosyasını kullanacağız:

```
#!/bin/bash

# sorun: yaygın hataları gösteren komut dosyası number=1

if [ $number = 1 ]; then
    echo "Sayı 1'e eşittir." else
    echo "Sayı 1'e eşit değil."
fi
```

Yazıldığı gibi, bu komut dosyası başarıyla çalışır.

```
[me@linuxbox ~]$ trouble
Sayı 1'e eşittir.
```

Eksik Tırnak İşaretleri

Komut dosyasını düzenleyelim ve ilk echo komutunu izleyen argümandan sondaki tırnağı kaldıralım. `echo` komutunu takip eden argümandan sondaki tırnak işaretini kaldıralım.

```
#!/bin/bash

# trouble: yaygın hataları gösteren komut dosyası

number=1

if [ $number = 1 ]; then
    echo "Sayı 1'e eşittir.
else
    echo "Sayı 1'e eşit değil."
fi
```

İşte olanlar:

```
[me@linuxbox ~]$ trouble
/home/me/bin/trouble: satır 10: eşleşen `"' aranırken beklenmedik
EOF
/home/me/bin/trouble: satır 13: sözdizimi hatası: beklenmedik dosya sonu
```

İki hata oluşturur. İlginçtir ki, hata mesajlarında bildirilen satır numaraları, eksik tırnak işaretinin kaldırıldığı yer değil, programın çok daha ilerleyen kısımlarıdır. Eksik tırnak işaretinden sonra programı takip edersek, bunun nedenini görebiliriz. `bash`, bir tane bulana kadar kapanış tırnak işaretini aramaya devam eder ve bunu ikinci `echo` komutunun hemen ardından bulur. Bundan sonra `bash` çok karışır. Sonraki `if` komutunun sözdizimi bozulur, çünkü `fi` ifadesi artık tırnak içine alınmış (ama açık) bir dize içindedir.

Uzun komut dosyalarında bu tür hataları bulmak oldukça zor olabilir. Sözdizimi vurgulamalı bir düzenleyici kullanmak yardımcı olacaktır, çünkü çoğu durumda tırnak içine alınmış dizeleri diğer kabuk sözdizimi türlerinden farklı bir şekilde görüntüler. Tam sürüm `vim` yüklüyse, şu komutu girerek sözdizimi vurgulamayı etkinleştirebilirsiniz:

```
:syntax on
```

Eksik veya Beklenmeyen Belirteçler

Bir başka yaygın hata ise `if` veya `while` gibi bileşik komutları tamamlamayı unutmaktır. if komutundaki `test` ifadesinden sonra noktalı virgülü kaldırırsak ne olacağına bir bakalım:

```
#!/bin/bash

# sorun: yaygın hataları gösteren komut dosyası number=1

if [ $number = 1 ] then
    echo "Sayı 1'e eşittir." else
    echo "Sayı 1'e eşit değil."
fi
```

Sonuç şöyledir:

```
[me@linuxbox ~]$ sorun
/home/me/bin/trouble: satır 9: beklenmedik token yakınında sözdizimi hatası
`else'
/home/me/bin/trouble: satır 9: `else'
```

Yine, hata mesajı asıl sorundan sonra ortaya çıkan bir hatayı işaret ediyor. Olanlar gerçekten çok ilginç. Hatırlayacağımız gibi, `if` bir komut listesi alır ve listedeki son komutun çıkış kodunu değerlendirir. Programımızda, bu listenin tek bir komuttan oluşmasını istiyoruz, `[`, `test'in` eşanlamlısı. `[` komutu, kendisini izleyenleri bir argüman listesi olarak alır; bizim durumumuzda, bu dört argüman: `$number`, 1, `=` ve `]`. Noktalı virgül kaldırıldığında, `then` kelimesi argümanlar listesine eklenir ve bu, sözdizimsel olarak geçerlidir. Aşağıdaki `echo` komutu da geçerlidir. Bu komut, `if'in` çıkış kodu için değerlendireceği komutlar listesindeki başka bir komut olarak yorumlanır. Ardından `else` ile karşılaşılır, ancak kabuk bunu bir komut adı olarak değil, *ayrılmış bir kelime* (kabuk için özel bir anlamı olan kelime) olarak tanıdığı için bu kelime yanlış yerde bulunur ve bu da hata mesajının nedenidir.

Beklenmedik Genişlemeler

Bir komut dosyasında sadece aralıklı olarak ortaya çıkan hatalar olabilir. Bazen komut dosyası sorunsuz çalışırken, bazen de genişletme sonuçları nedeniyle hata verebilir. Kayıp noktalı virgülümüzü yeniden

eksik noktalı virgülü geri koyup `sayının` değerini burada gösterildiği gibi boş bir değişkene değiştirirsek, bunu gösterebiliriz:

```
#!/bin/bash

# sorun: yaygın hataları gösteren komut dosyası

number=

if [ $number = 1 ]; then
    echo "Sayı 1'e eşittir." else
    echo "Sayı 1'e eşit değil."
fi
```

Bu değişiklikle komut dosyasını çalıştırdığınızda aşağıdaki çıktı elde edilir:

```
[me@linuxbox ~]$ trouble
/home/me/bin/trouble: satır 7: [: =: tekli operatör bekleniyor Sayı
 1'e eşit değil.
```

Bu oldukça şifreli hata mesajını ve ardından ikinci `echo` komutunun çıktısını alırız. Sorun, `test` komutu içindeki `sayı` değişkeninin genişletilmesidir. Komut:

```
[ $number = 1 ]
```

`sayı` boşken genişleme işlemine tabi tutulursa, sonuç

```
[ = 1 ]
```

olur ve bu geçersiz olduğundan hata oluşur. `=` operatörü ikili bir operatördür (her iki tarafta da bir değer gerektirir), ancak ilk değer eksik olduğundan `test` komutu bunun yerine tekli bir operatör (örneğin `-z`) bekler. Ayrıca, `test` başarısız olduğu için (hata nedeniyle), `if` komutu sıfır olmayan bir çıkış kodu alır ve buna göre hareket eder ve ikinci `echo` komutu yürütülür.

Bu sorun, `test`

komutunun ilk argümanının etrafına tırnak ekleyerek düzeltilebilir.

```
[ "$number" = 1 ]
```

Ardından genişleme gerçekleştiğinde, sonuç şu şekilde olacaktır:

```
[ "" = 1 ]
```

Bu, doğru sayıda argüman verir. Boş dizelere ek olarak, gömülü boşluklar içeren dosya adları gibi, bir değerin çok kelimeli dizelere genişleyebileceği durumlarda tırnak işaretleri kullanılmalıdır.

Not: Kelime bölünmesi gerekmedikçe, değişkenleri ve komut ikamelerini her zaman çift tırnak içine almayı bir kural haline getirin.

Mantıksal Hatalar

Sözdizimsel hatalardan farklı olarak, *mantıksal hatalar* bir komut dosyasının çalışmasını engellemez. Komut dosyası çalışır, ancak mantıkla ilgili bir sorun nedeniyle istenen sonucu vermez. Sayısız mantıksal hata olabilir, ancak komut dosyalarında en sık rastlananlardan bazıları şunlardır:

1. **Yanlış koşullu ifadeler.** if/then/else ifadesini yanlış kodlamak ve yanlış mantığın uygulanmasına neden olmak kolaydır. Bazen mantık tersine dönebilir veya eksik olabilir.
2. **"Bir farkla" hataları.** Sayaç kullanan döngüler kodlarken, sayımın doğru noktada sona ermesi için döngünün sayımın bir yerine sıfırdan başlaması gerektiğini gözden kaçırmak mümkündür. Bu tür hatalar, ya sayımın çok fazla ilerlemesi nedeniyle döngünün "sonuna gelmemesi" ya da bir yinelemeyi erken sonlandırarak son yinelemeyi kaçırmasıyla sonuçlanır.
3. **Öngörülemeyen durumlar.** Çoğu mantık hatası, programcının öngöremediği verilerle veya durumlarla karşılaşan programlardan kaynaklanır. Gördüğümüz gibi, bu, tek bir dosya adı yerine birden fazla komut argümanına genişleyen gömülü boşluklar içeren bir dosya adı gibi öngörülemeyen genişlemeleri de içerebilir.

Savunmacı Programlama

Programlama yaparken varsayımları doğrulamak önemlidir. Bu, bir komut dosyası tarafından kullanılan programların ve komutların çıkış durumunun dikkatli bir şekilde değerlendirilmesi anlamına gelir. İşte gerçek bir hikayeye dayanan bir örnek. Talihsiz bir sistem yöneticisi, önemli bir sunucuda bakım görevi gerçekleştirmek için bir komut dosyası yazdı. Komut dosyası aşağıdaki iki satırlık kodu içeriyordu:

```
cd $dir_name
rm *
```

`Dir_name` değişkeninde adı geçen dizin mevcut olduğu sürece, bu iki satırda esasen yanlış bir şey yoktur. Peki, dizin mevcut değilse ne olur? Bu durumda, `cd` komutu başarısız olur ve komut dosyası bir sonraki satıra geçer ve geçerli çalışma dizinindeki dosyaları siler. Bu, hiç de istenen bir sonuç değildir! Talihsiz yönetici, bu tasarım kararı nedeniyle sunucunun önemli bir bölümünü yok etti.

Bu tasarımın nasıl iyileştirilebileceğine bakalım. İlk olarak, `dir_name` değişkenini tırnak içine alarak tek bir kelimeye genişletilmesini sağlamak ve `rm` komutunun yürütülmesini `cd` komutunun başarısına bağlı hale getirmek akıllıca olabilir.

```
cd "$dir_name" && rm *
```

Bu şekilde, `cd` komutu başarısız olursa, `rm` komutu yürütülmez. Bu daha iyidir, ancak yine de `dir_name` değişkeninin ayarlanmamış veya boş olma olasılığını ortadan kaldırmaz, bu da kullanıcının ana dizinindeki dosyaların silinmesine neden olur. Bu durum, `dir_name'in` gerçekten mevcut bir dizinin adını içerip içermediğini kontrol ederek de önlenebilir.

```
[[ -d "$dir_name" ]] && cd "$dir_name" && rm *
```

Genellikle, yukarıda gösterilen gibi bir durum meydana geldiğinde komut dosyasını sonlandırmak ve bir hata bildirmek için mantık eklemek en iyisidir.

```
# $dir_name dizinindeki dosyaları sil
[[ ! -d "$dir_name" ]]; then
    echo "Böyle bir dizin yok: '$dir_name'" >&2
    exit 1
fi
```

```
if ! cd "$dir_name"; then
    echo "Cannot cd to '$dir_name'" >&2
    exit 1
fi
if ! rm *; then
    echo "Dosya silme işlemi başarısız. Sonuçları kontrol
    edin" >&2 exit 1
fi
```

Burada, hem adın mevcut bir dizin olup olmadığını hem de `cd` komutunun başarısını kontrol ediyoruz. Her ikisi de başarısız olursa, açıklayıcı bir hata mesajı standart hataya gönderilir ve komut dosyası, başarısızlığı belirtmek için bir çıkış durumu ile sonlandırılır.

`set -e, set -u` ve `set -o pipefail`

`Bash` hakkında fark ettiğimiz bir şey, bir komut dosyası çalıştırıldığında ve bir komut başarısız olduğunda (komut dosyasının kendisindeki sözdizimi hataları hariç), komut dosyasının bir sonraki komuta sorunsuz bir şekilde devam etmesidir. Bu genellikle istenmeyen bir durumdur ve POSIX standardı ve ardından `bash` bu sorunu çözmeye çalışır. `bash`, hataları otomatik olarak ele almaya çalışan bir ayar sunar. Bu, bu ayar etkinleştirildiğinde, herhangi bir komut (bazı gerekli istisnalar hariç) sıfır olmayan bir çıkış durumu döndürdüğünde komut dosyasının sonlandırılacağı anlamına gelir. Bu ayarı çağırmak için, komut dosyasının başına `set -e` komutunu yerleştiririz. Birkaç bash kodlama standardı, bu özelliği birkaç ilgili ayarla birlikte kullanmayı şart koşar:
`-u` (başlatılmamış bir değişken varsa komut dosyasını sonlandıran) ve `set -o pipefail` (boru hattındaki son öğe başarısız olursa komut dosyasını sonlandıran) gibi birkaç ilgili ayarla birlikte kullanılmasını şart koşar.

Bu özelliklerin kullanılması önerilmez. İyi kodlama uygulamalarının yerine `set -e'ye` güvenmek yerine, uygun hata işleme tasarımı yapmak daha iyidir.

Bash SSS #105 bu konuda şu görüşü sunar:

> *`set -e`, kabuğa "otomatik hata algılama" özelliği eklemek için yapılan bir girişimdi. Amacı, bir hata oluştuğunda kabuğun her zaman iptal edilmesini sağlamaktı, böylece her önemli komutun sonuna `|| exit 1` eklemenize gerek kalmazdı.*
>
> *Bu amaç önemsiz değildir, çünkü birçok komut kasıtlı olarak sıfırdan farklı bir değer döndürür. Örneğin,*
>
> *`if [ -d /foo ]; then ...; else ...; fi`*
>
> *`[ -d /foo ]` komutu sıfırdan farklı bir değer döndürdüğünde (çünkü dizin mevcut değil) kesintiye uğramak istemediğimiz açıktır -- betiğimiz*

else kısmında. Bu yüzden uygulayıcılar, "if testinin bir parçası olan komutlar bağışık" veya "sonuncusu hariç, bir ardışık düzen içindeki komutlar bağışık" gibi bir dizi özel kural oluşturmaya karar verdiler.

Bu kurallar son derece karmaşıktır ve bazı oldukça basit durumları bile yakalayamamaktadır. Daha da kötüsü, bash bu "özelliğin" son derece kaygan POSIX tanımını takip etmeye çalıştığı için kurallar bir Bash sürümünden diğerine değişmektedir. Bir alt kabuk söz konusu olduğunda durum daha da kötüleşir; davranış, bash'ın POSIX modunda çağrılıp çağrılmadığına bağlı olarak değişir.

ShellCheck sizin dostunuzdur

Çoğu dağıtım deposunda, shell betiklerini analiz eden ve birçok türde hatayı ve kötü betik uygulamalarını tespit eden `shellcheck` adlı bir program bulunmaktadır. Betiklerimizin kalitesini kontrol etmek için bu programı kullanmaya değer. Shebang içeren bir betikle kullanmak için şunu yapmanız yeterlidir:

```
shellcheck my_script
```

ShellCheck, shebang'e göre hangi kabuk lehçesinin kullanılacağını otomatik olarak algılar. İşlev kütüphaneleri gibi shebang içermeyen komut dosyası kodlarını test etmek için ShellCheck'i şu şekilde kullanabiliriz:

```
shellcheck -s bash my_library
```

İstenen kabuk lehçesini belirtmek için `-s` seçeneğini kullanın. ShellCheck hakkında daha fazla bilgiyi https://www.shellcheck.net web sitesinde bulabilirsiniz.

Dosya Adlarına Dikkat Edin

Bu dosya silme komut dosyasında daha belirsiz ancak çok tehlikeli olabilecek başka bir sorun daha vardır. Birçok kişinin görüşüne göre, Unix (ve Unix benzeri işletim sistemleri) dosya adları konusunda ciddi bir tasarım hatasına sahiptir. Unix bu konuda son derece hoşgörülüdür. Aslında, dosya adına eklenemeyen sadece iki karakter vardır. İlki, yol adının öğelerini ayırmak için kullanılan / karakteridir ve ikincisi, dizelerin sonunu işaretlemek için dahili olarak kullanılan boş karakterdir (sıfır bayt). Boşluklar, sekmeler, satır beslemeleri, baştaki tireler, satır başı dönüşleri vb. dahil olmak üzere diğer her şey geçerlidir.

Özellikle dikkat edilmesi gereken nokta, baştaki tire işaretleridir. Örneğin,

" -rf ~" adlı bir dosya oluşturmak tamamen yasaldır. Bu dosya adı `rm` komutuna aktarıldığında ne olacağını bir düşünün.

Bu soruna karşı önlem almak için, dosya silme komut dosyasında `rm` komutunu

```
rm *
```

'den

```
rm ./*
```

Bu, tire ile başlayan bir dosya adının komut seçeneği olarak yorumlanmasını önleyecektir. Genel bir kural olarak, komutlar tarafından yanlış yorumlanmasını önlemek için joker karakterlerin (örneğin `*` ve `?`) önüne her zaman `./` ekleyin. Buna, örneğin `*.pdf` ve `???.mp3` gibi şeyler dahildir.

Taşınabilir Dosya Adları

Bir dosya adının birden fazla platformda (yani farklı türdeki bilgisayarlar ve işletim sistemlerinde) taşınabilir olmasını sağlamak için, dosya adına dahil edilen karakterleri sınırlamak için özen gösterilmelidir. Bir dosya adının farklı sistemlerde çalışması olasılığını en üst düzeye çıkarmak için kullanılabilecek POSIX Taşınabilir Dosya Adı Karakter Kümesi adlı bir standart vardır. Standart oldukça basittir. İzin verilen tek karakterler büyük harfler A-Z, küçük harfler a-z, rakamlar 0-9, nokta (.), tire (-) ve alt çizgi (_) karakterleridir. Standart ayrıca dosya adlarının tire ile başlamaması gerektiğini önerir.

Giriş Doğrulama

İyi programlamanın genel bir kuralı, bir program giriş kabul ediyorsa, aldığı her şeyi işleyebilmesi gerektiğidir. Bu genellikle, daha sonraki işlemler için yalnızca geçerli girişlerin kabul edilmesini sağlamak için girişlerin dikkatlice taranması gerektiği anlamına gelir. Bunun bir örneğini önceki bölümde `read` komutunu incelerken görmüştük. Bir komut dosyası, menü seçimini doğrulamak için aşağıdaki testi içeriyordu:

```
[[ $REPLY =~ ^[0-3]$ ]]
```

Bu test çok spesifiktir. Yalnızca kullanıcı tarafından girilen dize sıfır ile üç arasındaki bir sayı ise sıfır çıkış durumu döndürür. Başka hiçbir şey kabul edilmez. Bazen bu tür testleri yazmak zor olabilir, ancak yüksek kaliteli bir komut dosyası oluşturmak için bu çaba gereklidir.

Tasarım, Zamanın Bir Fonksiyonudur

Üniversitede endüstriyel tasarım okurken, bilge bir profesörüm, bir projedeki tasarım miktarının tasarımcıya verilen süreye bağlı olduğunu söylemişti. "Sinekleri öldüren" bir cihaz tasarlamak için beş dakikanız varsa, sinek sinekliği tasarlarsınız. Beş ayınız varsa, bunun yerine lazer güdümlü bir "sinek önleme sistemi" tasarlayabilirsiniz.

Aynı ilke programlama için de geçerlidir. Bazen, bir kez ve sadece programcı tarafından kullanılacaksa, "hızlı ve basit" bir komut dosyası yeterli olabilir. Bu tür komut dosyaları yaygındır ve çabayı ekonomik hale getirmek için hızlı bir şekilde geliştirilmelidir. Bu tür komut dosyaları çok fazla yorum ve savunma kontrolüne ihtiyaç duymaz. Öte yandan, bir komut dosyası *üretim amaçlı* kullanılacaksa, yani önemli bir görev için veya birden fazla kullanıcı tarafından tekrar tekrar kullanılacaksa, çok daha dikkatli bir şekilde geliştirilmesi gerekir.

Test

Test etme, komut dosyaları dahil olmak üzere her türlü yazılım geliştirmede önemli bir adımdır. Açık kaynak dünyasında, bu gerçeği yansıtan "erken yayınla, sık yayınla" diye bir söz vardır. Erken ve sık yayınlayarak, yazılımın kullanımı ve test edilmesi daha fazla görünürlük kazanır. Deneyimler, hataların geliştirme döngüsünün erken aşamalarında bulunması halinde, bunların bulunmasının çok daha kolay ve düzeltilmesinin çok daha ucuz olduğunu göstermiştir.

26. bölümde, program akışını doğrulamak için stub'ların nasıl kullanılabileceğini gördük. Komut dosyası geliştirmenin en erken aşamalarından itibaren, bunlar çalışmamızın ilerlemesini kontrol etmek için değerli bir tekniktir.

Daha önce gösterilen dosya silme sorununa bakalım ve bunun kolay test için nasıl kodlanabileceğini görelim. Orijinal kod parçasını test etmek, amacı dosyaları silmek olduğu için tehlikeli olabilir, ancak testi güvenli hale getirmek için kodu değiştirebiliriz.

```
if [[ -d $dir_name ]]; then
    if cd $dir_name; then
         echo rm * # TEST
    else
         echo "cannot cd to '$dir_name'" >&2
         exit 1
    fi
else
    echo "böyle bir dizin yok: '$dir_name'" >&2
```

```
    çıkış 1
fi
çıkış # TEST
```

Hata koşulları zaten yararlı mesajlar çıktığı için, herhangi bir şey eklememize gerek yoktur. En önemli değişiklik, `rm` komutunun hemen önüne bir `echo` komutu yerleştirerek, komutun ve genişletilmiş argüman listesinin görüntülenmesini sağlamak, komutun gerçekten yürütülmesini engellemektir. Bu değişiklik, kodun güvenli bir şekilde yürütülmesini sağlar. Kod parçasının sonunda, testi sonlandırmak ve komut dosyasının diğer bölümlerinin yürütülmesini engellemek için bir `exit` komutu yerleştiririz. Bunun gerekliliği, komut dosyasının tasarımına göre değişir.

Ayrıca, testle ilgili değişiklikler için "işaretleyici" görevi gören bazı yorumlar da ekliyoruz. Bunlar, test tamamlandığında değişiklikleri bulmak ve kaldırmak için kullanılabilir.

Test Durumları

Yararlı testler gerçekleştirmek için, iyi *test senaryoları* geliştirmek ve uygulamak önemlidir. Bu, *uç* ve *köşe* durumlarını yansıtan giriş verilerini veya çalışma koşullarını dikkatlice seçerek yapılır. Basit kod parçacığımızda, kodun üç özel koşul altında nasıl çalıştığını bilmek istiyoruz:

1. `dir_name` mevcut bir dizinin adını içerir.
2. `dir_name,` var olmayan bir dizinin adını içerir.
3. dir`_name` boş.

Bu koşulların her biriyle test gerçekleştirilerek, iyi *bir test kapsamı* elde edilir.

Tasarımda olduğu gibi, test de zamanın bir fonksiyonudur. Her komut dosyası özelliğinin kapsamlı bir şekilde test edilmesi gerekmez. Aslında en önemli olanı belirlemek meselesidir. Arıza durumunda çok yıkıcı sonuçlar doğurabileceğinden, kod parçacığımız hem tasarım hem de test aşamasında dikkatle incelenmelidir.

Hata ayıklama

Testler bir komut dosyasında sorun olduğunu ortaya çıkarsa, bir sonraki adım hata ayıklamadır. Bir "sorun" genellikle komut dosyasının bir şekilde programcının beklentilerine uygun şekilde çalışmadığı anlamına gelir. Bu durumda, komut dosyasının gerçekte ne yaptığını ve neden yaptığını dikkatlice belirlememiz gerekir. Hataları bulmak bazen çok fazla dedektiflik çalışması gerektirebilir.

İyi tasarlanmış bir komut dosyası bu konuda yardımcı olacaktır. Anormal durumları tespit etmek ve kullanıcıya yararlı geri bildirim sağlamak için savunmacı bir şekilde programlanmalıdır. Ancak bazen sorunlar oldukça garip ve beklenmedik olabilir ve daha karmaşık teknikler gerekebilir.

Sorunlu Alanı Bulma

Bazı komut dosyalarında, özellikle uzun olanlarda, bazen sorunla ilgili komut dosyasının alanını izole etmek yararlı olabilir. Bu her zaman gerçek hata olmayabilir, ancak izolasyon genellikle gerçek neden hakkında fikir verir. Kodu izole etmek için kullanılabilecek bir teknik, komut dosyasının bölümlerini "yorumlamak"tır. Örneğin, dosya silme parçacığımız, kaldırılan bölümün bir hatayla ilgili olup olmadığını belirlemek için değiştirilebilir.

```
if [[ -d $dir_name ]]; then
    if cd $dir_name; then
         rm *
    else
         echo "cannot cd to '$dir_name'" >&2
         exit 1
    fi
 # else
#   echo "böyle bir dizin yok: '$dir_name'" >&2
#   çıkış 1
fi
```

Bir komut dosyasının mantıksal bölümündeki her satırın başına yorum sembolleri yerleştirerek, o bölümün yürütülmesini engelleriz. Ardından, kodun kaldırılmasının hatanın davranışına herhangi bir etkisi olup olmadığını görmek için test tekrar yapılabilir.

İzleme

Hatalar genellikle komut dosyası içindeki beklenmedik mantıksal akış durumlarıdır. Yani, komut dosyasının bazı bölümleri hiç yürütülmüyor veya yanlış sırayla ya da yanlış zamanda yürütülüyor. Programın gerçek akışını görmek için *izleme* adı verilen bir teknik kullanırız.

Bir izleme yöntemi, komut dosyasına yürütme konumunu gösteren bilgilendirici mesajlar yerleştirmektir. Kod parçacığımıza mesajlar ekleyebiliriz.

```
echo "dosyaları silmeye hazırlanıyor" >&2
if [[ -d $dir_name ]]; then
    if cd $dir_name; then
echo "dosyaları siliyorum" >&2
         rm *
    else
         echo "cannot cd to '$dir_name'" >&2
         exit 1
    fi
```

```
else
    echo "böyle bir dizin yok: '$dir_name'" >&2
    exit 1
fi
echo "dosya silme işlemi tamamlandı" >&2
```

Mesajları normal çıktılardan ayırmak için standart hataya göndeririz. Ayrıca, mesajları içeren satırları girintilemeyiz, böylece bunları kaldırma zamanı geldiğinde bulmak daha kolay olur.

Şimdi komut dosyası çalıştırıldığında, dosya silme işleminin gerçekleştirildiğini görebiliriz.

```
[me@linuxbox ~]$ deletion-script
dosyaları silmeye hazırlanıyor
dosyaları siliyor
dosya silme işlemi tamamlandı
[me@linuxbox ~]$
```

`bash` ayrıca `-x` seçeneği ve `-x` seçeneği ile `set` komutu tarafından uygulanan bir izleme yöntemi de sağlar. Daha önceki `sorun` senaryomuzu kullanarak, ilk satıra `-x` seçeneğini ekleyerek tüm senaryo için izlemeyi etkinleştirebiliriz.

```
#!/bin/bash -x

# trouble: yaygın hataları gösteren komut dosyası

number=1

if [ $number = 1 ]; then
    echo "Sayı 1'e eşittir." else
    echo "Sayı 1'e eşit değil."
fi
```

Çalıştırıldığında, sonuçlar şöyle görünür:

```
[me@linuxbox ~]$ trouble
+ number=1
```

```
+ '[' 1 = 1 ']'
+ echo 'Sayı 1'e eşittir.' Sayı 1'e
eşittir.
```

İzleme etkinleştirildiğinde, genişletmeler uygulanarak gerçekleştirilen komutları görürüz. Öndeki artı işaretleri, izlemeyi normal çıktı satırlarından ayırt etmek için izlemenin görüntülendiğini gösterir. Artı işareti, izleme çıktısı için varsayılan karakterdir. `PS4` (komut istemi dizesi 4) kabuk değişkeninde bulunur. Bu değişkenin içeriği, komut istemini daha kullanışlı hale getirmek için ayarlanabilir. Burada, değişkenin içeriğini, izlemenin gerçekleştirildiği komut dosyasında geçerli satır numarasını içerecek şekilde değiştiriyoruz. Komut istemi gerçekten kullanılana kadar genişlemeyi önlemek için tek tırnak işaretlerinin gerekli olduğunu unutmayın.

```
[me@linuxbox ~]$ export PS4='$LINENO + '
[me@linuxbox ~]$ trouble
5 + number=1
7 + '[' 1 = 1 ']'
8 + echo 'Sayı 1'e eşittir.' Sayı 1'e
eşittir.
```

Komut dosyasının tamamı yerine seçilen bir bölümünde izleme yapmak için, `-x` seçeneği ile `set` komutunu kullanabiliriz.

```
#!/bin/bash

# sorun: yaygın hataları gösteren komut dosyası

sayı=1

set -x # İzlemeyi aç
if [ $number = 1 ]; then
    echo "Sayı 1'e eşittir." else
    echo "Sayı 1'e eşit değil."
fi
set +x # İzlemeyi kapat
```

İzlemeyi etkinleştirmek için `-x` seçeneği ile `set` komutunu, izlemeyi devre dışı bırakmak için `+x` seçeneği ile `set` komutunu kullanırız. Bu teknik, sorunlu bir komut dosyasının birden fazla bölümünü incelemek için kullanılabilir.

Yürütme Sırasında Değerleri İnceleme

Çoğu zaman, izleme ile birlikte, bir komut dosyasının yürütülürken iç işleyişini görmek için değişkenlerin içeriğini görüntülemek yararlıdır. Ek `echo` ifadeleri uygulamak genellikle bu işi görür.

```
#!/bin/bash

# sorun: yaygın hataları gösteren komut dosyası number=1

echo "number=$number" # DEBUG
set -x # İzlemeyi aç if [
$number = 1 ]; then
    echo "Sayı 1'e eşittir." else
    echo "Sayı 1'e eşit değil."
fi
set +x # İzlemeyi kapat
```

Bu basit örnekte, değişken sayının değerini görüntüleyip, daha sonra tanımlanması ve kaldırılması kolay olsun diye eklenen satırı bir yorumla işaretliyoruz. Bu teknik, komut dosyalarındaki döngülerin ve aritmetik işlemlerin davranışını izlerken özellikle yararlıdır.

Özet

Bu bölümde, komut dosyası geliştirme sırasında ortaya çıkabilecek sorunlardan sadece birkaçını ele aldık. Elbette, daha pek çok sorun vardır. Burada açıklanan teknikler, en yaygın hataları bulmanızı sağlayacaktır. Hata ayıklama, hem hataları nasıl önleyeceğinizi (geliştirme süresince sürekli test etme) hem de hataları nasıl bulacağınızı (izlemeyi etkili bir şekilde kullanma) bilerek, deneyim yoluyla geliştirilen bir sanattır.

Daha Fazla Okuma

- Wikipedia'da sözdizimsel ve mantıksal hatalarla ilgili birkaç kısa makale bulunmaktadır: https://en.wikipedia.org/wiki/Syntax_error https://en.wikipedia.org/wiki/Logic_error
- `Bash` programlamanın teknik yönleri ile ilgili birçok çevrimiçi kaynak bulunmaktadır: https://mywiki.wooledge.org/BashPitfalls https://tldp.org/LDP/abs/html/gotchas.html

https://www.gnu.org/software/bash/manual/html_node/Reserved-Word-Index.html

- David Wheeler, Unix dosya adı sorunu ve bununla başa çıkmak için kabuk betiklerini nasıl kodlayacağınız konusunda mükemmel bir tartışma sunmaktadır: https://www.dwheeler.com/essays/filenames-in-shell.html
- Gerçekten zorlu hata ayıklama işlemleri için Bash Debugger vardır: https://bashdb.sourceforge.net/

31 – Akış Kontrolü: `Case` ile Dallanma

Bu bölümde, akış kontrolüne bakmaya devam edeceğiz. 28. bölümde, bazı basit menüler oluşturduk ve kullanıcının seçimine göre hareket etmek için kullanılan mantığı geliştirdik. Bunu yapmak için, olası seçeneklerden hangisinin seçildiğini belirlemek için bir dizi `if` komutu kullandık. Bu tür mantıksal yapılar programlarda sıklıkla görülür, o kadar ki birçok programlama dili (kabuk dahil) çoktan seçmeli kararlar için özel bir akış kontrol mekanizması sağlar.

`case`

`Bash'ta`, çoktan seçmeli bileşik komut `case olarak` adlandırılır. Aşağıdaki sözdizimine sahiptir:

```
case kelime in
      [desen [| desen]...) komutlar ;;]...
esac
```

28. bölümdeki read-menu programına bakarsak, kullanıcının seçimine göre hareket etmek için kullanılan mantığı görebiliriz.

```
#!/bin/bash

# read-menu: menü tabanlı bir sistem bilgisi programı

clear
echo "
Lütfen Seçin:

1. Sistem Bilgilerini Görüntüle
2. Disk Alanını Görüntüle
3. Ana Sayfa Alanı Kullanımını Görüntüle
0. Çıkış
"
read -r -p "Seçimi girin [0-3] > "
```

```
if [[ "$REPLY" =~ ^[0-3]$ ]]; then
    if [[ "$REPLY" == 0 ]]; then
         echo "Program sonlandırıldı."
         exit
    fi
    if [[ "$REPLY" == 1 ]]; then echo
         "Ana bilgisayar adı:
         $HOSTNAME" uptime
         çıkış
    fi
    if [[ "$REPLY" == 2 ]]; then
         df -h
         çık
    fi
    eğer [[ "$REPLY" == 3 ]]; o zaman
         if [[ "$(id -u)" -eq 0 ]]; then
              echo "Ana Sayfa Alanı Kullanımı (Tüm
              Kullanıcılar)" du -sh /home/*
         else
              echo "Ana Sayfa Alanı Kullanımı ($USER)"
              du -sh "$HOME"
         fi
         exit
    fi
else
    echo "Geçersiz giriş." >&2
    exit 1
fi
```

`Case` kullanarak, bu mantığı daha basit bir şeyle değiştirebiliriz.

```
#!/bin/bash

# case-menu: menü tabanlı bir sistem bilgisi programı clear
echo "
Lütfen Seçin:

1. Sistem Bilgilerini Görüntüle
2. Disk Alanını Görüntüle
3. Ana Sayfa Alanı Kullanımını Görüntüle
```

```
0. Çıkış
read -r -p "Seçimi girin [0-3] > "

case "$REPLY" in
    0)   echo "Program sonlandırıldı."
         exit
         ;;
    1)   echo "Ana bilgisayar adı:
         $HOSTNAME" uptime
         ;;
    2)   df -h
         ;;
    3)   if [[ "$(id -u)" -eq 0 ]]; then
              echo "Ev Alanı Kullanımı (Tüm Kullanıcılar)" du
              -sh /home/*
         else
              echo "Ana Sayfa Alanı Kullanımı ($USER)"
              du -sh "$HOME"
         fi
         ;;
    *)   echo "Geçersiz giriş" >&2
         exit 1
         ;;
esac
```

`case` komutu, örneğimizde REPLY değişkeninin değeri olan *word* değerine bakar ve ardından bunu belirtilen *desenlerden* biriyle eşleştirmeye çalışır. Eşleşme bulunduğunda, belirtilen desenle ilişkili *komutlar* yürütülür. Eşleşme bulunduktan sonra, başka eşleşme denemesi yapılmaz.

Desenler

`case` tarafından kullanılan desenler, yol adı genişletmede kullanılanlarla aynıdır. Desenler `)` karakteriyle sonlandırılır. Tablo 31-1, geçerli desen örneklerini listeler.

Tablo 31- 1: `case` Desen Örnekleri

Desen	Açıklama
`a`	*Kelime* "a" ile eşleşirse.
`[[:alpha:]])`	*Kelime* tek bir alfabetik karakter ise eşleşir.
`???)`	*Kelime* tam olarak üç karakter uzunluğunda ise eşleşir.

`*.txt)`	*Kelime* ".txt" karakterleriyle bitiyorsa eşleşir.
`*)`	*Kelimenin* herhangi bir değerini eşleştirir. Bunu, önceki desenle eşleşmeyen *kelimenin* herhangi bir değerini, yani olası geçersiz değerleri yakalamak için, `case` komutunda son desen olarak eklemek iyi bir uygulamadır.

İşte kalıpların nasıl çalıştığına dair bir örnek:

```
#!/bin/bash

read -r -p "Kelimeyi girin >

" case "$REPLY" in
    [[:alpha:]]) echo "tek bir alfabetik karakterdir." ;;
    [ABC][0-9])    echo "A, B veya C'nin ardından bir rakam
    gelir." ;;
    ???)           echo "üç karakter uzunluğundadır." ;;
    *.txt)         echo "'.txt' ile biten bir kelimedir." ;;
    *)             echo "başka bir şeydir." ;;
esac
```

Dikey çubuk karakterini ayırıcı olarak kullanarak birden fazla kalıbı birleştirmek de mümkündür. Bu, "veya" koşullu kalıp oluşturur. Bu, büyük ve küçük harfleri birlikte işlemek gibi durumlarda kullanışlıdır. İşte bir örnek:

```
#!/bin/bash

# case-menu: menü tabanlı bir sistem bilgisi programı clear
echo "
Lütfen Seçin:

A. Sistem Bilgilerini Göster
B. Disk Alanını Görüntüle
C. Ana Sayfa Alanı Kullanımını Görüntüle
Q. Çıkış
"
read -r -p "Seçiminizi girin [A, B, C veya Q] > "
```

```
case "$REPLY" in
    q|Q) echo "Program sonlandırıldı."
         exit
         ;;
    a|A) echo "Ana bilgisayar adı:
         $HOSTNAME" uptime
         ;;
    b|B) df -h
         ;;
    c|C) if [[ "$(id -u)" -eq 0 ]]; then
              echo "Ev Alanı Kullanımı (Tüm Kullanıcılar)" du
              -sh /home/*
         else
              echo "Ana Sayfa Alanı Kullanımı ($USER)"
              du -sh "$HOME"
         fi
         ;;
    *)   echo "Geçersiz giriş" >&2
         exit 1
         ;;
esac
```

Burada, menü seçimi için rakamlar yerine harfler kullanmak üzere `case-menu` programını değiştiriyoruz. Yeni desenlerin hem büyük hem de küçük harflerin girilmesine izin verdiğine dikkat edin.

Birden Çok Eylem Gerçekleştirme

4.0 öncesindeki `bash` sürümlerinde, `case` başarılı bir eşleşmede yalnızca bir eylemin gerçekleştirilmesine izin veriyordu. Başarılı bir eşleşmeden sonra komut sonlandırılıyordu. Burada bir karakteri test eden bir komut dosyası görüyoruz:

```
#!/bin/bash

# case4-1: bir karakteri test etme

read -r -n 1 -p "Bir karakter yazın > "
echo
case "$REPLY" in
    [[:upper:]])     echo "'$REPLY' büyük harftir." ;;
    [[:küçük:]])     echo "'$REPLY' küçük harftir." ;;
    [[:alpha:]])     echo "'$REPLY' alfabetiktir." ;;
```

```
    [[:rakam:]])     echo "'$REPLY' bir rakamdır." ;;
    [[:graph:]])     echo "'$REPLY' görünür bir karakterdir." ;;
    [[:punct:]])     echo "'$REPLY' bir noktalama işaretidir." ;;
    [[:space:]])     echo "'$REPLY' bir boşluk karakteridir." ;;
    [[:xdigit:]])    echo "'$REPLY' onaltılık bir rakamdır." ;;
esac
```

Bu komut dosyasını çalıştırdığınızda şu sonuç elde edilir:

```
[me@linuxbox ~]$ case4-1
Bir karakter yazın > a
'a' küçük harftir.
```

Komut dosyası çoğunlukla çalışır, ancak bir karakter birden fazla POSIX karakter sınıfıyla eşleşirse başarısız olur. Örneğin, "a" karakteri hem küçük harf hem de alfabetik karakterdir, aynı zamanda onaltılık bir rakamdır. 4.0 sürümünden önceki `bash` sürümlerinde, `büyük/küçük harflerin` birden fazla testle eşleşmesi mümkün değildi. Modern `bash` sürümleri, her eylemi sonlandırmak için `;;&` notasyonunu ekler, böylece artık şunu yapabiliriz:

```
#!/bin/bash

# case4-2: bir karakteri test et

read -r -n 1 -p "Bir karakter yazın > "
echo
case "$REPLY" in
    [[:upper:]])     echo "'$REPLY' büyük harftir." ;;&
    [[:küçük:]])     echo "'$REPLY' küçük harftir." ;;&
    [[:alpha:]])     echo "'$REPLY' alfabetiktir." ;;&
    [[:digit:]])     echo "'$REPLY' bir rakamdır." ;;& [[:graph:]])
                     echo "'$REPLY' görünür bir karakterdir." ;;&
    [[:punct:]])     echo "'$REPLY' noktalama işaretidir." ;;&
    [[:space:]])     echo "'$REPLY' bir boşluk karakteridir." ;;&
    [[:xdigit:]])    echo "'$REPLY' onaltılık bir rakamdır." ;;&
esac
```

Bu komut dosyasını çalıştırdığımızda şunu elde ederiz:

```
[me@linuxbox ~]$ case4-2
```

```
Bir karakter yazın > a
'a' küçük harftir. 'a'
alfabetik bir
karakterdir.
'a' görünür bir karakterdir.
'a' onaltılık bir rakamdır.
```

`;;&` sözdiziminin eklenmesi, `büyük/küçük harf durumunun` basitçe sonlandırılmak yerine bir sonraki teste devam etmesini sağlar.

Özet

`case` komutu, programlama hilelerimize kullanışlı bir ekleme. Bir sonraki bölümde göreceğimiz gibi, belirli türdeki sorunları ele almak için mükemmel bir araç.

Daha Fazla Okuma

- *Bash Referans Kılavuzu'nun* Koşullu Yapılar bölümünde `case` komutu ayrıntılı olarak açıklanmaktadır: https://tiswww.case.edu/php/chet/bash/bashref.html#SEC21
- *Gelişmiş Bash Komut Dosyası Yazma Kılavuzu*, `case` komutunun kullanımına ilişkin daha fazla örnek sunar: https://tldp.org/LDP/abs/html/testbranch.html

32 – Konumsal Parametreler

Şimdiye kadar programlarımızda eksik olan bir özellik, komut satırı seçeneklerini ve argümanlarını kabul etme ve işleme yeteneğidir. Bu bölümde, programlarımızın komut satırının içeriğine erişmesini sağlayan kabuk özelliklerini inceleyeceğiz.

Komut Satırına Erişim

Kabuk, komut satırındaki tek tek kelimeleri içeren *konumsal parametreler* adı verilen bir dizi değişken sağlar. Değişkenler `0'dan 9'a` kadar numaralandırılmıştır. Bunlar şu şekilde gösterilebilir:

```
#!/bin/bash

# posit-param: komut satırı parametrelerini görüntülemek için

komut dosyası echo "
\$0 = $0
\$1 = $1
\$2 = $2
\$3 = 3
\$4 = $4
\$5 = 5
\$6 = 6
\$7 = 7
\$8 = 8
\$9 = $9
"
```

Bu, `$0` ile `$9` arasındaki değişkenlerin değerlerini gösteren basit bir komut dosyasıdır. Komut satırı argümanları olmadan çalıştırıldığında, sonuç şöyledir:

```
[me@linuxbox ~]$ posit-param
```

```
$0 = /home/me/bin/posit-param
$1 =
$2 =
$3 =
 4 =
 5 =
 6 =
 7 =
 8 =
 9 =
```

Hiçbir argüman sağlanmasa bile, `$0` her zaman komut satırında görünen ilk öğeyi, yani yürütülen programın yol adını içerir. Argümanlar sağlandığında, şu sonuçları görürüz:

```
[me@linuxbox ~]$ posit-param a b c d
$0 = /home/me/bin/posit-param
$1 = a
$2 = b
$3 = c
$4 = d
$5 =
 6 =
 7 =
 8 =
 9 =
```

Not**:** Parametre genişletme kullanarak aslında dokuzdan fazla parametreye erişebilirsiniz. Dokuzdan büyük bir sayı belirtmek için, sayıyı `${10}`, `${55}`, `${211}` vb.

Argüman Sayısını Belirleme

Kabuk ayrıca, komut satırındaki argüman sayısını içeren bir değişken olan `$#` sağlar.

```
#!/bin/bash

# posit-param: komut satırı parametrelerini görüntülemek için

komut dosyası echo "
Argüman sayısı: $#
\$0 = $0
\$1 = $1
\$2 = $2
\$3 = 3
\$4 = $4
\$5 = 5
\$6 = 6
\$7 = 7
\$8 = 8
\$9 = $9
"
```

Sonuç şöyledir:

```
[me@linuxbox ~]$ posit-param a b c d

Argüman sayısı: 4
$0 = /home/me/bin/posit-param
$1 = a
$2 = b
$3 = c
$4 = d
$5 =
$6 =
$7 =
$8 =
$9 =
```

`shift` – Birçok Argümana Erişim Sağlama

Peki, programa aşağıdaki gibi çok sayıda argüman verdiğimizde ne olur?

```
[me@linuxbox ~]$ posit-param *
```

```
Argüman sayısı: 82
$0 = /home/me/bin/posit-param
$1 = adresler.ldif
$2 = bin
$3 = yer imleri.html
$4 = debian-500-i386-netinst.iso
$5 = debian-500-i386-netinst.jigdo
$6 = debian-500-i386-netinst.template
$7 = debian-cd_info.tar.gz
$8 = Masaüstü
$9 = dirlist-bin.txt
```

Bu örnek sistemde, joker karakter * 82 argümana genişler. Bu kadar çok argümanı nasıl işleyebiliriz? Kabuk, bunu yapmak için beceriksiz de olsa bir yöntem sunar. `Shift` komutu, her çalıştırıldığında tüm parametrelerin "bir aşağı kaymasını" sağlar. Aslında, `shift` kullanarak, tek bir parametreyle (asla değişmeyen `$0'a` ek olarak) idare etmek mümkündür.

```
#!/bin/bash

# posit-param2: tüm argümanları görüntülemek için komut

dosyası count=1

while [[ $# -gt 0 ]]; do
    echo "Argüman $count = $1"
    count=$((count + 1))
    shift
done
```

`Shift` her çalıştırıldığında, `$2` değerinin değeri `$1`'e, `$3` değerinin değeri `$2'ye` taşınır ve bu şekilde devam eder. `$#` değerinin değeri de bir azaltılır.

posit-param2 programında, kalan argüman sayısını değerlendiren ve en az bir argüman olduğu sürece devam eden bir döngü oluşturuyoruz. Mevcut argümanı görüntülüyoruz, işlenen argüman sayısının çalışan sayısını sağlamak için döngünün her yinelemesinde `count` değişkenini artırıyoruz ve son olarak, bir sonraki argümanla
`$1'i` bir sonraki argümanla yüklemek için bir kaydırma işlemi gerçekleştiririz. İşte programın çalışması:

```
[me@linuxbox ~]$ posit-param2 a b c d
Argument 1 = a
Argüman 2 = b
Argüman 3 = c
Argüman 4 = d
```

Basit Uygulamalar

`Shift` olmadan bile, konumsal parametreleri kullanarak yararlı uygulamalar yazmak mümkündür. Örnek olarak, işte basit bir dosya bilgisi programı:

```
#!/bin/bash

# file-info: basit dosya bilgisi programı

PROGNAME="$(basename "$0")"

if [[ -e "$1" ]]; then echo
    -e "\nDosya Türü:" file
    "$1"
    echo -e "\nDosya Durumu:"
    stat "$1"
else
    echo "$PROGNAME: kullanım: $PROGNAME dosya" >&2
    exit 1
fi
```

Bu program, belirtilen bir dosyanın dosya türünü (`file` komutu ile belirlenir) ve dosya durumunu (`stat` komutu ile belirlenir) görüntüler. Bu programın ilginç bir özelliği PROGNAME değişkenidir. Bu değişkene, `basename "$0"` komutunun sonucu olan değer verilir. `Basename` komutu, yol adının başındaki kısmı kaldırarak yalnızca dosyanın temel adını bırakır. Örneğimizde, `basename`, $0 parametresinde bulunan yol adının baş kısmını, yani örnek programımızın tam yol adını kaldırır. Bu değer, programın sonundaki kullanım mesajı gibi mesajlar oluştururken kullanışlıdır. Bu şekilde kodlayarak, komut dosyasının adı değiştirilebilir ve mesaj otomatik olarak programın adını içerecek şekilde ayarlanır.

Kabuk İşlevleriyle Konumsal Parametreleri Kullanma

Konumsal parametreler, kabuk betiklerine argüman aktarmak için kullanıldığı gibi, kabuk işlevlerine argüman aktarmak için de kullanılabilir.

kabuk işlevlerine argüman aktarmak için de kullanılabilir. Bunu göstermek için, `file_info` betiğini bir kabuk işlevine dönüştüreceğiz.

```
file_info () {

    # file_info: dosya bilgilerini görüntülemek için

    işlev if [[ -e "$1" ]]; then
         echo -e "\nDosya Türü:"
         file "$1"
         echo -e "\nDosya Durumu:"
         stat "$1"
    else
         echo "$FUNCNAME: kullanım: $FUNCNAME dosya" >&2
         return 1
    fi
}
```

Şimdi, `file_info` kabuk işlevini içeren bir komut dosyası, işlevini bir dosya adı argümanıyla çağırırsa, argüman işlevine aktarılacaktır.

Bu özellik sayesinde, yalnızca komut dosyalarında değil, `.bashrc` dosyalarımızda da kullanılabilen birçok yararlı kabuk işlevi yazabiliriz.

`PROGNAME` değişkeninin kabuk değişkeni `FUNCNAME` olarak değiştirildiğine dikkat edin. Kabuk, şu anda yürütülen kabuk işlevini takip etmek için bu değişkeni otomatik olarak günceller. `$0'ın` her zaman komut satırındaki ilk öğenin tam yol adını (yani programın adını) içerdiğini ve beklediğimiz gibi kabuk işlevinin adını içermediğini unutmayın.

Konumsal Parametreleri *Toplu Olarak* İşleme

Bazen tüm konumsal parametreleri bir grup olarak yönetmek yararlı olabilir. Örneğin, başka bir programın etrafına bir "sarmalayıcı" yazmak isteyebiliriz. Bu, başka bir programın çağrılmasını basitleştiren bir betik veya kabuk işlevi oluşturduğumuz anlamına gelir. Bu durumda sarmalayıcı, gizemli komut satırı seçeneklerinin bir listesini sağlar ve ardından argümanların bir listesini alt düzey programa iletir.

Kabuk, bu amaçla iki özel parametre sağlar. Her ikisi de konum parametrelerinin tam listesine genişler, ancak oldukça ince farklarla birbirinden ayrılır. Bunlar Tablo 32-1'de açıklanmaktadır.

*Tablo 32-1: * ve @ Özel Parametreleri*

Parametre	Açıklama
$	1 ile başlayan konumsal parametrelerin listesine genişler. Çift tırnak işaretleri ile çevrili olduğunda, her biri IFS kabuk değişkeninin ilk karakteri (varsayılan olarak boşluk karakteri) ile ayrılmış tüm konumsal parametreleri içeren çift tırnak işaretli bir dizeye genişler.
$	1'den başlayarak konumsal parametrelerin listesine genişler. Çift tırnak işaretleri ile çevrili olduğunda, her konumsal parametreyi çift tırnak işaretleri ile çevriliymiş gibi ayrı bir kelimeye genişler.

İşte bu özel parametrelerin nasıl çalıştığını gösteren bir komut dosyası:

```
#!/bin/bash

# posit-params3: $* ve $@'yi gösteren komut dosyası

print_params () {
    echo "\$1 = $1"
    echo "\$2 = $2"
    echo "\$3 = $3"
    echo "\$4 = $4"
}

pass_params () {
    echo -e "\n" '$* :';    print_params $*
    echo -e "\n" '"$*" :'; print_params "$*"
    echo -e "\n" '$@ :';    print_params $@
    echo -e "\n" '"$@" :'; print_params "$@"
}

pass_params "kelime" "boşluklu kelimeler"
```

Bu oldukça karmaşık programda, `word` ve `boşluklu kelimeler` olmak üzere iki argüman oluşturuyoruz ve bunları pass_params işlevine aktarıyoruz. Bu işlev de, özel parametreler `$*` ve `$@` ile kullanılabilen dört yöntemin her birini kullanarak bunları `print_params` işlevine aktarıyor. Komut dosyası çalıştırıldığında, farklılıklar ortaya çıkıyor.

```
[me@linuxbox ~]$ posit-param3

 $* :
$1 = word
$2 = kelime
$3 = ile
$4 = boşluk

 "$*" :
$1 = boşluklu kelimeler
$2 =
3 $ =
4 $ =

 $@ :
$1 = kelime
$2 = kelime
$3 = ile
$4 = boşluk

 "$@" :
$1 = kelime
$2 = boşluklu kelimeler
$3 =
$4 =
```

Bizim argümanlarımızla, hem `$*` hem de `$@` dört kelimelik bir sonuç üretir.

```
boşluklu kelimeler
```

`"$*"` tek kelimelik bir sonuç üretir.

```
"boşluklu kelimeler"
```

`"$@"` iki kelimelik bir sonuç üretir.

```
"kelime" "boşluklu kelimeler"
```

Bu, gerçek amacımıza uygundur. Buradan çıkarılacak ders, kabuk konum parametrelerinin listesini elde etmek için dört farklı yol sunsa da, `"$@"` çoğu durumda en kullanışlı olanıdır, çünkü her konum parametresinin bütünlüğünü korur. Güvenliği sağlamak için, kullanmamak için zorlayıcı bir nedenimiz olmadığı sürece, her zaman kullanılmalıdır.

Daha Tamamlanmış Bir Uygulama

Uzun bir aradan sonra, 27. bölümde gördüğümüz sys_info_page programımız üzerindeki çalışmalara devam edeceğiz. Bir sonraki eklememiz, programa aşağıdaki gibi birkaç komut satırı seçeneği ekleyecektir:

- **Çıktı dosyası.** Programın çıktısını içerecek dosyanın adını belirtmek için bir seçenek ekleyeceğiz. Bu seçenek `-f file` veya `--file file` olarak belirtilecektir.
- **Etkileşimli mod.** Bu seçenek, kullanıcıdan bir çıktı dosya adı girmesi istenecek ve belirtilen dosyanın zaten var olup olmadığını belirleyecektir. Varsa, mevcut dosya üzerine yazılmadan önce kullanıcıya sorulacaktır. Bu seçenek `-i` veya `--interactive` olarak belirtilecektir.
- **Yardım.** Programın bilgilendirici bir kullanım mesajı görüntülemesini sağlamak için `-h` veya `--help` belirtilebilir.

Komut satırı işlemeyi uygulamak için gereken kod şöyledir:

```
usage () {
    echo "$PROGNAME: kullanım: $PROGNAME [-f dosya | -
    i]" return
}

# komut satırı seçeneklerini işleme

interactive=
dosya adı=

while [[ -n "$1" ]]; do
    case "$1" in
         -f | --file)         shift dosya
                              adı="$1"
                              ;;
         -i | --interactive) interactive=1
                              ;;
         -h | --help)
                                   k
                              ullan
                              ım
                              çıkış
                              ;;
    esac *)                   kullanım >&2
    shift                     çıkış 1
bitti                         ;;
```

İlk olarak, yardım seçeneği çağrıldığında veya bilinmeyen bir seçenek denendiğinde bir mesaj görüntülemek için `usage` adlı bir kabuk işlevi ekliyoruz.

Ardından, işleme döngüsüne başlıyoruz. Bu döngü, konumsal parametre `$1` boş değildir. Döngünün sonunda, döngünün sonunda sona ermesini sağlamak için konumsal parametreleri ilerletmek için bir `kaydırma` komutumuz var.

Döngü içinde, desteklenen seçeneklerden herhangi biriyle eşleşip eşleşmediğini görmek için mevcut konumsal parametreyi inceleyen bir `case` ifadesi vardır. Desteklenen bir parametre bulunursa, bu parametreye göre işlem yapılır. Bilinmeyen bir seçenek bulunursa, kullanım mesajı görüntülenir ve komut dosyası bir hata ile sonlandırılır.

`-f` parametresi ilginç bir şekilde işlenir. Algılandığında, ek bir `kayma` gerçekleşir ve bu da konum parametresi `$1'i -f` seçeneğine sağlanan dosya adı argümanına ilerletir.

`getopts` Seçeneği

Daha önce gösterilen konumsal parametre ayrıştırma kodu, elimizdeki göreve iyi bir çözümdür, ancak bu, uygulayabileceğimiz tek yaklaşım değildir. `Getopts` adlı bir kabuk yerleşik komutu vardır (benzer adlı harici `getopt` komutuyla karıştırılmamalıdır) ve bu komut bizim için bazı işleri yapabilir. `Getopts` her çağrıldığında (genellikle bir döngüde), geçerli argümanı belirtilen bir değişkende döndürür ve `OPTIND` sayacını bir sonraki konum parametresini gösterecek şekilde artırır. Bir seçenek argüman gerektiriyorsa, argüman `OPTARG` değişkeninde döndürülür.

`getopts` sözdizimi şöyledir:

```
getopts optstring var [arg ...]
```

`optstring` argümanı, tek karakterli seçenek adlarından oluşan bir dizidir. `getopts`, uzun biçimli seçenekleri (kolayca) desteklemez. Ayrıca, bir seçenek adının ardından iki nokta üst üste karakteri geliyorsa, bu seçenek bir argüman gerektirdiği anlamına gelir.

`getopts` yalnızca tek karakterli seçenek adlarını destekler, ancak `ls` gibi birçok komutta gördüğümüz gibi, tire karakteri olmadan birden fazla seçeneğin birbirine eklenmesine izin verir.

```
ls -la
```

`optstring` iki nokta üst üste ile başlıyorsa, geçersiz bir seçenek veya eksik bir argüman olduğunda `getopts` kendi hata mesajlarını susturur.

var argümanı, geçerli seçenek adını tutacak değişkenin adıdır.

Normalde, getopts konum parametrelerini işler, ancak *var'dan* sonra listelenen ek argümanları da işleyebilir.

getopts yerleşik komutu, işleyecek argüman kalmayıncaya kadar başarılı bir çıkış durumu döndürür.

Evet, bu biraz karmaşık görünebilir, ancak uygulamada gördüğümüzde daha net anlaşılır. getopts kullanarak, bu alternatif tekniğin kısa bir demosunu kodlayabiliriz.

```
#!/bin/bash

# getopts-test: getopts kullanarak komut satırı seçeneklerini işleme

PROGNAME="$(basename "$0")"
interactive=
filename=

kullanım () {
    echo "$PROGNAME: kullanım: $PROGNAME [-f dosya | -i]"
    return
}

while getopts :f:ih opt; do
    case "$opt" in
        f)  dosya adı="$OPTARG" ;;
        i)  interactive=1 ;;
        h) kullanım ;;
        \?) echo "seçenek '$OPTARG' geçersiz" ;;
        :) echo "seçenek '$OPTARG' eksik argüman";;
    esac
bitti
echo "etkileşimli = '$interactive' dosya adı = '$filename'"
```

Bu kodu ayrıntılı olarak inceleyelim. Her zamanki değişken tanımlarımızla ve kullanım mesajı için bir işlev tanımlamasıyla başlıyoruz. Ardından, getopts komutunu bir while döngüsüne yerleştirerek eğlenceli kısma geçiyoruz. Optstring'imiz, hata mesajlarını bastıracak bir iki nokta üst üste ile başlar. İki nokta üst üste'nin ardından seçenek harflerimiz gelir. f seçeneği bir argüman gerektirir (bu durumda bir dosya adı), bu nedenle seçenek harfinin ardından iki nokta üst üste gelir. Son olarak, sonucumuzu tutacak bir değişken (opt) belirleriz. while döngüsü, getopts'un işleyecek parametresi kalmayıncaya kadar devam eder.

While döngüsü içinde, opt içinde döndürülen sonucu işlemek için bir case ifadesi var.

Case ifadesindeki son iki desen hariç, her şey beklendiği gibi görünüyor. ?, geçersiz bir seçenek (yani listede olmayan bir harf) algılandığında getopts tarafından döndürülür. Case ifadesinde, kabuk bunu bir dosya joker karakteri olarak yorumladığı için soru işaretinden kaçınmamız gerektiğini unutmayın. Sonra, bir seçenekte gerekli bir argüman eksik olduğunda getopts tarafından döndürülen : vardır. Bu iki hata durumundan herhangi biri meydana geldiğinde, getopts sorunlu seçenek harfini OPTARG'a yerleştirir.

Bu demoyu çalışırken izleyelim:

```
[me@linuxbox ~]$ getopts-test
interactive = '' filename = ''
[me@linuxbox ~]$ getopts-test -i
interactive = '1' filename = ''
[me@linuxbox ~]$ getopts-test -f foo.html
interactive = '' filename = 'foo.html'
[me@linuxbox ~]$ getopts-test -if foo.html
interactive = '1' filename = 'foo.html'
[me@linuxbox ~]$ getopts-test -i -f foo.html
interactive = '1' filename = 'foo.html'
[me@linuxbox ~]$ getopts-test -a
seçenek 'a' geçersiz
interactive = '' filename = ''
[me@linuxbox ~]$ getopts-test -f
seçenek 'f' eksik argüman
interactive = '' filename = ''
```

Peki, hangi tekniği kullanmalıyız, while/shift mi yoksa getopts mu? Her şey ihtiyaçlarımıza bağlı. While/shift yöntemi en fazla kontrolü sağlar (uzun seçenek adlarının kolay uygulanması dahil), getopts ise daha az kod gerektirir ve tek tireli çoklu seçenek sözdizimini destekler.

Etkileşimli Mod

Konumsal parametre kodu hazır olduğuna göre, etkileşimli modu uygulayalım.

```
# etkileşimli mod

if [[ -n "$interactive" ]]; then
    while true; do
        read -r -p "Çıktı dosyasının adını girin: " dosya_adı
```

```
        if [[ -e "$filename" ]]; then
             read -r -p "'$filename' var. Üzerine yazmak ister misiniz?
             [y/n/q] > " case "$REPLY" in
                  Y|y) break
                       ;;
                  Q|q) echo "Program sonlandırıldı."
                       exit
                       ;;
                  *)   devam
                       ;;
             esac
        elif [[ -z "$filename" ]]; then devam
             et
        else
             break
        fi
    done
fi
```

`Etkileşimli` değişken boş değilse, dosya adı istemini ve ardından mevcut dosya işleme kodunu içeren sonsuz bir döngü başlatılır. İstenen çıktı dosyası zaten mevcutsa, kullanıcıya üzerine yazma, başka bir dosya adı seçme veya programdan çıkma seçenekleri sunulur. Kullanıcı mevcut bir dosyanın üzerine yazmayı seçerse, döngüyü sonlandırmak için bir `kesme` işlemi gerçekleştirilir. `case` ifadesinin yalnızca kullanıcının üzerine yazmayı mı yoksa çıkmayı mı seçtiğini algıladığını dikkat edin. Diğer herhangi bir seçim döngünün devam etmesine ve kullanıcıya tekrar sorulmasına neden olur.

Dosya Çıktısı

Çıktı dosya adı özelliğini uygulamak için, birazdan açıklığa kavuşacak nedenlerden dolayı, önce mevcut sayfa yazma kodunu bir kabuk işlevine dönüştürmeliyiz.

```
write_html_page () {
    cat << _EOF_
<html>
    <head>
         <title>$TITLE</title>
    </head>
    <body>
         <h1>$TITLE</h1>
         <p>$TIMESTAMP</p>
         $(report_uptime)
```

```
        $(report_disk_space)
        $(report_home_space)
    </body>
</html>
_EOF_
    dön
}

# html sayfası çıktısı

if [[ -n "$filename" ]]; then
    if touch "$filename" && [[ -f "$filename" ]]; then
        write_html_page > "$filename"
    else
        echo "$PROGNAME: '$filename' dosyası yazılamıyor" >&2
        exit 1
    fi
else
    write_html_page
fi
```

-f seçeneğinin mantığını işleyen kod, önceki listenin sonunda yer almaktadır. Bu kodda, bir dosya adının varlığını test ediyoruz ve bir dosya bulunursa, dosyanın gerçekten yazılabilir olup olmadığını görmek için bir test gerçekleştiriyoruz. Bunu yapmak için, bir `touch` işlemi gerçekleştiriliyor ve ardından sonuçta ortaya çıkan dosyanın normal bir dosya olup olmadığını belirlemek için bir test yapılıyor. Bu iki test, geçersiz bir yol adı girildiğinde (`touch` işlemi başarısız olur) ve dosya zaten mevcutsa, dosyanın normal bir dosya olup olmadığını kontrol eder.

Gördüğümüz gibi, sayfanın gerçek oluşturulmasını gerçekleştirmek için `write_html_page` işlevi çağrılır. Çıktısı ya standart çıktıya yönlendirilir (`file-name` değişkeni boşsa) ya da belirtilen dosyaya yeniden yönlendirilir. HTML kodu için iki olası hedefimiz olduğundan, gereksiz kodu önlemek için `write_html_page` rutinini bir kabuk işlevine dönüştürmek mantıklıdır.

Özet

Konumsal parametrelerin eklenmesiyle, artık oldukça işlevsel komut dosyaları yazabiliriz. Basit, tekrarlayan görevler için, konumsal parametreler, kullanıcının `.bashrc` dosyasına yerleştirilebilecek yararlı kabuk işlevleri yazmayı mümkün kılar.

`sys_info_page` programımız karmaşıklık ve gelişmişlik açısından büyümüştür. İşte en son değişiklikler kalın yazılmış şekilde tam liste:

```
#!/bin/bash

# sys_info_page: sistem bilgisi sayfasını görüntüleyen program

PROGNAME="$(basename "$0")"
TITLE="$HOSTNAME için Sistem Bilgisi Raporu" CURRENT_TIME="$(date
+"%x %r %Z")"
TIMESTAMP="$CURRENT_TIME tarihinde $USER tarafından oluşturuldu"

report_uptime () {
    cat << _EOF_
    <h2>Sistem Çalışma Süresi</h2>
    <pre>$(uptime)</pre>
_EOF_
    dön
}

report_disk_space () {
    cat << _EOF_
    <h2>Disk Alanı Kullanımı</h2>
    <pre>$(df -h)</pre>
_EOF_
    dön
}

report_home_space () {
    if [[ "$(id -u)" -eq 0 ]]; then
         cat << _EOF_
    <h2>Ana Alan Kullanımı (Tüm Kullanıcılar)</h2>
    <pre>$(du -sh /home/*)</pre>
_EOF_
    else
         cat << _EOF_
    <h2>Ana Sayfa Alanı Kullanımı ($USER)</h2>
    <pre>$(du -sh "$HOME")</pre>
_EOF_
    fi
    dönüş
}

kullanım () {
    echo "$PROGNAME: kullanım: $PROGNAME [-f dosya | -i]"
    return
}
```

```
html_sayfa_yaz () {
    cat << _EOF_
<html>
    <head>
        <title>$TITLE</title>
    </head>
    <body>
        <h1>$TITLE</h1>
        <p>$TIMESTAMP</p>
        $(report_uptime)
        $(report_disk_space)
        $(report_home_space)
    </body>
</html>
_EOF_
    dön
}

# komut satırı seçeneklerini
işleme

etkileşimli= dosya
adı=

while [[ -n "$1" ]]; do
    case "$1" in
        -f | --file)          shift
                              dosyaadı="$1"
                              ;;
        -i | --interaktif) etkileşimli=1
                              ;;
        -h | --help)
                                kul
                              lanım
                              çıkış
                              ;;
        *)                    kullanım >&2
                              çıkış 1
                              ;;
    esac
    vardi
    yası
tamam

# etkileşimli mod
```

```
    true ise; yap
         read -r -p "Çıktı dosyasının adını girin: " dosya adı
         if [[ -e "$dosya adı" ]]; then
              read -r -p "'$filename' var. Üzerine yazmak istiyor musunuz?
              [y/n/q] > " case "$REPLY" in
                   Y|y) break
                        ;;
                   Q|q) echo "Program sonlandırıldı."
                        exit
                        ;;
                   *)   devam
                        ;;
              esac
         elif [[ -z "$filename" ]]; then devam
              et
         else
              break
         fi
    done
fi

# çıktı html sayfası

if [[ -n "$filename" ]]; then
    if touch "$filename" && [[ -f "$filename" ]]; then
         write_html_page > "$filename"
    else
         echo "$PROGNAME: '$filename' dosyası yazılamıyor" >&2 exit
         1
    fi
else
    write_html_page
fi
```

Henüz işimiz bitmedi. Yapabileceğimiz birkaç şey ve yapabileceğimiz iyileştirmeler daha var.

Daha Fazla Okuma

- *Bash Referans Kılavuzu'nda* özel parametreler hakkında bir makale bulunmaktadır. `$*` ve `$@`: https://www.gnu.org/software/bash/manual/bashref.html#Special-Parameters
-

33 – Akış Kontrolü: `for` ile Döngü

Akış kontrolü ile ilgili bu son bölümde, kabuğun başka bir döngü yapısına bakacağız. *for döngüsü*, bir döngü sırasında dizileri işleme imkanı sağlaması bakımından while ve until döngülerinden farklıdır. Bu, programlama sırasında yararlı olmaktadır. Dolayısıyla, `for` döngüsü `bash` betiklerinde sık kullanılan bir yapıdır.

`for` döngüsü, doğal olarak `for` bileşik komutuyla uygulanır.
`bash`'ta `for` iki şekilde kullanılabilir.

`for`: Geleneksel Kabuk Biçimi

Orijinal `for` komutunun sözdizimi aşağıdaki gibidir:

```
for değişken [kelimelerle]; do
      komutlar
done
```

Burada, *`variable`* döngünün yürütülmesi sırasında artacak bir değişkenin adıdır, *`words değişkene`* sırayla atanacak öğelerin isteğe bağlı bir listesidir ve *`commands`* döngünün her yinelemesinde yürütülecek komutlardır.

`for` komutu komut satırında kullanışlıdır. Nasıl çalıştığını kolayca gösterebiliriz.

```
[me@linuxbox ~]$ for i in A B C D; do echo $i; done A
B
C
D
```

Bu örnekte, dört kelimeden oluşan bir liste verilmiştir: `A`, `B`, `C` ve `D`. Dört kelimelik liste ile döngü dört kez yürütülür. Döngü her yürütüldüğünde, `değişken i'ye` bir kelime atanır. Döngünün içinde, atamayı göstermek için `i'nin` değerini görüntüleyen bir `echo` komutu vardır. `While` ve `until` döngülerinde olduğu gibi, `done` anahtar sözcüğü döngüyü kapatır

döngüyü kapatır.

`for'un` gerçekten güçlü özelliği, kelime listesini oluşturabileceğimiz ilginç yolların sayısıdır. Örneğin, bunu şakül genişletme yoluyla yapabiliriz, şöyle:

```
[me@linuxbox ~]$ for i in {A..D}; do echo $i; done A
B
C
D
```

veya aşağıdaki gibi yol adı genişletme kullanabiliriz:

```
[me@linuxbox ~]$ for i in distros*.txt; do echo "$i"; done
distros-by-date.txt
distros-dates.txt
distros-key-names.txt
distros-key-vernums.txt
distros-names.txt
distros.txt
distros-vernums.txt
distros-versions.txt
```

Yol adı genişletme, döngüde işlenebilecek güzel ve temiz bir yol adı listesi sağlar. Alınması gereken tek önlem, genişletmenin gerçekten bir şeyle eşleştiğini kontrol etmektir. Varsayılan olarak, genişletme herhangi bir dosyayla eşleşmezse, joker karakterler (yukarıdaki örnekte "distros*.txt") döndürülür. Bundan korunmak için, yukarıdaki örneği bir betikte şu şekilde kodlayabiliriz:

```
distros*.txt içindeki i
    için; [[ -e "$i" ]];
    ise
         echo "$i"
    fi
done
```

Dosya varlığı için bir test ekleyerek, başarısız genişletmeyi yok sayacağız.

Kelime üretiminin bir başka yaygın yöntemi de komut ikamesidir.

```
#!/bin/bash
```

```
# en uzun kelime: bir dosyadaki en uzun dizeyi

bul [[ -n "$1" ]]; do
    if [[ -r "$1" ]]; then
         max_word=
         max_len=0
         for i in $(strings "$1"); do
              len="$(echo -n "$i" | wc -c)"
              if (( len > max_len )); then
                   max_len="$len"
                   max_word="$i"
              fi
         done
         echo "$1: '$max_word' ($max_len karakter)"
    fi
    shift
bitti
```

Bu örnekte, bir dosya içinde bulunan en uzun dizeyi arıyoruz. Komut satırında bir veya daha fazla dosya adı verildiğinde, bu program `strings` programını (GNU binutils paketinde bulunur) kullanarak her dosyada okunabilir metin "kelimeler" listesi oluşturur. `for` döngüsü her kelimeyi sırayla işler ve mevcut kelimenin o ana kadar bulunan en uzun kelime olup olmadığını belirler. Döngü sona erdiğinde, en uzun kelime görüntülenir.

Burada dikkat edilmesi gereken bir nokta, normal uygulamamızın aksine, komut ikamesi `$(strings "$1")`'i çift tırnak içine almamamızdır. Bunun nedeni, listemizi elde etmek için kelime bölünmesinin gerçekleşmesini istememizdir. Komut ikamesini tırnak içine alsaydık, dosyadaki her dizgiyi içeren tek bir kelime üretilirdi. Bu, tam olarak aradığımız şey değildir.

for komutunun isteğe bağlı `in` *`words`* kısmı atlanırsa, `for` varsayılan olarak konumsal parametreleri işler. `En uzun kelime` komut dosyasını bu yöntemi kullanacak şekilde değiştireceğiz:

```
#!/bin/bash

# longest-word2: bir dosyadaki en uzun dizgiyi

bul for i; do
    if [[ -r "$i" ]]; then
         max_word=
```

```
        max_len=0
        for j in $(strings "$i"); do
             len="$(echo -n "$j" | wc -c)"
             if (( len > max_len )); then
                  max_len="$len"
                  max_word="$j"
             fi
        done
        echo "$i: '$max_word' ($max_len karakter)"
    fi
tamam
```

Gördüğümüz gibi, en dıştaki döngüyü `while` yerine `for` kullanacak şekilde değiştirdik. For komutundaki kelime listesini çıkararak, bunun yerine konumsal parametreler kullanıldı. Döngünün içinde, `değişken i'nin` önceki örnekleri değişken `j` olarak değiştirildi. `Shift` kullanımı da ortadan kaldırıldı.

Neden i?

Önceki `for` döngüsü örneklerinin her birinde `i` değişkeninin seçildiğini fark etmiş olabilirsiniz. Neden? Aslında gelenek dışında belirli bir nedeni yok. `for` ile kullanılan değişken herhangi bir geçerli değişken olabilir, ancak `i` en yaygın olanıdır, onu `j` ve `k` izler.

Bu geleneğin temeli Fortran programlama dilinden gelmektedir. Fortran'da, I, J, K, L ve M harfleriyle başlayan beyan edilmemiş değişkenler otomatik olarak tamsayı olarak yazılırken, diğer harflerle başlayan değişkenler gerçek sayı olarak yazılır (ondalık kesirli sayılar). Bu davranış, programcıların döngü değişkenleri için I, J ve K değişkenlerini kullanmalarına neden olmuştur, çünkü geçici bir değişken (döngü değişkenleri genellikle öyledir) gerektiğinde bunları kullanmak daha az iş gerektiriyordu.

Bu durum, Fortran tabanlı şu esprili sözün ortaya çıkmasına da neden oldu: "GOD gerçektir, tamsayı olarak beyan edilmedikçe."

`for`: C Dili Formu

`Bash'ın` son sürümlerinde, C programlama dilinde bulunan biçime benzeyen ikinci bir `for` komut sözdizimi biçimi eklenmiştir. Diğer birçok dil de bu biçimi desteklemektedir.

```
for (( ifade1; ifade2; ifade3 )); do
      komutlar
done
```

Burada *ifade1*, *ifade2* ve *ifade3* aritmetik ifadelerdir ve *komutlar* döngünün her yinelemesi sırasında gerçekleştirilecek komutlardır.

Davranış açısından, bu biçim aşağıdaki yapıya eşdeğerdir:

```
(( ifade1 ))
while (( ifade2 )); yap
      komutlar
      (( ifade3 ))
bitti
```

ifade1 döngü koşullarını başlatmak için kullanılır, *ifade2* döngünün ne zaman biteceğini belirlemek için kullanılır ve *ifade3* döngünün her yinelemesinin sonunda gerçekleştirilir.

İşte tipik bir uygulama:

```
#!/bin/bash

# simple_counter: C tarzı komut demosu for ((

i=0; i<5; i=i+1 )); do
    echo $i
tamam
```

Çalıştırıldığında, aşağıdaki çıktıyı verir:

```
[me@linuxbox ~]$ basit_sayaç 0
1
2
3
4
```

Bu örnekte, *ifade1* değişken `i'yi` sıfır değeriyle başlatır, *ifade2* `i'nin` değeri 5'ten küçük olduğu sürece döngünün devam etmesini sağlar ve *ifade3* döngü her tekrarlandığında `i'nin` değerini 1 artırır.

C dilindeki `for` yapısı, sayısal bir dizi gerektiğinde her zaman kullanışlıdır. Önümüzdeki iki bölümde bunun çeşitli uygulamalarını göreceğiz.

Özet

`for` komutu hakkındaki bilgilerimizle, şimdi `sys_info_page` betiğimize son iyileştirmeleri uygulayacağız. Şu anda, `report_home_space` işlevi şöyle görünüyor:

```
report_home_space () {
    if [[ "$(id -u)" -eq 0 ]]; then
         cat << _EOF_
    <h2>Ana Sayfa Alanı Kullanımı (Tüm Kullanıcılar)</h2>
    <pre>$(du -sh /home/*)</pre>
_EOF_
    else
         cat << _EOF_
    <h2>Ana Sayfa Alanı Kullanımı ($USER)</h2>
    <pre>$(du -sh "$HOME")</pre>
_EOF_
    fi
    dönüş
}
```

Ardından, her kullanıcının ana dizini hakkında daha fazla ayrıntı sağlamak ve her birindeki toplam dosya ve alt dizin sayısını eklemek için bunu yeniden yazacağız.

```
report_home_space () {

    local format="%8s%10s%10s\n"
    yerel i dir_list toplam_dosya toplam_dizin toplam_boyut kullanıcı_adı

    if [[ "$(id -u)" -eq 0 ]]; then
         dir_list=/home/*
         user_name="Tüm Kullanıcılar"
    else
         dir_list="$HOME"
         kullanıcı_adı="$USER"
    fi

    echo "<h2>Ana Sayfa Alanı Kullanımı

    ($kullanıcı_adı)</h2>" for i in $dir_list; do

         toplam_dosya="$(find "$i" -type f | wc -l)"
```

```
        toplam_dizin="$(find "$i" -type d | wc -l)"
        toplam_boyut="$(du -sh "$i" | cut -f 1)"

        echo "<H3>$i</H3>"
        echo "<pre>"
        printf "$format" "Dizinler" "Dosyalar"
        "Boyut" printf "$format" "----" "-----"
        "------------------------------------------------------------"
        printf "$format" "$total_dirs" "$total_files" "$total_size" echo
        "</pre>"
    done
    return
}
```

Bu yeniden yazım, şimdiye kadar öğrendiklerimizin çoğunu uygulamaktadır. Hala süper kullanıcıyı test ediyoruz, ancak `if` ifadesinin bir parçası olarak tüm eylemleri gerçekleştirmek yerine, daha sonra for döngüsünde kullanılacak bazı değişkenleri ayarlıyoruz. İşleve birkaç yerel değişken ekledik ve bazı çıktıları biçimlendirmek için `printf işlevini` kullandık.

Daha Fazla Okuma

- *Advanced Bash-Scripting Guide* kitabında, `for` kullanılarak çeşitli örneklerin verildiği döngülerle ilgili bir bölüm bulunmaktadır: https://tldp.org/LDP/abs/html/loops1.html
- *Bash Referans Kılavuzu*, `for` dahil olmak üzere döngüsel bileşik komutları açıklamaktadır: https://www.gnu.org/software/bash/manual/bashref.html#Looping-Constructs

34 – Diziler ve Sayılar

Bilgisayar programları tamamen verilerle çalışmakla ilgilidir. Önceki bölümlerde, dosya düzeyinde veri işleme üzerinde durduk. Ancak, birçok programlama problemi, dizeler ve sayılar gibi daha küçük veri birimleri kullanılarak çözülmelidir.

Bu bölümde, dizeleri ve sayıları işlemek için kullanılan birkaç kabuk özelliğine bakacağız. Kabuk, dize işlemlerini gerçekleştiren çeşitli parametre genişletmeleri sağlar. Aritmetik genişletmeye (7. bölümde değindiğimiz) ek olarak, daha üst düzey matematik işlemleri gerçekleştiren `bc` adlı iyi bilinen bir komut satırı programı vardır.

Parametre Genişletme

Parametre genişletme 7. bölümde ele alınmış olsa da, çoğu parametre genişletmesi komut satırında değil, komut dosyalarında kullanıldığından bu konuyu ayrıntılı olarak ele almadık. Bazı parametre genişletme biçimleriyle, örneğin kabuk değişkenleriyle, daha önce çalıştık. Kabuk, daha birçok parametre genişletme biçimi sunar.

Not: İstenmeyen kelime bölünmelerini önlemek için, parametre genişletmelerini çift tırnak içine almak her zaman iyi bir uygulamadır, bunu yapmamak için özel bir neden olmadığı sürece. Bu, dosya adlarıyla çalışırken özellikle geçerlidir, çünkü dosya adları genellikle gömülü boşluklar ve diğer çeşitli sorunlar içerebilir.

Temel Parametreler

Parametre genişletmenin en basit şekli, değişkenlerin normal kullanımında görülür. İşte bir örnek:

```
$a
```

Genişletildiğinde, bu değişken `a'nın` içerdiği her neyse o olur. Basit parametreler ayrıca köşeli parantezlerle de çevrelenebilir.

```
${a}
```

Bu, genişletme üzerinde herhangi bir etkisi yoktur, ancak değişken diğer metinlere bitişikse ve bu durum kabuğu karıştırabilirse gereklidir. Bu örnekte, _file dizesini değişken a'nın

içeriğine ekleyerek bir dosya adı oluşturmaya çalışıyoruz.

`_file` dizesini değişken `a`'nın içeriğine ekleyerek bir dosya adı oluşturmaya çalışıyoruz.

```
[me@linuxbox ~]$ a="foo"
[me@linuxbox ~]$ echo "$a_file"
```

Bu komut dizisini uygularsak, kabuk `a` yerine `a_file` adlı değişkeni genişletmeye çalışacağı için sonuç hiçbir şey olmayacaktır. Bu sorun, "gerçek" değişken adının etrafına parantez ekleyerek çözülebilir.

```
[me@linuxbox ~]$ echo "${a}_file" foo_file
```

Dokuzdan büyük konumsal parametrelere, sayıyı parantez içine alarak erişilebileceğini de gördük. Örneğin, 11. konumsal parametreye erişmek için şunu yapabiliriz:

```
${11}
```

Boş Değişkenleri Yönetmek için Genişletmeler

Birkaç parametre genişletmesi, mevcut olmayan ve boş değişkenleri ele almak için tasarlanmıştır. Bu genişletmeler, eksik konumsal parametreleri işlemek ve parametrelere varsayılan değerler atamak için kullanışlıdır. İlki, ayarlanmamış bir parametre için varsayılan bir değer sağlar.

`${`*`parameter:-word`*`}`

Parametre ayarlanmamışsa (yani mevcut değilse) veya boşsa, bu genişletme *word* değerini verir. *Parametre* boş değilse, genişletme *parameter* değerini verir.

```
[me@linuxbox ~]$ foo=
[me@linuxbox ~]$ echo ${foo:-"ayarlanmamışsa ikame değeri"}
ayarlanmamışsa ikame değeri
[me@linuxbox ~]$ echo $foo

[me@linuxbox ~]$ foo=bar
[me@linuxbox ~]$ echo ${foo:-"ayarlanmamışsa ikame değeri"} bar
[me@linuxbox ~]$ echo $foo
bar
```

Bu genişletme, bir parametre ayarlanmamışsa veya boşsa varsayılan bir değer sağlar.

`${parameter:=word}`

Parametre ayarlanmamışsa veya boşsa, bu genişletme *kelimenin* değerini verir. Ayrıca, *kelimenin* değeri *parametreye* atanır. *Parametre* boş değilse, genişletme *parametrenin* değerini verir.

```
[me@linuxbox ~]$ foo=
[me@linuxbox ~]$ echo ${foo:="ayarlanmamışsa varsayılan
değer"} ayarlanmamışsa varsayılan değer
[me@linuxbox ~]$ echo $foo
ayarlanmamışsa varsayılan değer
[me@linuxbox ~]$ foo=bar
[me@linuxbox ~]$ echo ${foo:="ayarlanmamışsa varsayılan
değer"} bar
[me@linuxbox ~]$ echo $foo
bar
```

Not: Konumsal ve diğer özel parametreler bu şekilde atanamaz.

Bu genişletme, gerekli bir parametre ayarlanmamışsa veya boşsa komut dosyasının sonlandırılmasına neden olur.

`${parameter:?word}`

Parametre ayarlanmamışsa veya boşsa, bu genişletme komut dosyasının bir hata ile sonlanmasına neden olur ve *kelimenin* içeriği standart hataya gönderilir. *Parametre* boş değilse, genişletme *parametrenin* değerini verir.

```
[me@linuxbox ~]$ foo=
[me@linuxbox ~]$ echo ${foo:?"parameter is empty"}
bash: foo: parameter is empty
[me@linuxbox ~]$ echo $?
1
[me@linuxbox ~]$ foo=bar
[me@linuxbox ~]$ echo ${foo:?"parametre boş"} bar
[me@linuxbox ~]$ echo $?
0
```

Bu genişletme, boş olmayan bir parametre için bir değer ikame eder.

`${parameter:+word}`

parametre ayarlanmamışsa veya boşsa, genişletme hiçbir sonuç vermez. *parametre* boş değilse, *kelimenin* değeri *parametre* ile değiştirilir; ancak *parametrenin* değeri değiştirilmez.

```
[me@linuxbox ~]$ foo=
[me@linuxbox ~]$ echo ${foo:+"ayarlanmışsa değerle değiştir"}

[me@linuxbox ~]$ foo=bar
[me@linuxbox ~]$ echo ${foo:+"ayarlanmışsa değer ikame et"}
ayarlanmışsa değer ikame et
```

Değişken Adlarını Döndüren Genişletmeler

Kabuk, değişkenlerin adlarını döndürme yeteneğine sahiptir. Bu, oldukça egzotik durumlarda kullanılır.

```
${!prefix*}
${!prefix@}
```

Bu genişletme, *prefix* ile başlayan mevcut değişkenlerin adlarını döndürür. `Bash` belgelerine göre, bu genişletmenin her iki biçimi de tırnak içine alınmadığında aynı şekilde çalışır. Burada, BASH ile başlayan adlara sahip ortamdaki tüm değişkenleri listeliyoruz:

```
[me@linuxbox ~]$ echo ${!BASH*}
BASH BASH_ARGC BASH_ARGV BASH_COMMAND BASH_COMPLETION
BASH_COMPLETION_DIR BASH_LINENO BASH_SOURCE BASH_SUBSHELL
BASH_VERSINFO BASH_VERSION
```

Dize İşlemleri

Dizgiler üzerinde işlem yapan çok sayıda genişletme vardır. Bu genişletmelerin çoğu, yol adları üzerinde işlem yapmak için özellikle uygundur. Aşağıdaki genişletme, *parametre* tarafından içerilen dizginin uzunluğuna genişler:

`${#parameter}`

Normalde, *parametre* bir dizidir; ancak, *parametre* `@` veya `*` ise, genişletme konum parametrelerinin sayısıyla sonuçlanır.

```
[me@linuxbox ~]$ foo="Bu dize uzundur."
[me@linuxbox ~]$ echo "'$foo' ${#foo} karakter uzunluğundadır."
'Bu dize uzundur.' 20 karakter uzunluğundadır.
```

`${parameter:offset}`
`${parameter:offset:length}`

Bu genişletmeler, *parametrede* bulunan dizenin bir bölümünü çıkarmak için kullanılır. Çıkarma işlemi, *uzunluk* belirtilmedikçe, dizenin başından *offset* karakterinden başlar ve dizenin sonuna kadar devam eder.

```
[me@linuxbox ~]$ foo="Bu dize uzundur."
[me@linuxbox ~]$ echo ${foo:5}
dizesi uzundur.
[me@linuxbox ~]$ echo ${foo:5:6}
string
```

Ofset değeri negatifse, dizgenin başından değil sonundan başladığı anlamına gelir. Negatif değerlerin önüne, `${parameter:-word}` genişletmesiyle karışıklığı önlemek için bir boşluk konulması gerektiğini unutmayın. *Uzunluk*, varsa, sıfırdan küçük olmamalıdır.

parameter `@` ise, genişlemenin sonucu *offset'ten* başlayan *length* konumsal parametreler olur.

```
[me@linuxbox ~]$ foo="Bu dize uzundur."
[me@linuxbox ~]$ echo ${foo: -5}
uzun.
[me@linuxbox ~]$ echo ${foo: -5:2}
lo
```

`${parameter#pattern}`
`${parameter##pattern}`

Bu genişletmeler, *pattern* ile tanımlanan *parametrede* bulunan dizgenin başındaki kısmı kaldırır. *pattern*, yol adı genişletmede kullanılanlar gibi bir joker karakter desenidir. İki biçimin arasındaki fark, `#` biçiminin en kısa eşleşmeyi kaldırması, `##` biçiminin ise en uzun eşleşmeyi kaldırmasıdır.

```
[me@linuxbox ~]$ foo=file.txt.zip
```

harici programların kullanımını ortadan kaldırarak komut dosyalarının verimliliğini artırır. Örnek olarak, önceki bölümde ele alınan `en uzun kelime` programını, komut ikamesi `$(echo - n $j | wc -c)` ve bunun sonucunda ortaya çıkan alt kabuk yerine `${#j}` parametre genişletmesini kullanacak şekilde şu şekilde değiştireceğiz:

```
#!/bin/bash

# en uzun kelime3: bir dosyadaki en uzun dizgiyi

bul i için; yap
    if [[ -r "$i" ]]; then
         max_word=
         max_len=0
         j için $(strings $i); yap
              len="${#j}"
              if (( len > max_len )); then
                   max_len="$len"
                   max_word="$j"
              fi
         done
         echo "$i: '$max_word' ($max_len karakter)"
    fi
tamam
```

Ardından, `time` komutunu kullanarak iki sürümün verimliliğini karşılaştıracağız.

```
[me@linuxbox ~]$ time longest-word2 dirlist-usr-bin.txt
dirlist-usr-bin.txt: 'scrollkeeper-get-extended-content-list' (38
karakter)

gerç      0m3.618s
ek        0m1.544s
sistem 0m1.768s
[me@linuxbox ~]$ time longest-word3 dirlist-usr-bin.txt
dirlist-usr-bin.txt: 'scrollkeeper-get-extended-content-list' (38
karakter)

gerç      0m0.060s
ek        0m0.056s
sistem    0m0.008s
```

Komut dosyasının orijinal sürümü metin dosyasını taramak için 3,618 saniye sürerken, parametre genişletme kullanan yeni sürüm sadece 0,06 saniye sürüyor — bu önemli bir gelişme.

Büyük/Küçük Harf Dönüştürme

`bash`, dizelerin büyük/küçük harf dönüşümünü desteklemek için dört parametre genişletmesi ve iki `declare` komut seçeneğine sahiptir.

Peki, büyük/küçük harf dönüşümü ne işe yarar? Estetik değerinin yanı sıra, programlamada da önemli bir rolü vardır. Bir veritabanı aramasını ele alalım. Bir kullanıcının, veritabanında aramak istediğimiz bir veri giriş alanına bir dize girdiğini düşünelim. Kullanıcı, değeri tamamen büyük harflerle, tamamen küçük harflerle veya her ikisinin bir kombinasyonu ile girebilir. Veritabanımızı büyük ve küçük harflerin tüm olası permütasyonlarıyla doldurmak istemeyiz. Ne yapmalıyız?

Bu soruna yaygın bir yaklaşım, kullanıcının girdisini *normalleştirmektir*. Yani, veritabanı aramasını denemeden önce onu standart bir biçime dönüştürmek. Bunu, kullanıcının girdisindeki tüm karakterleri küçük veya büyük harfe dönüştürerek ve veritabanı girişlerinin de aynı şekilde normalleştirilmesini sağlayarak yapabiliriz.

`Declare` komutu, dizeleri büyük veya küçük harfe normalleştirmek için kullanılabilir. `Declare` kullanarak, bir değişkene ne atanırsa atansın, her zaman istenen biçimi içermesini sağlayabiliriz.

```
#!/bin/bash

# ul-declare: declare ile büyük/küçük harf dönüşümünü

göster declare -u upper
declare -l lower

if [[ $1 ]]; then
        upper="$1"
        lower="$1"
        echo "$upper"
        echo "$lower"
fi
```

Yukarıdaki komut dosyasında, `declare komutunu` kullanarak `upper` ve `lower` adlı iki değişken oluşturuyoruz. İlk komut satırı argümanının (konumsal parametre 1) değerini her bir değişkene atıyoruz ve ardından bunları ekranda görüntülüyoruz.

```
[me@linuxbox ~]$ ul-declare aBc
ABC
abc
```

Gördüğümüz gibi, komut satırı argümanı (`aBc`) normalleştirilmiştir.

`Declare'e` ek olarak, Tablo 34-1'de açıklandığı gibi büyük/küçük harf dönüşümü gerçekleştiren dört parametre genişletmesi vardır.

Tablo 34-1: Büyük/Küçük Harf Dönüştürme Parametre Genişletmeleri

Biçim	Sonuç
`${`*`parameter`*`,,`*`pattern`*`}`	*Parametrenin* değerini tamamen küçük harfe genişletir. *`pattern`*, dönüştürülecek karakterleri sınırlayacak isteğe bağlı bir kabuk desenidir (örneğin, `[A-F]`). Desenlerin tam açıklaması için `bash` man sayfasına bakın.
`${`*`parametre`*`,`*`desen`*`}`	*Parametrenin* değerini genişletir, yalnızca ilk karakteri küçük harfe dönüştürür.
`${`*`parameter^^pattern`*`}`	*Parametrenin* değerini tümü büyük harflerle genişletin.
`${`*`parameter^pattern`*`}`	*Parametrenin* değerini genişletin, yalnızca ilk karakteri büyük harfe çevirin (büyük harf kullanımı).

İşte bu genişletmeleri gösteren bir komut dosyası:

```
#!/bin/bash

# ul-param: parametre genişletme yoluyla büyük/küçük harf dönüşümünü

gösterir if [[ "$1" ]]; then
        echo "${1,,}"
        echo "${1,}"
        echo "${1^^}"
        echo "${1^}"
fi
```

İşte komut dosyasının çalışması:

```
[me@linuxbox ~]$ ul-param aBc abc
aBc
ABC
ABc
```

Yine, ilk komut satırı argümanını işliyor ve parametre genişletmelerinin desteklediği dört varyasyonu çıktı olarak veriyoruz. Bu betik ilk konumsal parametreyi kullanıyor olsa da, *parametre* herhangi bir dize, değişken veya dize ifadesi olabilir.

Aritmetik Değerlendirme ve Genişletme

Aritmetik genişletmeyi 7. Bölümde incelemiştik. Tamsayılar üzerinde çeşitli aritmetik işlemler gerçekleştirmek için kullanılır. Temel biçimi aşağıdaki gibidir; burada *ifade* geçerli bir aritmetik ifadedir:

`$((`*ifade*`))`

Bu, Bölüm 27'de karşılaştığımız aritmetik değerlendirme (doğruluk testleri) için kullanılan bileşik komut `(( ))` ile ilgilidir.

Önceki bölümlerde, yaygın ifade ve operatör türlerinden bazılarını gördük. Burada, daha eksiksiz bir listeye bakacağız.

Sayı Tabanları

9. bölümde sekizlik (8 tabanlı) ve onaltılık (16 tabanlı) sayıları inceledik. Aritmetik ifadelerde, kabuk herhangi bir tabandaki tamsayı sabitlerini destekler. Tablo 34-2, tabanları belirtmek için kullanılan gösterimleri listeler.

Tablo 34-2: Farklı Sayı Tabanlarını Belirtme

Notasyon	Açıklama
sayı	Varsayılan olarak, herhangi bir gösterimi olmayan sayılar ondalık (10 tabanlı) tamsayılar olarak kabul edilir.
0sayı	Aritmetik ifadelerde, başında sıfır bulunan sayılar sekizlik olarak kabul edilir.
0xsayı	Onaltılık gösterim.
taban#sayı	*s*ayı t*aban*ında

İşte bazı örnekler:

```
[me@linuxbox ~]$ echo $((0xff))
255
[me@linuxbox ~]$ echo $((2#11111111))
255
```

Önceki örneklerde, onaltılık sayı `ff`'nin (en büyük iki basamaklı sayı) ve en büyük sekiz basamaklı ikili (2 tabanlı) sayının değerini yazdırıyoruz.

Tekli İşlemciler

Bir sayının pozitif mi negatif mi olduğunu belirtmek için kullanılan `+` ve `-` olmak üzere iki tekli işleç vardır. Bir örnek olarak `-5` verilebilir.

Basit Aritmetik

Sıradan aritmetik operatörler Tablo 34-3'te listelenmiştir.

Tablo 34-3: Aritmetik İşlemciler

İşleç	Açıklama
+	Toplama
-	Çıkarma
*	Çarpma
/	Tamsayı bölme
**	Üs alma
%	Modulo (kalan)

Bunların çoğu kendinden anlaşılırdır, ancak tamsayı bölme ve modül daha ayrıntılı bir tartışma gerektirir.

Kabuğun aritmetik işlemleri yalnızca tamsayılar üzerinde çalıştığı için, bölme sonuçları her zaman tam sayılardır.

```
[me@linuxbox ~]$ echo $(( 5 / 2 ))
2
```

Bu, bölme işleminde kalanın belirlenmesini daha önemli hale getirir.

```
[me@linuxbox ~]$ echo $(( 5 % 2 )) 1
```

Bölme ve modulo operatörlerini kullanarak, 5'in 2'ye bölünmesinin sonucu 2 ve kalanın 1 olduğunu belirleyebiliriz.

Kalanın hesaplanması döngülerde kullanışlıdır. Döngünün yürütülmesi sırasında belirli aralıklarla bir işlemin gerçekleştirilmesini sağlar. Aşağıdaki örnekte, bir satırdaki sayıları göstererek 5'in katlarını vurgularız:

```
#!/bin/bash

# modulo: modulo operatörünü göster ((i = 0;

i <= 20; i = i + 1)); do
    remainder=$((i % 5))
    if (( kalan == 0 )); then printf
         "<%d> " "$i"
    else
         printf "%d " "$i"
    fi
bitti
printf "\n"
```

Çalıştırıldığında, sonuçlar şöyle görünür:

```
[me@linuxbox ~]$ modulo
<0> 1 2 3 4 <5> 6 7 8 9 <10> 11 12 13 14 <15> 16 17 18 19 <20>
```

Atama

Kullanımları hemen anlaşılmasa da, aritmetik ifadeler atama işlevi görebilir. Farklı bir bağlamda olsa da, atama işlemini birçok kez gerçekleştirdik. Bir değişkene değer verdiğimiz her seferinde, atama işlemi gerçekleştirmiş oluyoruz. Bunu aritmetik ifadeler içinde de yapabiliriz.

```
[me@linuxbox ~]$ foo= [me@linuxbox
~]$ echo $foo
```

```
[me@linuxbox ~]$ if (( foo = 5 )); then echo "Doğru."; fi Doğru.
[me@linuxbox ~]$ echo $foo
5
```

Önceki örnekte, önce `foo` değişkenine boş bir değer atadık ve gerçekten boş olduğunu doğruladık. Ardından, bileşik komutla bir `if` işlemi gerçekleştirdik `(( foo = 5 ))` ifadesini kullanarak bir if ifadesi gerçekleştiriyoruz. Bu işlem iki ilginç şey yapıyor: `foo` değişkenine 5 değerini atıyor ve `foo'ya` sıfır olmayan bir değer atandığı için true sonucunu veriyor.

Not: Önceki ifadede `=` işaretinin tam anlamını hatırlamak önemlidir. Tek bir `=` atama işlevi görür. `foo = 5`, "`foo'yu` 5'e eşit yap" anlamına gelirken, `==` eşdeğerliği değerlendirir. `foo == 5`, "`foo` 5'e eşit mi?" anlamına gelir. Bu, birçok programlama dilinde yaygın bir özelliktir. Kabukta bu biraz kafa karıştırıcı olabilir, çünkü `test` komutu dize eşdeğerliği için tek bir = kabul eder. Bu, `test` yerine daha modern `[[ ]]` ve `(( ))` bileşik komutlarını kullanmak için bir başka nedendir.

= notasyonuna ek olarak, kabuk ayrıca Tablo 34-4'te gösterildiği gibi bazı yararlı atamaları gerçekleştiren notasyonlar da sağlar.

Tablo 34-4: Atama İşlemcileri

Notasyon	Açıklama
parametre = değer	Basit atama. *Parametreye değer* atar.
parametre += değer	Toplama. *parametre* = *parametre* + *değer* ile eşdeğerdir.
parametre -= değer	Çıkarma. *parametre* = *parametre* – *değer* ile eşdeğerdir.
*parametre *= değer*	Çarpma. *parametre* = *parametre* * *değer*.
parametre /= değer	Tamsayı bölme. *parametre* = *parametre* / *değer ile eşdeğerdir.*
parametre %= değer	Modulo. *parametre* = *parametre* % *değer* ile eşdeğerdir.
parametre++	Değişken son artış. *parametre* = *parametre* + 1 ile eşdeğerdir (ancak, aşağıdaki

	tartışmaya bakın).
parametre--	Değişken sonrasındaki azalma. *parametre* = *parametre* − 1.
++*parametre*	Değişken ön artış. *Parametre* = *parametre* + 1 ile eşdeğerdir.
--*parametre*	Değişken ön azaltma. *parametre* = *parametre* − 1'e eşdeğerdir.

Bu atama operatörleri, birçok yaygın aritmetik görev için kullanışlı bir kısaltma sağlar. Özellikle ilgi çekici olan, parametrelerinin değerini bir artırıp azaltan artış (++) ve azalma (--) operatörleridir. Bu notasyon stili C programlama dilinden alınmıştır ve `bash` dahil olmak üzere bir dizi başka programlama diline de dahil edilmiştir.

Operatörler, parametrenin başında veya sonunda görünebilir. Her ikisi de parametreyi bir artırır veya azaltır, ancak iki konum arasında ince bir fark vardır. Parametrenin başına yerleştirilirse, parametre döndürülmeden önce parametre artırılır (veya azaltılır). Sonra yerleştirilirse, işlem parametre döndürüldükten *sonra* gerçekleştirilir. Bu oldukça garip, ancak amaçlanan davranıştır. İşte bir örnek:

```
[me@linuxbox ~]$ foo=1
[me@linuxbox ~]$ echo $((foo++)) 1
[me@linuxbox ~]$ echo $foo
2
```

`foo` değişkenine bir değer atarsak ve ardından parametre adının arkasına ++ operatörünü ekleyerek onu artırırsak, `foo` bir değerle geri döner. Ancak, değişkenin değerine ikinci kez baktığımızda, artırılmış değeri görürüz. ++ operatörünü parametrenin önüne koyarsak, daha beklenen davranışı elde ederiz.

```
[me@linuxbox ~]$ foo=1
[me@linuxbox ~]$ echo $((++foo)) 2
[me@linuxbox ~]$ echo $foo
2
```

Çoğu kabuk uygulaması için, operatöre önek eklemek en yararlı yöntemdir.

++ ve -- operatörleri genellikle döngülerle birlikte kullanılır. Modulo betiğimizi biraz daha sıkılaştırmak için bazı iyileştirmeler yapacağız.

```
#!/bin/bash

# modulo2: modulo operatörünü göster ((i = 0;

i <= 20; ++i )); yap
    if (((i % 5) == 0 )); then printf
         "<%d> " "$i"
    else
         printf "%d " "$i"
    fi
done
printf "\n"
```

Bit İşlemleri

Bir sınıf operatörler, sayıları alışılmadık bir şekilde işler. Bu operatörler bit düzeyinde çalışır. Genellikle bit bayraklarının ayarlanması veya okunmasıyla ilgili belirli türden düşük düzeyli görevler için kullanılırlar. Bit operatörleri Tablo 34-5'te listelenmiştir.

Tablo 34-5: Bit Operatörleri

Operatör	Açıklama
~	Bitwise negation. Bir sayıdaki tüm bitleri tersine çevirir.
<<	Sola bitwise kaydırma. Bir sayıdaki tüm bitleri sola kaydırır.
>>	Sağ bit kaydırma. Bir sayıdaki tüm bitleri sağa kaydırır.
&	Bitwise AND. İki sayıdaki tüm bitler üzerinde AND işlemi gerçekleştirir.
|	Bitwise OR. İki sayının tüm bitleri üzerinde OR işlemi gerçekleştirir.
^	Bitwise XOR. İki sayının tüm bitleri üzerinde özel OR işlemi gerçekleştirir.

Bitwise negation hariç, tüm işlemler için karşılık gelen atama operatörleri (örneğin, <<=) olduğunu unutmayın.

Burada, sol bitwise kaydırma operatörünü kullanarak 2'nin kuvvetlerinin bir listesini oluşturmayı göstereceğiz:

```
[me@linuxbox ~]$ for ((i=0;i<8;++i)); do echo $((1<<i)); done 1
2
4
8
16
32
64
128
```

Mantık

Bölüm 27'de gördüğümüz gibi, `(( ))` bileşik komutu çeşitli karşılaştırma operatörlerini destekler. Mantığı değerlendirmek için kullanılabilecek birkaç operatör daha vardır. Tablo 34-6 bunların tam listesini vermektedir.

Tablo 34-6: Karşılaştırma İşlemcileri

İşleç	Açıklama
<=	Küçük veya eşit.
>=	Büyük veya eşit.
<	Küçük.
>	Büyük.
==	Eşit.
!=	Eşit değil.
&&	Mantıksal VE.
||	Mantıksal VEYA.
expr1?*expr2*:*expr3*	Karşılaştırma (üçlü) işleci. İfade *expr1* sıfırdan farklı (aritmetik olarak doğru) olarak değerlendirilirse, *expr2;* aksi takdirde *expr3*.

Mantıksal işlemlerde kullanıldığında, ifadeler aritmetik mantık kurallarına uyar; yani, sıfır olarak değerlendirilen ifadeler yanlış, sıfır olmayan ifadeler ise

doğru kabul edilir. `(( ))` bileşik komutu, sonuçları kabuğun normal çıkış kodlarına eşler.

```
[me@linuxbox ~]$ if ((1)); then echo "true"; else echo "false"; fi
true
[me@linuxbox ~]$ if ((0)); then echo "true"; else echo "false"; fi
false
```

Mantıksal operatörlerin en garibi *üçlü operatördür*. Bu operatör (C programlama dilindeki operatörü örnek alarak modellenmiştir) bağımsız bir mantıksal test gerçekleştirir. Bir tür `if/then/else` ifadesi olarak kullanılabilir. Üç aritmetik ifade üzerinde çalışır (diziler çalışmaz) ve ilk ifade doğruysa (veya sıfırdan farklıysa) ikinci ifade gerçekleştirilir. Aksi takdirde, üçüncü ifade gerçekleştirilir. Bunu komut satırında deneyebiliriz:

```
[me@linuxbox ~]$ a=0 [me@linuxbox
~]$ ((a<1?++a:--a)) [me@linuxbox
~]$ echo $a
1
[me@linuxbox ~]$ ((a<1?++a:--a))
[me@linuxbox ~]$ echo $a
0
```

Burada üçlü operatörün nasıl çalıştığını görüyoruz. Bu örnek bir geçiş işlevi gerçekleştirir. Operatör her gerçekleştirildiğinde, `a` değişkeninin değeri sıfırdan bire veya tam tersi şekilde değişir.

İfadeler içinde atama yapmanın basit bir işlem olmadığını lütfen unutmayın. Denendiğinde, `bash` bir hata bildirir.

```
[me@linuxbox ~]$ a=0
[me@linuxbox ~]$ ((a<1?a+=1:a-=1))
bash: ((: a<1?a+=1:a-=1: değişken olmayan bir değişkene atama denemesi
(hata belirteci "-=1")
```

Bu sorun, atama ifadesini parantezlerle çevreleyerek azaltılabilir.

```
[me@linuxbox ~]$ ((a<1?(a+=1):(a-=1)))
```

Aşağıda, basit bir sayı tablosu üreten bir komut dosyasında aritmetik operatörlerin kullanıldığı daha kapsamlı bir örnek verilmiştir.

```
#!/bin/bash

# arith-loop: aritmetik operatörleri gösteren komut dosyası

finished=0
a=0
printf "a\ta**2\ta**3\n"
printf "=\t====\t====\n"

until ((finished)); do
    b=$((a**2))
    c=$((a**3))
    printf "%d\t%d\t%d\n" "$a" "$b" "$c"
    ((a<10?++a:(bitti=1)))
bitti
```

Bu betikte, `finished` değişkeninin değerine dayalı bir until döngüsü uyguluyoruz. Başlangıçta, değişken sıfıra (aritmetik olarak yanlış) ayarlanır ve sıfırdan farklı bir değer alana kadar döngüye devam ederiz. Döngü içinde, sayaç değişkeninin karesini ve küpünü hesaplarız
`a`'nın karesini ve küpünü hesaplıyoruz. Döngünün sonunda, sayaç değişkeninin değeri değerlendirilir. Eğer 10'dan (maksimum yineleme sayısı) küçükse, bir artırılır, aksi takdirde `finished` değişkenine bir değeri verilir, böylece `finished` aritmetik olarak doğru hale gelir ve döngü sonlandırılır. Komut dosyasını çalıştırdığımızda şu sonuç elde edilir:

```
[me@linuxbox ~]$ arith-loop
a   a**2 a**3
=   ==== ====
0   0    0
1   1    1
2   4    8
3   9    27
4   16   64
5   25   125
6   36   216
7   49   343
8   64   512
9   81   729
10  100  1000
```

Virgül İşleci

İncelenecek son operatör virgül operatörüdür. Bu operatör, değerlendirme içinde birden fazla ifadeyi şu şekilde yerleştirmemizi sağlar: `(( expr, expr, expr ))`. Birden fazla işlemi tek bir değerlendirmede birleştirmek istediğimiz durumlarda bunu kullanabiliriz. Bunu göstermek için, `aritmetik döngü` komut dosyasını virgül operatörünü kullanarak basitleştirmek üzere değiştireceğiz.

```
#!/bin/bash

# arith-loop2: virgül aritmetik operatörünü gösteren komut dosyası

a=0
printf "a\ta**2\ta**3\n"
printf "=\t====\t====\n"

until (( a++, b=a**2, c=a**3, a>10 )); do printf
    "%d\t%d\t%d\n" $a $b $c
done
```

Burada tüm hesaplamaları tek bir satıra sığdırmayı başardık. Değerlendirmedeki son ifadenin değerlendirmenin çıkış durumunu belirlediğini unutmayın.

`bc` – Keyfi Hassasiyetli Hesap Makinesi Dili

Kabuğun tamsayı aritmetiğini nasıl işleyebildiğini gördük, peki ya daha yüksek matematik işlemleri yapmamız veya sadece kayan noktalı sayılar kullanmamız gerekirse ne yapacağız? Cevap, yapamayız. En azından kabukla doğrudan yapamayız. Bunu yapmak için harici bir program kullanmamız gerekir. Bunun için birkaç yaklaşım vardır. Perl veya AWK programlarını gömmek olası bir çözümdür, ancak ne yazık ki bu kitabın kapsamı dışındadır.

Başka bir yaklaşım ise özel bir hesap makinesi programı kullanmaktır. Birçok Linux sisteminde bulunan bu tür bir program `bc` olarak adlandırılır.

`bc` programı, kendi C benzeri dilinde yazılmış bir dosyayı okur ve yürütür. Bir `bc` betiği ayrı bir dosya olabilir veya standart girişten okunabilir. `bc` dili, değişkenler, döngüler ve programcı tarafından tanımlanan işlevler dahil olmak üzere pek çok özelliği destekler. Burada `bc'yi` tamamen ele almayacağız, sadece bir fikir edinmek için yeterli kadarını anlatacağız. `bc`, man sayfasında ayrıntılı olarak açıklanmıştır.

Basit bir örnekle başlayalım. 2 artı 2'yi eklemek için bir `bc` komut dosyası yazacağız.

```
/* Çok basit bir bc komut dosyası */

2 + 2
```

Komut dosyasının ilk satırı bir yorumdur. bc, yorumlar için C programlama diliyle aynı sözdizimini kullanır. Birden fazla satıra yayılabilen yorumlar /* ile başlar ve */ ile biter.

bc kullanımı

Önceki bc betiğini foo.bc olarak kaydedersek, şu şekilde çalıştırabiliriz:

```
[me@linuxbox ~]$ bc foo.bc
bc 1.06.94
Telif hakkı 1991-1994, 1997, 1998, 2000, 2004, 2006 Free Software
Foundation, Inc.
Bu, KESİNLİKLE HİÇBİR GARANTİ VERİLMEYEN ücretsiz bir
yazılımdır. Ayrıntılar için "garanti" yazın.
4
```

Dikkatli bakarsak, telif hakkı mesajının altında sonucu görebiliriz. Bu mesaj -q (sessiz) seçeneği ile gizlenebilir.

bc etkileşimli olarak da kullanılabilir.

```
[me@linuxbox ~]$ bc -q
2 + 2
4
çık
```

bc'yi etkileşimli olarak kullanırken, gerçekleştirmek istediğimiz hesaplamaları yazmamız yeterlidir ve sonuçlar hemen görüntülenir. bc komutu quit, etkileşimli oturumu sonlandırır.

Standart giriş yoluyla bc'ye bir komut dosyası da aktarılabilir.

```
[me@linuxbox ~]$ bc < foo.bc 4
```

Standart girişi alabilme özelliği, burada belgeleri, burada dizeleri ve boruları kullanarak komut dosyalarını aktarabileceğimiz anlamına gelir. İşte bir burada dize örneği:

```
[me@linuxbox ~]$ bc <<< "2+2" 4
```

Bir Komut Dosyası Örneği

Gerçek hayattan bir örnek olarak, yaygın bir hesaplama olan aylık kredi ödemelerini gerçekleştiren bir komut dosyası oluşturacağız. Aşağıdaki komut dosyasında, `bc'ye` bir komut dosyası aktarmak için bir burada belge kullanıyoruz:

```
#!/bin/bash

# loan-calc: aylık kredi ödemelerini hesaplamak için komut dosyası

PROGNAME="${0##*/}" # Parametre genişletmeyi kullanarak temel adı

alın usage () {
    cat << _EOF_
Kullanım: $PROGNAME ANA PARA FAİZ AY

Nerede:

PRINCIPAL kredi tutarıdır. INTEREST sayı olarak
yıllık faiz oranıdır (7% = 0,07). MONTHS
kredinin vadesidir.

_EOF_
}

if (($# != 3)); then
    kullanım
    çıkış 1
fi

ana para=$1
faiz=$2
aylar=$3

bc <<- EOF
```

```
    ölçek = 10
    i = $faiz / 12 p =
    $ana para
    n = $ay
    a = p * ((i * ((1 + i) ^ n)) / (((1 + i) ^ n) - 1)) a,
    "\n" yazdır
EOF
```

Çalıştırıldığında, sonuçlar şöyle görünür:

```
[me@linuxbox ~]$ loan-calc 135000 0.0775 180
1270,7222490000
```

Bu örnekte, 135.000 dolarlık bir kredinin 180 ay (15 yıl) boyunca yıllık yüzde 7,75 faizle aylık ödemesi hesaplanmaktadır. Cevabın hassasiyetine dikkat edin. Bu, bc betiğinde özel `ölçek` değişkenine verilen değerle belirlenir. `bc` betik dilinin tam açıklaması `bc` man sayfasında verilmiştir. Matematiksel notasyonu kabuğunkinden biraz farklı olsa da (`bc` daha çok C'ye benzer), şimdiye kadar öğrendiklerimize dayanarak çoğu oldukça tanıdık gelecektir.

Özet

Bu bölümde, komut dosyalarında "gerçek işleri" yapmak için kullanılabilecek birçok küçük şey öğrendik. Komut dosyası yazma deneyimimiz arttıkça, dizeleri ve sayıları etkili bir şekilde işleme yeteneği son derece değerli olacaktır. `loan-calc` komut dosyamız, gerçekten yararlı şeyler yapmak için basit komut dosyaları bile oluşturulabileceğini göstermektedir.

Ekstra Puan

loan-calc betiğinin temel işlevselliği mevcut olsa da, betik henüz tamamlanmış değildir. Ekstra kredi için, `loan-calc` betiğini aşağıdaki özelliklerle geliştirmeyi deneyin:

- Komut satırı argümanlarının tam doğrulaması
- Kullanıcıdan kredinin anaparasını, faiz oranını ve vadesini girmesi istenen "etkileşimli" bir modu uygulamak için bir komut satırı seçeneği
- Çıktı için daha iyi bir format

Daha Fazla Okuma

- *Bash Referans Kılavuzu*, parametre genişletme konusunda iyi bir tartışma içerir: https://www.gnu.org/software/bash/manual/bashref.html#Shell-Parameter-Expan-sion
- *Wikipedia'da* bit işlemlerini açıklayan iyi bir makale var: https://en.wikipedia.org/wiki/Bit_operation
- ve üçlü işlemler hakkında bir makale: https://en.wikipedia.org/wiki/Ternary_operation
- ayrıca, https://en.wikipedia.org/wiki/Amortization_calculator

35 – Diziler

Önceki bölümde, kabuğun dizeleri ve sayıları nasıl işleyebileceğini inceledik. Şimdiye kadar incelediğimiz veri türleri, bilgisayar bilimleri çevrelerinde *skaler değişkenler* olarak bilinir; yani, tek bir değer içeren değişkenlerdir.

Bu bölümde, birden fazla değer tutan *dizi* adı verilen başka bir veri yapısı türünü inceleyeceğiz. Diziler, hemen hemen her programlama dilinin bir özelliğidir. Kabuk da dizileri destekler, ancak oldukça sınırlı bir şekilde. Yine de, bazı programlama problemlerini çözmek için yararlı olabilirler.

Diziler Nedir?

Diziler, aynı anda birden fazla değer tutan değişkenlerdir. Diziler tablo gibi düzenlenmiştir. Örnek olarak bir elektronik tabloyu ele alalım. Elektronik tablo, *iki boyutlu* bir *dizi* gibi davranır. Hem satırları hem de sütunları vardır ve elektronik tablodaki tek bir hücre, satır ve sütun adresine göre bulunabilir. Dizi de aynı şekilde davranır. Dizide, *elemanlar* olarak adlandırılan hücreler vardır ve her eleman veri içerir. Tek bir dizi elemanına, *indeks* veya *alt simge* olarak adlandırılan bir adres kullanılarak erişilir.

Çoğu programlama dili *çok boyutlu dizileri* destekler. Elektronik tablo, genişlik ve yükseklik olmak üzere iki boyuta sahip çok boyutlu bir dizi örneğidir. Birçok dil, rastgele sayıda boyuta sahip dizileri destekler, ancak iki ve üç boyutlu diziler muhtemelen en yaygın olarak kullanılanlardır.

`Bash'teki` diziler tek boyutlu ile sınırlıdır. Bunları tek sütunlu bir hesap tablosu olarak düşünebiliriz. Bu sınırlamaya rağmen, diziler için birçok uygulama vardır. Dizi desteği ilk olarak `bash` sürüm 2'de ortaya çıkmıştır. Orijinal Unix kabuk programı olan `sh`, dizileri hiç desteklemiyordu.

Dizi Oluşturma

Dizi değişkenleri, diğer `bash` değişkenleri gibi adlandırılır ve erişildiğinde otomatik olarak oluşturulur. İşte bir örnek:

```
[me@linuxbox ~]$ a[1]=foo
[me@linuxbox ~]$ echo ${a[1]}
foo
```

Burada, bir dizi öğesinin atanması ve erişilmesinin bir örneğini görüyoruz. İlk komutla, `dizi a'nın` 1. öğesine "foo" değeri atanır. İkinci komut, 1. öğenin depolanan değerini görüntüler. İkinci komutta köşeli parantezlerin kullanılması, kabuğun dizi öğesinin adında yol adı genişletme girişiminde bulunmasını önlemek için gereklidir.

Bir dizi, `declare` komutuyla da oluşturulabilir.

```
[me@linuxbox ~]$ declare -a a
```

`-a` seçeneği kullanılarak, bu `declare` örneği `a` dizisini oluşturur.

Dizilere Değer Atama

Değerler iki yoldan biriyle atanabilir. Tekil değerler aşağıdaki sözdizimi kullanılarak atanabilir; burada *`name`* dizinin adı, *`subscript`* ise sıfırdan büyük veya sıfıra eşit bir tamsayı (veya aritmetik ifade)dır:

`name``[`*`subscript`*`]=`*`value`*

Dizinin ilk öğesinin alt simgesi bir değil sıfırdır. *değer*, dizi öğesine atanan bir dize veya tamsayıdır.

Birden çok değer, aşağıdaki sözdizimi kullanılarak atanabilir. Burada *`name`*`,` dizinin adıdır ve *`value`* yer tutucuları, sıfır öğesinden başlayarak dizinin öğelerine sırayla atanan değerlerdir:

`ad=(`*`değer1 değer2`* `...)`

Örneğin, `days` dizisine haftanın günlerinin kısaltmalarını atamak istersek, bunu şu şekilde yapabiliriz:

```
[me@linuxbox ~]$ days=(Sun Mon Tue Wed Thu Fri Sat)
```

Her değer için bir alt simge belirleyerek belirli bir öğeye değer atamak da mümkündür.

```
[me@linuxbox ~]$ days=([0]=Sun [1]=Mon [2]=Tue [3]=Wed [4]=Thu
```

```
[5]=Cuma [6]=Cumartesi)
```

Dizi Öğelerine Erişim

Peki diziler ne işe yarar? Birçok veri yönetimi görevi bir elektronik tablo programı ile gerçekleştirilebildiği gibi, birçok programlama görevi de diziler ile gerçekleştirilebilir.

Basit bir veri toplama ve sunum örneğini ele alalım. Belirli bir dizindeki dosyaların değiştirilme zamanlarını inceleyen bir komut dosyası oluşturacağız. Bu verilerden yola çıkarak, komut dosyamız dosyaların en son hangi saatte değiştirildiğini gösteren bir tablo oluşturacak. Böyle bir komut dosyası, bir sistemin en aktif olduğu zamanı belirlemek için kullanılabilir. `Saatler` adlı bu komut dosyası şu sonucu verir:

```
[me@linuxbox ~]$ hours .
Saat   Dosya   Saat   Dosya
       lar            lar
---    -----   ---    -----
0      0       12     11
01     1       13     7
02     0       14     1
03     0       15     7
04     1       16     6
05     1       17     5
06     6       18     4
07     3       19     4
08     1       20     1
09     14      21     0
10     2       22     0
11     5       23     0

Toplam dosya
sayısı = 80
```

`Hours` programını, geçerli dizini hedef olarak belirterek çalıştırıyoruz. Program, günün her saati (0-23) için en son ne zaman değiştirilen dosya sayısını gösteren bir tablo oluşturur. Bunu oluşturmak için kullanılan kod aşağıdaki gibidir:

```
#!/bin/bash

# saatler: dosyaları değiştirilme zamanına göre sayan

komut dosyası kullanım () {
```

```
    echo "kullanım: ${0##*/} dizin" >&2
}

# Argümanın bir dizin olup olmadığını
kontrol et if [[ ! -d "$1" ]]; then
    kullanı
    m çıkış
    1
fi

# Diziyi başlat
for i in {0..23}; do hours[i]=0; done

# Verileri topla
i için $(stat -c %y "$1"/* | cut -c 12-13); yap j="${i#0}"
    ((++saatler[j]))
    ((++sayı))
done

# Verileri görüntüle
echo -e
"Saat\tDosyalar\tSaat\tDosyalar"
echo -e "----\t-----\t----\t----------"
 i için {0..11}; yap
    j=$((i + 12))
    printf "%02d\t%d\t%02d\t%d\n" \
         "$i" \
         "${saatler[i]}"
         \ "$j" \
         "${saatler[j]}"
done
printf "\nToplam dosya sayısı = %d\n" "$count"
```

Komut dosyası bir işlev (`kullanım`) ve dört bölümden oluşan bir ana gövdeden oluşur. İlk bölümde, bir komut satırı argümanı olup olmadığını ve bunun bir dizin olup olmadığını kontrol ederiz. Değilse, kullanım mesajını görüntüler ve çıkarız.

İkinci bölüm, `hours` dizisini başlatır. Bunu, her bir öğeye sıfır değeri atayarak yapar. Kullanımdan önce dizileri hazırlamak için özel bir gereklilik yoktur, ancak betiğimiz hiçbir öğenin boş olmadığından emin olmalıdır. Döngünün ilginç bir şekilde oluşturulduğuna dikkat edin. Köşeli parantez genişletme (`{0..23}`) kullanarak, `for` komutu için bir dizi kelimeyi kolayca oluşturabiliriz.

Sonraki bölüm, dizindeki her dosya üzerinde `stat` programını çalıştırarak verileri toplar. Sonuçtan iki basamaklı saati çıkarmak için `cut` kullanırız. Döngü içinde,

saat alanından baştaki sıfırları kaldırmamız gerekir, çünkü kabuk `00` ile `09` arasındaki değerleri sekizlik sayılar olarak yorumlamaya çalışır (ve sonuçta başarısız olur) (bkz. Tablo 34-2). Ardından, günün saatine karşılık gelen dizi öğesinin değerini artırırız. Son olarak, dizindeki toplam dosya sayısını izlemek için bir sayacı (`count`) artırırız.

Komut dosyasının son bölümü dizinin içeriğini görüntüler. Önce birkaç başlık satırı çıktıktan sonra dört sütunlu çıktı üreten bir döngüye giriyoruz. Son olarak, dosyaların nihai sayısını çıktıyoruz.

Dizi İşlemleri

Birçok yaygın dizi işlemi vardır. Dizileri silme, boyutlarını belirleme, sıralama vb. gibi işlemlerin komut dosyalarında birçok uygulaması vardır.

Dizinin Tüm İçeriğini Çıktılamak

`*` ve `@` alt simgeleri, bir dizideki her öğeye erişmek için kullanılabilir. Konumsal parametrelerde olduğu gibi, `@` notasyonu ikisi arasında daha kullanışlı olanıdır. İşte bir örnek:

```
[me@linuxbox ~]$ animals=("a dog" "a cat" "a fish")
[me@linuxbox ~]$ for i in ${animals[*]}; do echo $i; done a
köpe
k
kedi
balı
k
[me@linuxbox ~]$ for i in ${animals[@]}; do echo $i; done a
köpe
k
kedi
balı
k
[me@linuxbox ~]$ for i in "${animals[*]}"; do echo $i; done a
köpek kedi balık
[me@linuxbox ~]$ for i in "${animals[@]}"; do echo $i; done a
köpek
bir
kedi
bir
balık
```

`animals` dizisini oluşturuyoruz ve ona üç adet iki kelimelik dize atıyoruz. Ardından, dizinin içeriği üzerinde kelime bölünmesinin etkisini görmek için dört döngü çalıştırıyoruz. `$`

`{animals[*]}` ve `${animals[@]}` notasyonlarının davranışı, tırnak içine alınana kadar aynıdır. `*` notasyonu, dizinin içeriğini içeren tek bir kelimeyle sonuçlanırken, `@` notasyonu, dizinin "gerçek" içeriğiyle eşleşen üç iki kelimelik dizeyle sonuçlanır.

Dizi Öğelerinin Sayısını Belirleme

Parametre genişletmeyi kullanarak, bir dizideki öğe sayısını, bir dizgenin uzunluğunu bulmakla hemen hemen aynı şekilde belirleyebiliriz. İşte bir örnek:

```
[me@linuxbox ~]$ a[100]=foo
[me@linuxbox ~]$ echo ${#a[@]} # dizi elemanlarının sayısı
1
[me@linuxbox ~]$ echo ${#a[100]} # öğe 100'ün uzunluğu
3
```

Dizi `a'yı` oluşturuyoruz ve 100 numaralı öğeye `foo` dizesini atıyoruz. Ardından, `@` notasyonunu kullanarak dizi uzunluğunu incelemek için parametre genişletme kullanıyoruz. Son olarak, `foo` dizesini içeren 100 numaralı öğenin uzunluğuna bakıyoruz. İlginçtir ki, dizgiyi 100 numaralı öğeye atamış olsak da, bash dizide yalnızca bir öğe olduğunu bildirir. Bu, dizinin kullanılmayan öğelerinin (0 ila 99 numaralı öğeler) boş değerlerle başlatılıp sayıldığı diğer bazı dillerin davranışından farklıdır. `Bash'ta`, dizi öğeleri, alt simgelerine bakılmaksızın yalnızca bir değer atanmışsa var olurlar.

Bir Dizinin Kullandığı Alt Simgeleri Bulma

`Bash`, dizilerin alt simge atamalarında "boşluklar" içermesine izin verdiği için, bazen hangi öğelerin gerçekten var olduğunu belirlemek yararlı olabilir. Bu, aşağıdaki formları kullanarak parametre genişletme ile yapılabilir; burada *dizi,* bir dizi değişkeninin adıdır:

`${!`*`array`*`[*]}`

`${!`*`array`*`[@]}`

`*` ve `@` kullanan diğer genişletmeler gibi, tırnak içine alınmış `@` biçimi ayrı kelimelere genişlediğinden en kullanışlı olanıdır.

```
[me@linuxbox ~]$ foo=([2]=a [4]=b [6]=c)
[me@linuxbox ~]$ for i in "${foo[@]}"; do echo $i; done a
b
c
[me@linuxbox ~]$ for i in "${!foo[@]}"; do echo $i; done
```

```
2
4
6
```

`read -a` ile Dizi Elemanlarını Atama

`read` yerleşik komutu, daha önce yaptığımız gibi bir dizi değişken yerine kelimeleri indeksli bir diziye yerleştirmek için bir seçeneğe (`-a`) sahiptir. İşte bir örnek:

```
[me@linuxbox ~]$ declare -a foo
[me@linuxbox ~]$ read -a foo <<< "0. 1. 2. 3. 4."
[me@linuxbox ~]$ for i in "${foo[@]}"; do echo "$i"; done 0.
1.
2.
3.
4.
```

Dizinin Sonuna Öğe Ekleme

Bir dizinin sonuna değerler eklememiz gerektiğinde, dizideki öğe sayısını bilmek bir fayda sağlamaz, çünkü `*` ve `@` gösterimleri tarafından döndürülen değerler, kullanılan maksimum dizi indeksini bize söylemez. Neyse ki, kabuk bize bir çözüm sunar.
`+=` atama operatörünü kullanarak, bir dizinin sonuna otomatik olarak değerler ekleyebiliriz. Burada, `foo` dizisine üç değer atıyoruz ve ardından üç değer daha ekliyoruz.

```
[me@linuxbox ~]$ foo=(a b c)
[me@linuxbox ~]$ echo ${foo[@]} a
b c
[me@linuxbox ~]$ foo+=(d e f)
[me@linuxbox ~]$ echo ${foo[@]} a
b c d e f
```

Bir Dosyayı Dizgiye Okuma

Bash'ın son sürümleri, standart girişi doğrudan indeksli bir diziye okuyabilen `mapfile` adlı yeni bir yerleşik işlev içerir. Sözdizimi şöyledir:

```
mapfile -seçenekler dizi
```

`mapfile` komutu bazı ilginç seçeneklere sahiptir, bunlardan bazıları Tablo 35-1'de gösterilmiştir.

Tablo 35-1: mapfile Seçenekleri

Seçenek	Açıklama
`-d` *sınırlayıcı*	Satır sonlarını yeni satır yerine *`delim`* ile sonlandırın.
`-n` *sayı*	Sadece *`count`* satırını okuyun.
`-O` *başlangıç*	Dizi öğelerini 0 indeksinden ziyade *`origin`* indeksinden atamaya başlayın.
`-s` *sayı*	Dosyanın başındaki *`count`* satırını atla.
`-t`	Her satırın sonundaki sınırlayıcıyı kesin.

`Mapfile'ın` nasıl çalıştığını göstermek için, rastgele dört kelimelik şifreler üreten kısa bir komut dosyası oluşturacağız. Bunlar, uzun olmalarına rağmen hatırlanması kolay oldukları için (yazımını biliyorsanız ;-)) geleneksel şifrelerin alternatifi olarak kullanışlıdır.

```
#!/bin/bash

# array-mapfile - mapfile yerleşik işlevini göster
```

```
DICTIONARY=/usr/share/dict/words
WORDLIST=~/wordlist.txt
declare -a words

# filtrelenmiş kelime listesi
oluştur grep -v \' <
"$DICTIONARY" \
    | grep -v "[[:upper:]]" \
    | shuf > "$WORDLIST"

# WORDLIST'i diziye oku
mapfile -t -n 32767 kelime < "$WORDLIST"
```

```
# dört kelimelik parola oluştur
while [[ -z $REPLY ]]; do
    echo  "${words[$RANDOM]}"  \
          "${words[$RANDOM]}"  \
          "${words[$RANDOM]}"  \
          "${words[$RANDOM]}"
    echo
    read -r -p "Devam etmek için Enter, çıkmak için q
    tuşuna basın > " echo
done
```

Bu komut dosyası, apostrofları ve büyük harfleri kaldırmak için filtrelenmiş `/usr/share/dict/words` dosyasını kullanır. Kelime listesini karıştırmak için `shuf` komutunu kullanarak güzel bir rastgele sıra elde ederiz.

Ardından, dosyadaki ilk 32.767 kelimeyi `words` dizisine yükleriz. Neden 32767? Çünkü `RANDOM` değişkenini kullanarak diziden rastgele öğeler seçeceğiz ve `RANDOM` değişkenine her başvurulduğunda, 0 ile 32.767 arasında rastgele bir tamsayı döndürür.

```
[me@linuxbox ~]$ ./array-mapfile
dönüşümler uykular uzantılar metastaz Enter tuşuna

basarak devam edin, q tuşuna basarak çıkın >

kettles rhinestones unused demagnetizes

Devam etmek için Enter, çıkmak için q

tuşuna basın >

aşınma karakteristik ekstrüzyonu ifade eder

Devam etmek için Enter, çıkmak için q tuşuna basın > q
```

Burada çıktıyı görüyoruz. `ENTER tuşuna` her bastığımızda, komut dosyası dört rastgele kelime görüntüler.

Diziyi Dilimleme

Bir diziden bir grup bitişik öğeyi (*dilim* olarak adlandırılır) çıkarmak için kullanabileceğimiz bir parametre genişletme biçimi vardır. Bu genişletme, aşağıda gösterildiği gibi orijinal dizinin istenen diliminden dizi öğeleriyle sonuçlanır.

```
[me@linuxbox ~]$ arr=(0. 1. 2. 3. 4.)
[me@linuxbox ~]$ echo "${arr[@]:2:3}"
2. 3. 4.
```

Bu örnekte, beş öğeden oluşan bir dizi oluşturuyoruz. Ardından, diziden ikinci indeksden başlayarak üç öğeyi çıkarıyoruz. Negatif bir indeks değeri belirleyerek, dizinin başından değil sonundan itibaren sayıyoruz. Aşağıdaki örnekte, dizinin son iki öğesini çıkarıyoruz. Eksi işaretinden önce gerekli olan boşluğa dikkat edin.

```
[me@linuxbox ~]$ echo "${arr[@]: -2:2}" 3.
4.
```

Ayrıca, bir dilimin öğelerini içeren bir diziyi kolayca oluşturabiliriz.

```
[me@linuxbox ~]$ arr2=("${arr[@]:2:3}")
[me@linuxbox ~]$ echo "${arr2[@]}"
2. 3. 4.
```

Burada `arr2` dizisini oluşturduk ve `arr` dizisinden üç öğe ile doldurduk.

Diziyi Sıralama

Tıpkı elektronik tablolarda olduğu gibi, genellikle bir veri sütunundaki değerleri sıralamak gerekir. Kabukta bunu doğrudan yapmanın bir yolu yoktur, ancak biraz kodlama ile bunu yapmak zor değildir.

```
#!/bin/bash

# array-sort: Bir diziyi

sıralama a=(f e d c b a)

echo "Orijinal dizi: " "${a[@]}"
a_sorted=($(for i in "${a[@]}"; do echo "$i"; done | sort))
echo "Sıralanmış dizi:   " "${a_sorted[@]}"
```

Çalıştırıldığında, komut dosyası şu sonucu verir:

```
[me@linuxbox ~]$ array-sort
Orijinal dizi: f e d c b a
Sıralanmış dizi:    a b c d e
f
```

Komut dosyası, orijinal dizinin (`a`) içeriğini, karmaşık bir komut ikamesi ile ikinci bir diziye (`a_sorted`) kopyalayarak çalışır. Bu temel teknik, boru hattının tasarımını değiştirerek dizi üzerinde birçok türde işlem gerçekleştirmek için kullanılabilir.

Diziyi Silme

Bir diziyi silmek için `unset` komutunu kullanın.

```
[me@linuxbox ~]$ foo=(a b c d e f)
[me@linuxbox ~]$ echo ${foo[@]}
a b c d e f
[me@linuxbox ~]$ unset foo
[me@linuxbox ~]$ echo ${foo[@]}

[me@linuxbox ~]$
```

`unset` tek bir dizi öğesini silmek için de kullanılabilir.

```
[me@linuxbox ~]$ foo=(a b c d e f)
[me@linuxbox ~]$ echo ${foo[@]}
a b c d e f
[me@linuxbox ~]$ unset 'foo[2]'
[me@linuxbox ~]$ echo ${foo[@]}
a b d e f
```

Bu örnekte, dizinin üçüncü öğesini, alt simge 2'yi siliyoruz. Diziler bir ile değil, sıfır ile başlar, unutmayın! Ayrıca, kabuğun yol adı genişletmesini önlemek için dizi öğesinin tırnak içine alınması gerektiğini de unutmayın.

İlginç bir şekilde, bir diziye boş bir değer atamak, dizinin içeriğini boşaltmaz.

```
[me@linuxbox ~]$ foo=(a b c d e f)
[me@linuxbox ~]$ foo=
[me@linuxbox ~]$ echo ${foo[@]}
```

```
b c d e f
```

Alt simgesi olmayan bir dizi değişkenine yapılan herhangi bir başvuru, dizinin sıfırıncı elemanını ifade eder.

```
[me@linuxbox ~]$ foo=(a b c d e f)
[me@linuxbox ~]$ echo ${foo[@]}
a b c d e f [me@linuxbox
~]$ foo=A
[me@linuxbox ~]$ echo ${foo[@]} A
b c d e f
```

İlişkisel Diziler

`bash` sürüm 4.0 ve üstü, *ilişkisel dizileri* destekler. İlişkisel diziler, dizi indeksleri olarak tamsayılar yerine dizeler kullanır, böylece sözlük veya karma tablo gibi anahtar-değer çiftleri oluşturur. Bu özellik, veri yönetimine ilginç yeni yaklaşımlar getirir. Örneğin, `colors` adlı bir dizi oluşturabilir ve renk adlarını indeks olarak kullanabiliriz.

```
declare -A colors
colors["red"]="#ff0000"
colors["green"]="#00ff00"
colors["blue"]="#0000ff"
```

Sadece referansla oluşturulan tamsayı indeksli dizilerden farklı olarak, ilişkisel diziler `-A` seçeneği kullanılarak declare komutuyla açıkça oluşturulmalıdır. İlişkisel dizi öğelerine, tamsayı indeksli dizilere erişilmesiyle hemen hemen aynı şekilde erişilir.

```
echo ${colors["blue"]}
```

İlişkisel dizilerin anahtar-değer çifti özelliğini arama görevlerini gerçekleştirmek için kullanabiliriz. İlişkisel dizi, indeksli dizilerde olduğu gibi verileri sırayla aramak yerine, istenen verilere doğrudan erişmemizi sağlar. Aşağıdaki programda, /usr/bin dizinindeki tüm dosyaların adlarını ve boyutlarını bir diziye okuduktan sonra, kullanıcı girdisine göre aramalar gerçekleştiriyoruz.

```
#!/bin/bash
```

```
# dizi-arama - ilişkisel dizi kullanarak arama yapmayı gösterir

declare -A cmds

# diziyi komutlar ve dosya boyutlarıyla
doldur cd /usr/bin || exit 1
echo "Komutlar yükleniyor..."
for i in ./*; do
    cmds["$i"]=$(stat -c "%s" "$i")
done
echo "${#cmds[@]} komut yüklendi"

# arama yap while
true; do
    read -r -p "Komut girin (çıkmak için boş bırakın) ->
    " [[ -z $REPLY ]] && break
    if [[ -x $REPLY ]]; then
        echo "$REPLY" "${cmds[./$REPLY]}" "bayt"
    else
        echo "Böyle bir komut yok '$REPLY'."
  fi
tama
m
```

Programı çalıştırdığımızda aşağıdakileri elde ederiz:

```
[me@linuxbox ~]$ ./array-lookup
Komutlar yükleniyor...
2329 komut yüklendi
Komut girin (çıkmak için boş bırakın) ->
ls ls 138216 bayt
Komut girin (çıkmak için boş bırakın) ->
cp cp 141832 bayt
Komut girin (çıkmak için boş bırakın) ->
mv mv 137752 bayt
Komut girin (çıkmak için boş bırakın) ->
rm rm 59912 bayt
Komut girin (çıkmak için boş
bırakın) -> [me@linuxbox ~]$
```

Bir sonraki bölümde, ilginç bir rapor oluşturmak için ilişkisel dizileri iyi bir şekilde kullanan bir komut dosyasına bakacağız.

Birden Çok Boyutu Simüle Etmek İçin İlişkisel Dizileri Kullanma

`Bash'ın` yalnızca tek boyutlu dizileri doğrudan desteklediği doğru olsa da, ilişkisel bir dizi kullanarak ve çok boyutlu dizi adreslerine benzeyen dizin dizeleri oluşturarak çok boyutlu dizileri "taklit etmek" zor değildir. İşte bir örnek:

```
#!/bin/bash

# array-multi - çok boyutlu bir diziyi simüle edin declare

-A multi_array

# Diziyi bir sayı dizisiyle yükle counter=1
for row in {1..10}; do for
    col in {1..5}; do
        adres="$satır, $sütun"
        çoklu_dizi["$adres"]=$sayaç ((sayaç++))
    done
done

# Dizi içeriğini çıktı
olarak ver for row in
{1..10}; do
    sütun için {1..5}; yap
        adres="$satır, $sütun"
        echo -ne "${multi_array["$adres"]}" "\t"
    bitt
    i
    echo
bitti
```

Bu programı çalıştırdığınızda aşağıdaki sonuç elde edilir:

```
[me@linuxbox ~]$ ./array-multi
1   2    3    4    5
6   7    8    9    10
16  17   18   19   20
21  22   23   24   25
26  27   28   29   30
31  32   33   34   35
36  37   38   39   40
```

```
41 42    43    44    45
46 47    48    49    50
```

Özet

`Bash` man sayfasında "*array"* kelimesini arattığımızda, `bash'ın` dizi değişkenlerini kullandığı birçok örnek buluyoruz. Bunların çoğu oldukça belirsizdir, ancak bazı özel durumlarda ara sıra yararlı olabilirler. Aslında, diziler konusu, büyük ölçüde geleneksel Unix kabuk programlarının (`sh` gibi) dizileri desteklememesi nedeniyle, kabuk programlamada oldukça az kullanılmaktadır. Bu popülerlik eksikliği talihsiz bir durumdur, çünkü diziler diğer programlama dillerinde yaygın olarak kullanılmaktadır ve birçok programlama problemini çözmek için güçlü bir araç sağlarlar.

Diziler ve döngüler doğal olarak birbirine yakındır ve sıklıkla birlikte kullanılır. Aşağıdaki döngü biçimi, indeksli dizi alt simgelerini hesaplamak için özellikle uygundur:

```
for ((expr; expr; expr))
```

Daha Fazla Okuma

- Bu bölümde ele alınan veri yapıları hakkında birkaç Wikipedia makalesi: https://en.wikipedia.org/wiki/Scalar_(computing)

  https://en.wikipedia.org/wiki/Associative_array

36 – Egzotik

Yolculuğumuzun bu son bölümünde, bazı ufak tefek konulara değineceğiz. Önceki bölümlerde pek çok konuyu ele almış olsak da, henüz ele almadığımız birçok `bash` özelliği var. Bunların çoğu oldukça belirsiz ve esas olarak `bash'ı` bir Linux dağıtımına entegre edenler için yararlıdır. Ancak, yaygın olarak kullanılmasa da belirli programlama sorunları için yararlı olan birkaç özellik vardır. Bunları burada ele alacağız.

Grup Komutları ve Alt Kabuklar

`bash,` komutların gruplandırılmasına izin verir. Bu, iki yoldan biriyle yapılabilir: *grup komutu* veya *alt kabuk*. Grup komutlarını 6. bölümde tanıtmıştık ve hatırlayacağınız gibi, grup komutu şu sözdizimini kullanır:

```
{ komut1; komut2; [komut3; ...] }
```

Alt kabuklar da benzer bir sözdizimi kullanır.

```
(komut1; komut2; [komut3;...])
```

Grup komutu komutlarını köşeli parantezlerle çevrelerken, alt kabuk parantezler kullanır. `Bash'ın` grup komutlarını uygulama şekli nedeniyle, köşeli parantezlerin komutlardan bir boşlukla ayrılması ve son komutun kapanış köşeli parantezinden önce bir noktalı virgül veya yeni satır ile sonlandırılması gerektiği unutulmamalıdır.

Peki, grup komutları ve alt kabuklar ne işe yarar? Aralarında önemli bir fark olsa da (buna birazdan değineceğiz), her ikisi de yönlendirmeyi yönetmek için kullanılabilir. Birden fazla komutta yönlendirme yapan bir komut dosyası segmentini ele alalım.

```
ls -l > output.txt
echo "foo.txt listesi" >> çıktı.txt cat
foo.txt >> çıktı.txt
```

Bu oldukça basittir. Üç komutun çıktısı `output.txt` adlı bir dosyaya yönlendirilir. Grup komutunu kullanarak bunu şu şekilde kodlayabiliriz:

```
{ ls -l; echo "foo.txt listesi"; cat foo.txt; } > output.txt
```

Alt kabuk kullanmak da benzerdir.

```
(ls -l; echo "foo.txt listesi"; cat foo.txt) > çıktı.txt
```

Bu tekniği kullanarak yazma işinden biraz kurtulduk, ancak grup komutlarının veya alt kabukların asıl parladığı yer boru hatlarıdır. Komutlardan oluşan bir boru hattı oluştururken, genellikle birkaç komutun sonuçlarını tek bir akışta birleştirmek yararlıdır. Grup komutları ve alt kabuklar bunu kolaylaştırır.

```
{ ls -l; echo "foo.txt dosyasının listesi"; cat foo.txt; } | lpr
```

Burada, üç komutumuzun çıktısını birleştirdik ve bunları `lpr` komutunun girdisine aktardık, böylece basılı bir rapor oluşturduk.

Grup komutları ve alt kabuklar benzer görünse de ve her ikisi de yönlendirme için akışları birleştirmek için kullanılabilse de, ikisi arasında önemli bir fark vardır. Grup komutu tüm komutlarını geçerli kabukta yürütürken, alt kabuk (adından da anlaşılacağı gibi) komutlarını geçerli kabuğun bir alt kopyasında yürütür. Bu, ortamın (kabuk işlevleri, takma adlar ve değişkenler gibi şeyler dahil) kopyalanıp kabuğun yeni bir örneğine verildiği anlamına gelir. Bu ortam kopyası, alt kabuğun ebeveynin ortamının tüm kopyasını miras alması, alt işlemin ise ebeveyn kabuğundan yalnızca dışa aktarılan değişkenleri miras alması bakımından, alt işlemin normalde çalışma şeklinden farklıdır. Alt kabuk çıktığında, ortam kopyası kaybolur, bu nedenle alt kabuğun ortamında yapılan tüm değişiklikler (değişken atamaları dahil) de kaybolur.

En önemlisi, alt kabuklar, alt işlemler gibi ve grup komutlarından farklı olarak, ana kabuğun ortamını değiştiremez. Bunu gösterelim; önce bir grup komutuyla:

```
[me@linuxbox ~]$ foo="Orijinal Değer"
[me@linuxbox ~]$ { foo="Değiştirilmiş Değer"; echo $foo; }
Değiştirilmiş Değer
[me@linuxbox ~]$ echo $foo
Değiştirilmiş Değer
```

Şimdi, aynı adımları bir alt kabukla yapacağız.

```
[me@linuxbox ~]$ foo="Orijinal Değer"
[me@linuxbox ~]$ ( foo="Değiştirilen Değer"; echo $foo )
Değiştirilen Değer
[me@linuxbox ~]$ echo $foo
Orijinal Değer
```

Gördüğümüz gibi, grup komutu mevcut kabukta `foo` değerini değiştirebilirken, alt kabuk bunu yapamaz. Alt kabukların bu özelliği, dizinleri değiştirmek gibi işlemler yapmak istediğimizde kullanışlıdır. Alt kabukta `cd` komutunu uyguladığımızda, mevcut çalışma dizini alt kabuk süresince değişir, ancak ana kabuğa döndükten sonra mevcut çalışma dizini değişmez.

```
[me@linuxbox ~]$ pwd
/home/me
[me@linuxbox ~]$ ( cd /usr/bin; pwd )
/usr/bin [me@linuxbox
~]$ pwd
/home/me
```

Ancak genel olarak, bir komut dosyası alt kabuk gerektirmedikçe, grup komutları alt kabuklara tercih edilir. Grup komutları hem daha hızlıdır hem de daha az bellek gerektirir.

Aşağıdaki komut dosyasında, grup komutlarını kullanacak ve ilişkisel dizilerle birlikte kullanılabilecek birkaç programlama tekniğine bakacağız. `array-2` adlı bu komut dosyası, bir dizin adı verildiğinde, dizindeki dosyaların listesini, dosyanın sahibi ve grup sahibinin adlarıyla birlikte yazdırır. Listenin sonunda, komut dosyası her sahibine ve gruba ait dosya sayısının toplamını yazdırır. Burada, komut dosyasına `/usr/bin` dizini verildiğinde elde edilen sonuçları (kısalık için özetlenmiş) görüyoruz:

```
[me@linuxbox ~]$ array-2 /usr/bin
/usr/bin/2to3-2.6                       kök         root
/usr/bin/2to3                           kök         root
/usr/bin/a2p                            kök         root
/usr/bin/abrowser                       kök         root
/usr/bin/aconnect                       kök         root
/usr/bin/acpi_fakekey                   kök         root
/usr/bin/acpi_listen                    kök         root
/usr/bin/add-apt-repository             kök         root
.                                       root
```

```
.
.
/usr/bin/zipgrep                                kök         root
/usr/bin/zipinfo                                kök         root
/usr/bin/zipnote                                kök         root
/usr/bin/zip                                    kök         root
/usr/bin/zipsplit                               kök         root
/usr/bin/zjsdecode                              kök         root
/usr/bin/zsoelim                                kök         root

Dosya sahipleri:
daemon     :      1 dosya(lar)
root       : 1394   dosya(lar)

Dosya grubu
sahipleri:
crontab    :      1 dosya(lar)
daemon     :      1 dosya(lar)
lpadmin    :      1 dosya(lar)
posta      :      4 dosya(lar)
mlocate    :      1 dosya(lar)
kök        :   1380 dosya(lar)
gölge      :      2 dosya(lar)
ssh        :      1 dosya(lar)
tty        :      2 dosya(lar)
utmp       :      2 dosya(lar)
```

İşte komut dosyasının satır numaralarıyla birlikte listesi:

```
     1    #!/bin/bash
     2
     3    # dizi-2: Dosya sahiplerini saymak için dizileri
     4    kullanın
     5
     6    declare -A dosyalar dosya_grubu dosya_sahibi gruplar
     7
     8    sahipler if [[ ! -d "$1" ]]; then
     9        echo "Kullanım: array-2 dir" >&2
    10        exit 1
    11    fi
    12
    13    for i in "$1"/*; do owner="$(stat
    14        -c %U "$i")"
              group="$(stat -c %G "$i")"
```

```
15          dosyalar["$i"]="$i"
16          dosya_sahibi["$i"]="$sahib
17          i"
18          dosya_grubu["$i"]="$grup"
19          ((++sahipler[$sahibi]))
20          ((++gruplar[$grup]))
21      bitti
22
23      # Toplanan dosyaları listele
24      { for i in "${files[@]}"; do printf
25          "%-40s %-10s %-10s\n" \
26              "$i" "${file_owner["$i"]}" "${file_group["$i"]}"
27      done } | sort
28      echo
29
30      # Sahipleri listele
31      echo "Dosya sahipleri:"
32      { for i in "${!owners[@]}"; do
33          printf "%-10s: %5d dosya(lar)\n" "$i"
34      "${owners["$i"]}" done } | sort
35      echo
36
37      # Grupları listele
38      echo "Dosya grubu sahipleri:"
39      { for i in "${!groups[@]}"; do
40          printf "%-10s: %5d dosya(lar)\n" "$i"
        "${groups["$i"]}" done } | sort
```

Bu komut dosyasının işleyişine bir göz atalım:

Satır 5: İlişkisel diziler, `-A` seçeneği kullanılarak `declare` komutuyla oluşturulmalıdır seçeneği ile declare komutu kullanılarak oluşturulmalıdır. Bu komut dosyasında, aşağıdaki gibi beş dizi oluşturuyoruz:

- `files`, dizindeki dosyaların adlarını içerir ve dosya adına göre indekslenir.
- `file_group`, dosya adına göre indekslenmiş her dosyanın grup sahibini içerir.
- `file_owner`, dosya adına göre indekslenmiş her dosyanın sahibini içerir.
- `groups`, indekslenen gruba ait dosya sayısını içerir.
- `owners,` indekslenen sahibine ait dosya sayısını içerir.

Satır 7-10: Bu satırlar, geçerli bir dizin adının konumsal parametre olarak geçirilip geçirilmediğini kontrol eder. Geçirilmediyse, kullanım mesajı görüntülenir ve komut dosyası 1 çıkış durumuyla sonlandırılır.

Satır 12-20: Bu satırlar, dizindeki dosyaları döngüsel olarak tarar. `stat` komutunu kullanarak

komutunu kullanarak, 13. ve 14. satırlar dosya sahibinin ve grup sahibinin adlarını çıkarır ve dosya adını dizi indeksi olarak kullanarak değerleri ilgili dizilere (16. ve 17. satırlar) atar. Benzer şekilde, dosya adı da `dosya` dizisine atanır (15. satır).

Satır 18-19: Dosya sahibine ve grup sahibine ait toplam dosya sayısı bir artırılır.

Satır 22-27: Dosya listesi çıktısı alınır. Bu, `"${array[@]}"` parametre genişletmesi kullanılarak yapılır; bu genişletme, her bir öğeyi ayrı bir kelime olarak ele alarak dizi öğelerinin tüm listesini genişletir. Bu, dosya adının gömülü boşluklar içermesine olanak tanır. Ayrıca, tüm döngünün parantez içine alındığını ve böylece bir grup komutu oluşturduğunu da unutmayın. Bu, döngünün tüm çıktısının `sort` komutuna aktarılmasını sağlar. Bu, ilişkisel dizi öğelerinin genişletilmesinin sıralanmaması nedeniyle gereklidir.

Satır 29-40: Bu iki döngü, `"${!array[@]}"` genişletmesini kullanmalarıdır. Bu genişletme, dizi öğelerinin listesi yerine dizi indekslerinin listesini genişletir.

İşlem Değiştirme

Bölüm 28'de, bir boru hattındaki `read` komutunun sezgisel olarak beklediğimiz gibi çalışmadığını keşfettiğimizde, alt kabuk ortamı sorununun bir örneğini gördük. Özetlemek gerekirse, aşağıdaki gibi bir boru hattı oluşturursak, `read` komutu bir alt kabukta yürütüldüğü ve alt kabuk sona erdiğinde `REPLY` değişkeninin kopyası yok edildiği için REPLY değişkeninin içeriği her zaman boş olur.

```
echo "foo" | read
echo $REPLY
```

Boru hatlarındaki komutlar her zaman alt kabuklarda yürütüldüğünden, değişkenleri atayan herhangi bir komut bu sorunla karşılaşacaktır. Neyse ki, kabuk bu sorunu çözmek için kullanılabilecek, *işlem ikamesi* adı verilen egzotik bir genişletme biçimi sunar.

İşlem ikamesi iki şekilde ifade edilir.

Standart çıktı üreten işlemler için şöyle görünür:

`<(`*`liste`*`)`

veya standart girişi alan işlemler için şöyle görünür:

`>(`*`liste`*`)`

burada *liste,* komutların bir listesidir.

`read` ile ilgili sorunumuzu çözmek için, şu şekilde işlem ikamesi kullanabiliriz:

```
read < <(echo "foo")
echo $REPLY
```

İşlem ikamesi, alt kabuğun çıktısını yeniden yönlendirme amacıyla sıradan bir dosya olarak ele almamızı sağlar. Aslında, bu bir genişletme biçimi olduğu için, gerçek değerini inceleyebiliriz.

```
[me@linuxbox ~]$ echo <(echo "foo")
/dev/fd/63
```

Genişlemenin sonucunu görüntülemek için `echo komutunu` kullanarak, alt kabuğun çıktısının `/dev/fd/63` adlı bir dosya tarafından sağlandığını görüyoruz.

İşlem ikamesi genellikle `read` içeren döngülerle kullanılır. İşte bir örnek alt kabuk tarafından oluşturulan bir dizin listesinin içeriğini işleyen `okuma` döngüsü:

```
#!/bin/bash

# pro-sub: işlem ikamesi demosu

while read -r attr links owner group size date time filename; do cat
    << _EOF_
Dosya adı:    $filename
Boyut:        $size
Sahip:        $owner
Grup:         $group
Değiştirilme tarihi:
              $date $time
Bağlantılar:  $links
Öznitelikler: $attr

_EOF_
done < <(ls -l --time-style="+%F %H:%m"| tail -n +2)
```

Döngü, dizin listesinin her satırı için `okuma işlemini` gerçekleştirir. Liste, komut dosyasının son satırında oluşturulur. Bu satır, işlem ikamesinin çıktısını döngünün standart girişine yönlendirir. `Tail` komutu, listeye gerek olmayan ilk satırı ortadan kaldırmak için işlem ikamesi ardışık düzenine dahil edilmiştir.

Yürütüldüğünde, komut dosyası şu şekilde bir çıktı üretir:

```
[me@linuxbox ~]$ pro-sub | head -n 20
Dosya adı:    addresses.ldif
Boyut:        14540
Sahibi:       me
Grup:         ben
Değiştirilme tarihi:     02.04.2009 11:12
Bağlantılar:  1
Özellikler: -rw-r--r--

Dosya adı:    bin
Boyut:        4096
Sahibi:       ben
Grup:         ben
Değiştirilme tarihi:     10.07.2009 07:31
Bağlantılar:  2
Özellikler: drwxr-xr-x

Dosya adı:    bookmarks.html
Boyut:        394213
Sahibi:       ben
Grup:         ben
```

`eval` ile Komut Oluşturma

`Eval` yerleşik komutu garip ve gizemli bir komuttur. Basitçe açıklamak gerekirse, bir dizi argüman alır, bunları bir dizge halinde birleştirir ve yürütülmesi için kabuğa iletir. Bu durumda doğal olarak şu soru ortaya çıkar: Bu ne işe yarar? Neden kullanırız?

Kabuk betiklerinde, çalışma zamanında dinamik olarak bir komut oluşturmamız gereken belirli durumlar vardır ve `eval` bunu yapmamızı sağlar. Bir deney yapalım. Açıkça ele almamış olsak da, bir komutu bir dize değişkenine yerleştirmek ve ardından parametre genişletme özelliğini kullanarak değişkeni bir komuta genişletmek mümkündür.

```
[me@linuxbox ~]$ cmd="echo foo"
[me@linuxbox ~]$ $cmd
foo
[me@linuxbox ~]$
```

Şimdi komutumuza bir değişken eklediğimizde ne olacağına bakalım.

```
[me@linuxbox ~]$ str=abcde
[me@linuxbox ~]$ cmd="echo $str"
[me@linuxbox ~]$ $cmd
abcde
[me@linuxbox ~]$
```

Bu, beklediğimiz sonucu veriyor, ancak
`str'ye yeni bir değer` atadığımızda ne olacağını görelim.

```
[me@linuxbox ~]$ str=ABCDE
[me@linuxbox ~]$ $cmd abcde
[me@linuxbox ~]$
```

Sonuçlar değişmedi. Bu mantıklıdır, çünkü `cmd` atadığımızda `$str` üzerinde parametre genişletme işlemini zaten gerçekleştirmiştik. Peki, değişkeni daha sonra genişletmek istersek ne yapmalıyız? İlk değişken ataması sırasında parametre genişletmeyi engellemek için komutumuzu tek tırnak içine alabiliriz.

```
[me@linuxbox ~]$ cmd='echo $str'
[me@linuxbox ~]$ $cmd
$str [me@linuxbox
~]$
```

`cmd` ataması sırasında parametre genişletmesi elde edemedik, ancak `cmd'yi` çalıştırdığımızda da elde edemedik. Bunun nedeni, genişletme (`$cmd`) daha fazla genişletilebilecek bir sonuç (`$str`) verse bile, kabuk genişletmeyi yalnızca bir kez gerçekleştirmesidir.

`Eval` kullanırsak, ek genişletme düzeyi elde ederiz.

```
[me@linuxbox ~]$ str=abcde
[me@linuxbox ~]$ cmd='echo $str'
[me@linuxbox ~]$ eval "$cmd" abcde
[me@linuxbox ~]$ str=ABCDE
[me@linuxbox ~]$ eval "$cmd"
ABCDE
[me@linuxbox ~]$
```

Bu bize `eval'in` ne yaptığını gösterir. Argümanlarını tek bir dizeye birleştirir; dizenin içeriği üzerinde tilde, parametre ve yol adı genişletmesi gerçekleştirir; ve ardından dizeyi yürütülmesi için geçerli kabuğa (alt kabuk oluşturulmaz) geçirir.

`eval` ile Dikkatli Olun

`eval` komutu kötü bir üne sahiptir. Bunun nedeni, `eval` ile bir kabuk betiğine güvenlik açıkları eklemenin ne kadar kolay olduğu konusundaki endişelerdir. `eval'e` verilen dize harici bir kaynaktan (yani kullanıcı girdisi) geliyorsa, dizeye yetkisiz komutlar gömülmediğinden emin olmak için dikkatli olunmalıdır (buna *kod enjeksiyon saldırısı* denir). `eval'e` verilen dizenin içeriğini her zaman doğrulayın.

Bir Wordle Yardımcısı

Aşağıdaki betikte, Wordle bulmacasının olası cevaplarını bulmak için `eval komutunu` kullanacağız.

Bu popüler oyuna aşina olmayanlar için Wikipedia oyunu şu şekilde tanımlamaktadır:

> *Wordle, Galli yazılım mühendisi Josh Wardle tarafından oluşturulan ve geliştirilen web tabanlı bir kelime oyunudur. Oyuncular beş harfli bir kelimeyi tahmin etmek için altı deneme hakkına sahiptir ve her tahmin için harflerin eşleştiği veya doğru pozisyonda olduğu durumları gösteren renkli karolar şeklinde geri bildirim verilir.*

Temel olarak, bir tahminde bulunduğumuzda oyun, tahminimizdeki harflerin gizli kelimede olup olmadığını ve tahminimizdeki bir harfin gizli kelimedeki konumuyla eşleşip eşleşmediğini bize söyler. Böylece, her tahminde bulunduğumuzda, (umarız) gizli kelimenin ne olduğunu bulana kadar gizli kelime hakkında daha fazla bilgi ediniriz. Oyunun amacı, en az deneme sayısıyla kelimeyi tahmin etmektir.

Yardımcı komut dosyası, o ana kadar yapılan tahminlerden bilinenlerle eşleşen kelimelerin bir listesini çıkarır. Bunu yapmak için, komut dosyasına, hangi harflerin bilindiğini ve gizli kelimede hangi konumlarda bulunduklarını belirten basit bir düzenli ifade verilir. Ayrıca, gizli kelimede bulunduğu bilinen harflerin ve gizli kelimede bulunmadığı bilinen harflerin bir listesini de sağlıyoruz.

Gizli kelimenin "about" olduğunu ve bizim "bloat" tahmininde bulunduğumuzu varsayalım. Wordle bize "o" ve "t" harflerinin kelimede ilgili konumlarda yer aldığını ve kelimenin bilinmeyen konumlarda "a" ve "b" harflerini de içerdiğini ve ayrıca gizli kelimenin "l" harfini içermediğini söyler. Bu durumda, komut dosyasını şu şekilde çağırırız:

```
[me@linuxbox ~]$ eval-wordle ..o.t +a +b -l
```

İlk argüman, "o" ve "t" harflerinin sırasıyla yer aldığı beş karakterlik bir düzenli ifadedir. Sonra, gizli kelimede bulunan bilinen karakterler ve gizli kelime olmadığı bilinen harfler gelir. Komut dosyası, sözlükte yalnızca iki kelimenin seçim kriterlerini karşıladığını gösteren aşağıdaki yanıtı verir:

```
abort
yakla
şık 2
```

Komut dosyası, seçim kriterlerine göre bir sözlüğü (`/usr/share/dict/words`) filtreleyerek ve ardından kalanları görüntüleyerek çalışır. Bunu yapmak için, komut satırında kaç tane konumsal parametre sağlandığına bağlı olarak uzunluğu değişen uzun, çok parçalı bir ardışık düzen oluşturmamız gerekir. Bu şekilde, ara sonuçları tutmak için geçici dosyalara olan ihtiyacı ortadan kaldırırız.

```
 1  #!/bin/bash

 2  # eval-wordle - wordle bulmacalarını çözerek eval'i göster

 3  PROGNAME=${0##*/}
 4  DICTIONARY=/usr/share/dict/words

 5  kullanım() {
 6    printf "%s\n" "Kullanım: ${PROGNAME} [-h|--help]"
 7    printf "%s\n" "        ${PROGNAME} regex +-char..."
 8  }

 9  help_message() {
10    cat << _EOF_
11  $PROGNAME - Wordle
    Yardımcısı

12  $(kullanım)
13  Seçenekler:
14    -h, --help     Bu yardım mesajını görüntüleyin ve
                     çıkın.
15
Argümanlar:
```

```
16      İlk argüman, cevapta beş bilinmeyen karakteri temsil
17      eden en az beş karakterlik bir düzenli ifade ('.....')
18      olmalıdır. Bunu, cevapta mevcut olduğu veya bulunmadığı
19      bilinen sıfır veya daha fazla karakter izler. Bunlar,
20      mevcut olduğu bilinen harfler için
21      +char olarak ifade edilir.
22      cevapta bulunmadığı bilinen harfler için -char olarak
        ifade edilir.
23     Örnek:

24      $PROGNAME a.... +o -v

25      Bu, cevabın ilk konumda 'a' ile başladığını ve ayrıca 'o'
        harfini içerdiğini bildiğimiz anlamına gelir.
26      ve ayrıca 'o' harfini içerdiğini, ancak
27      'v' harfi içermemelidir.

    _EOF_
28      dönüş
29    }

30    add_plus() { # Bir harf ekle
31      local char="$1"

32      echo " | grep $char"
33      dön
34    }

35    add_minus() { # Bir harfi çıkar
36      yerel char="$1"

37      echo " | grep -v $char"
38      dön
39    }

40    # Komut satırını ayrıştır

41    if [[ "$1" == "-h" || "$1" == "--help" ]]; then
42      yardım_mesajı
43      çıkış 0
44    fi

45    eğer (( "${#1}" == 5 )); o zaman
49      bilinen_karakterler="$1"
```

```
50     shift
   else
46     echo "İlk argüman 5 karakterlik bir düzenli ifade olmalıdır." >&2
47     çıkış 1
51   fi

52   cmd="LANG=POSIX grep '^.....$' $DICTIONARY \
53     | grep -v '[[:punct:]]' \
54     | grep -v '[[:upper:]]' \
55     | grep '$known_chars'"

56   while [[ -n "$1" ]]; do
57     case "$1" in
58       -[[:alpha:]])
59         cmd="${cmd}$(add_minus "${1:1}")"
60         ;;
61       +[[:alpha:]])
62         cmd="${cmd}$(add_plus "${1:1}")"
63         ;;
64       *)
65         echo "Geçersiz argüman '$1'" >&2
66         çıkış 1
67         ;;
68     esac
69     shift
70   bitti

71   eval "$cmd | tee >(wc -l)"
```

Komut dosyasının ilginç özelliklerini inceleyelim:

Satır 30-39: Son boru hattının tekil öğelerini oluşturmak için kullanılan iki işlev tanımlar. Bir işlev, belirtilen harfi gerektiren bir filtre eklerken, diğer işlev belirtilen harfi içeren tüm kelimeleri hariç tutan bir filtre ekler.

Satır 40-51: İlk parametreden başlayarak konumsal parametrelerin işlenmesine başlar. İlk komut satırı argümanının yardım seçeneği mi yoksa gerekli beş karakterli düzenli ifade mi olduğunu kontrol ederiz.

Satır 52: Boru hattının başlangıcını oluştururuz. İlk adım, sözlüğü filtreleyerek tam olarak beş karakter uzunluğunda olmayan tüm kelimeleri kaldırmaktır. Ayrıca, Wordle asla özel isimler kullanmadığından, noktalama işaretleri (genellikle kesme işaretleri) ve büyük harfler içeren tüm kelimeleri filtreleriz.

Satır 56: Komut satırı argümanlarının geri kalanını işleyen bir döngü başlatır. Her

bir "+harf" veya "-harf" bulunduğunda, baştaki artı veya eksi işaretini kaldırır ve ilgili işlevi çağırarak boru hattının bir sonraki öğesini komut dizesine ekleriz.

Satır 71: Asıl işin yapıldığı yer burasıdır. Toplam kelime sayısını elde etmek için, filtrelenmiş kelime listesini `tee komut`una aktarırız. Bu komut, listeyi standart çıktıya (ekranda görüntülenmesi için) ve aslında bir işlem ikamesi içinde çalışan `wc` programına aktarır. Zor bir işlem.

Tuzaklar

10. bölümde, programların sinyallere nasıl yanıt verebileceğini gördük. Bu özelliği komut dosyalarımıza da ekleyebiliriz. Şimdiye kadar yazdığımız komut dosyaları bu özelliğe ihtiyaç duymamıştır (çünkü yürütme süreleri çok kısadır ve geçici dosyalar oluşturmazlar), ancak daha büyük ve daha karmaşık komut dosyaları sinyal işleme rutininden yararlanabilir.

Büyük, karmaşık bir komut dosyası tasarlarken, komut dosyası çalışırken kullanıcının oturumu kapatması veya bilgisayarı kapatması durumunda ne olacağını dikkate almak önemlidir. Böyle bir durum meydana geldiğinde, etkilenen tüm süreçlere bir sinyal gönderilir. Buna karşılık, bu süreçleri temsil eden programlar, programın düzgün ve düzenli bir şekilde sonlandırılmasını sağlamak için eylemler gerçekleştirebilir. Örneğin, yürütülmesi sırasında geçici bir dosya oluşturan bir komut dosyası yazdığımızı varsayalım. İyi bir tasarım için, komut dosyası işini bitirdiğinde dosyayı silmesini isteriz. Programın erken sonlandırılacağını belirten bir sinyal alındığında komut dosyasının dosyayı silmesi de akıllıca olacaktır.

`bash`, bu amaç için *trap* olarak bilinen bir mekanizma sağlar. Trap'ler, uygun şekilde adlandırılmış yerleşik komut olan `trap` ile uygulanır. `trap`, aşağıdaki sözdizimini kullanır; burada *argument*, okunacak ve komut olarak işlenecek bir dizedir ve *signal*, yorumlanan komutun yürütülmesini tetikleyecek bir sinyalin belirtisidir:

`trap` *`argüman sinyal`* [*`sinyal...`*]

İşte basit bir örnek:

```
#!/bin/bash

# trap-demo: basit sinyal işleme demosu

trap "echo 'Seni görmezden geliyorum.'" SIGINT

SIGTERM for i in {1..5}; do
    echo "5'in $i yinelemesi"
    sleep 5
tamam
```

Bu komut dosyası, komut dosyası çalışırken SIG-INT veya SIGTERM sinyali alındığında her seferinde `echo` komutunu çalıştıracak bir tuzak tanımlar. Kullanıcı `Ctrl-c` tuşuna basarak komut dosyasını durdurmaya çalıştığında programın çalışması şöyle görünür:

```
[me@linuxbox ~]$ trap-demo
5'in 1. yinelemesi
5'in 2. yinelemesi
^CSeni görmezden geliyorum.
5'in 3. yinelemesi
^CSeni görmezden geliyorum.
5'in 4. yinelemesi
5'in 5'inci yinelemesi
```

Gördüğümüz gibi, kullanıcı programı her kesintiye uğratmaya çalıştığında, bunun yerine mesaj yazdırılıyor.

Yararlı bir komut dizisi oluşturmak için bir dize oluşturmak zor olabilir, bu nedenle komut olarak bir kabuk işlevi belirtmek yaygın bir uygulamadır. Bu örnekte, işlenecek her sinyal için ayrı bir kabuk işlevi belirtilmiştir:

```
#!/bin/bash

# trap-demo2: basit sinyal işleme demosu

exit_on_signal_SIGINT () {
    echo "Komut dosyası kesildi."
    2>&1 exit 0
}

exit_on_signal_SIGTERM () {
    echo "Komut dosyası sonlandırıldı."
    2>&1 exit 0
}

trap exit_on_signal_SIGINT SIGINT trap
exit_on_signal_SIGTERM SIGTERM

i için {1..5}; yap
    echo "5'in $i yinelemesi"
    sleep 5
done
```

Bu komut dosyası, her sinyal için birer tane olmak üzere iki `trap` komutu içerir. Her trap, sırayla, belirli bir sinyal alındığında yürütülecek bir kabuk işlevini belirtir. Her sinyal işleme işlevinde bir `exit` komutunun bulunduğuna dikkat edin. `Exit` komutu olmasaydı, komut dosyası işlevi tamamladıktan sonra devam ederdi.

Kullanıcı bu betiğin yürütülmesi sırasında `Ctrl-c` tuşuna bastığında, sonuçlar şöyle görünür:

```
[me@linuxbox ~]$ trap-demo2 5'in
1. yinelemesi
5'in 2. yinelemesi
^CScript kesildi.
```

Tüm programlarda kullanılabilen sinyallerin yanı sıra, `trap` yerleşik komutu birkaç dahili ve bash'a özgü sinyali de destekler. Bunlar arasında özellikle `EXIT` ve `ERR` ilgi çekicidir. Beklenebileceği gibi, `EXIT` trap bir betik sona erdiğinde etkinleştirilir. `ERR` trap, bir komut (bazı istisnalar dışında) sıfır olmayan bir çıkış durumuyla sona erdiğinde etkinleştirilir. Bu, Bölüm 30'da incelediğimiz `set -e` seçeneğinin daha kullanışlı bir versiyonu gibi çalışır. Şimdi `EXIT` ve `ERR` tuzaklarının kısa bir gösterim betiği var. 5. satırdaki kasıtlı hataya dikkat edin.

```
 1  #!/bin/bash

 2  # trap-demo3 - ERR ve EXIT sinyal işlemeyi gösterir

 3  trap "echo \"Bir hata var.\"" ERR
 4  trap "echo \"Program sona erdi.\"" EXIT

 5  echox "Çalışıyor..."

 6  read -r -p "Bir şey söyle... " bir şey
 7  echo "$something"
```

Bu komut dosyası tuzakları tanımlar ve ardından bir satır metin görüntülemeyi dener, ancak echo komutu yanlış yazılmıştır. Komut dosyası, kullanıcı girişini beklediği bir sonraki satıra geçer. Son olarak, kullanıcının girdiği her şeyi görüntüler.

Bu komut dosyasını çalıştırdığımızda, aşağıdakileri elde ederiz:

```
[me@linuxbox ~]$ trap-demo3
```

```
./trap-demo3: satır 8: echox: komut bulunamadı
Bir hata var.
Bir şey söyleyin...

Program sona erdi.
```

İlk olarak, yanlış yazdığımız komutla ilgili kabuktan bir hata mesajı alırız. Ardından, ERR tuzağının etkinleştirildiğini gösteren mesaj gelir. Sonra okuma istemini görürüz ve ENTER tuşuna basarsak, boş bir satır çıkar ve EXIT tuzağı mesajı programın bittiğini bildirir.

Geçici Dosyalar

Sinyal işleyicilerin komut dosyalarına dahil edilmesinin bir nedeni, komut dosyasının yürütme sırasında ara sonuçları saklamak için oluşturabileceği geçici dosyaları kaldırmaktır. Geçici dosyalara isim vermek bir tür sanattır. Geleneksel olarak, Unix benzeri sistemlerdeki programlar geçici dosyalarını bu tür dosyalar için tasarlanmış paylaşımlı bir dizin olan /tmp dizininde oluşturur. Ancak, dizin paylaşımlı olduğu için, bu durum özellikle süper kullanıcı ayrıcalıklarıyla çalışan programlar için belirli güvenlik sorunları yaratır. Sistemin tüm kullanıcılarına açık olan dosyalar için uygun izinleri ayarlamanın yanı sıra, geçici dosyalara tahmin edilemeyen dosya adları vermek de önemlidir. Bu, *geçici yarış saldırısı* olarak bilinen bir istismarı önler. Tahmin edilemeyen (ancak yine de açıklayıcı) bir ad oluşturmanın bir yolu şöyledir:

```
tempfile=/tmp/$(basename $0).$$.$RANDOM
```

Bu, programın adından, ardından işlem kimliğinden (PID) ve ardından rastgele bir tamsayıdan oluşan bir dosya adı oluşturur. Ancak, $RAN-DOM kabuk değişkeninin yalnızca 1-32767 aralığında bir değer döndürdüğünü ve bu aralığın bilgisayar terimleriyle büyük bir aralık olmadığını, bu nedenle değişkenin tek bir örneğinin kararlı bir saldırganı alt etmek için yeterli olmadığını unutmayın.

Daha iyi bir yöntem, geçici dosyayı adlandırmak ve oluşturmak için mktemp programını (mktemp standart kütüphane işleviyle karıştırılmamalıdır) kullanmaktır. mk-temp programı, dosya adını oluşturmak için kullanılan bir şablonu argüman olarak kabul eder. Şablon, bir dizi büyük harf "X" karakteri içermelidir; bu karakterler, karşılık gelen sayıda rastgele harf ve rakamla değiştirilir. "X" karakterlerinin dizisi ne kadar uzunsa, rastgele karakterlerin dizisi de o kadar uzun olur. İşte bir örnek:

```
tempfile=$(mktemp /tmp/foobar.$$.XXXXXXXXXX)
```

Bu, geçici bir dosya oluşturur ve adını `tempfile` değişkenine atar. Şablondaki "X" karakterleri rastgele harf ve sayılarla değiştirilir, böylece son dosya adı (bu örnekte, PID'yi elde etmek için özel parametre `$$`'nin genişletilmiş değeri de içerir) şunun gibi olabilir:

```
/tmp/foobar.6593.UOZuvM6654
```

Normal kullanıcılar tarafından yürütülen komut dosyaları için, `/tmp` dizinini kullanmaktan kaçınmak ve kullanıcının ana dizininde geçici dosyalar için bir dizin oluşturmak akıllıca olabilir. Bunun için aşağıdaki gibi bir kod satırı kullanabilirsiniz:

```
[[ -d $HOME/tmp ]] || mkdir $HOME/tmp
```

Asenkron Yürütme

Bazen aynı anda birden fazla görevi gerçekleştirmek istenebilir. Modern işletim sistemlerinin, çok kullanıcılı olmasa da en azından çok görevli olduğunu gördük. Komut dosyaları, çok görevli bir şekilde çalışacak şekilde oluşturulabilir.

Genellikle bu, bir komut dosyasının başlatılmasını ve ardından ana komut dosyası çalışmaya devam ederken ek bir görevi gerçekleştirmek için bir veya daha fazla alt komut dosyasının başlatılmasını içerir. Ancak, bir dizi komut dosyası bu şekilde çalıştırıldığında, ana ve alt komut dosyalarının koordinasyonunu sağlamak sorun yaratabilir. Yani, ana veya alt komut dosyası diğerine bağımlıysa ve bir komut dosyası, kendi görevini tamamlamadan önce diğerinin görevini tamamlamasını beklemesi gerekiyorsa ne olur?

`bash`, bu tür *eşzamansız yürütmeyi* yönetmeye yardımcı olan yerleşik bir komuta sahiptir. `wait` komutu, ana komut dosyasının belirtilen işlem (yani alt komut dosyası) bitene kadar duraklamasını sağlar.

`wait`

Önce `wait` komutunu göstereceğiz. Bunun için iki komut dosyasına ihtiyacımız olacak. İlk olarak bir ana komut dosyası.

```
#!/bin/bash

# async-parent: Asenkron yürütme demosu (ana) echo "Ana:

başlıyor..."

echo "Ana: alt komut dosyasını
başlatıyor..." async-child &
```

```
pid=$!
echo "Ebeveyn: çocuk (PID= $pid) başlatıldı."

echo "Ana: devam ediyor..."
sleep 2

echo "Ana: çocuk işlemin bitmesini beklemek için
duraklatılıyor..." wait "$pid"

echo "Ana: çocuk tamamlandı. Devam ediyor..." echo
"Ana: ana tamamlandı. Çıkılıyor."
```

İkincisi bir alt komut dosyasıdır.

```
#!/bin/bash

# async-child: Asenkron yürütme demosu (alt) echo

"Alt: alt çalışıyor..."
sleep 5
echo "Child: child tamamlandı. Çıkılıyor."
```

Bu örnekte, alt komut dosyasının basit olduğunu görüyoruz. Asıl eylem üst komut dosyası tarafından gerçekleştiriliyor. Üst komut dosyasında, alt komut dosyası başlatılır ve arka plana alınır. Alt komut dosyasının işlem kimliği, `pid` değişkenine `$!` kabuk parametresinin değeri atanarak kaydedilir; bu parametre her zaman arka plana alınan son işin işlem kimliğini içerir.

Ana komut dosyası devam eder ve ardından alt işlemin PID'si ile bir `bekleme` komutu yürütür. Bu, alt komut dosyası sonlanana kadar ana komut dosyasının duraklamasını sağlar ve bu noktada ana komut dosyası sonlanır.

Yürütüldüğünde, ana ve alt komut dosyaları aşağıdaki çıktıyı üretir:

```
[me@linuxbox ~]$ async-parent Ana:
başlıyor...
Ana: alt komut dosyasını başlatıyor...
Ana: alt (PID= 6741) başlatıldı. Ana:
devam ediyor...
Alt: alt çalışıyor...
Ana: alt komutun bitmesini beklemek için duraklıyor...
```

```
Çocuk: Çocuk bitti. Çıkıyor.
Ebeveyn: Çocuk bitti. Devam ediyor... Ebeveyn:
Ebeveyn bitti. Çıkıyor.
```

Adlandırılmış Borular

Çoğu Unix benzeri sistemde, *adlandırılmış boru* adı verilen özel bir dosya türü oluşturmak mümkündür. Adlandırılmış borular, iki işlem arasında bağlantı oluşturmak için kullanılır ve diğer dosya türleri gibi kullanılabilir. Çok popüler değildirler, ancak bilinmesi yararlıdır.

İsimli borular gibi bir iletişim yönteminin yanı sıra ağ bağlantıları gibi diğer *işlemler arası iletişim* türlerini de kullanabilen, *istemci-sunucu* adı verilen yaygın bir programlama mimarisi vardır.

En yaygın kullanılan istemci-sunucu sistemi türü, elbette, bir web sunucusuyla iletişim kuran bir web tarayıcısıdır. Web tarayıcısı istemci görevi görür, sunucuya isteklerde bulunur ve sunucu tarayıcıya web sayfalarıyla yanıt verir.

Adlandırılmış borular dosya gibi davranır, ancak aslında ilk giren ilk çıkar (FIFO) tamponları oluşturur. Sıradan (adlandırılmamış) borularda olduğu gibi, veriler bir uçtan girer ve diğer uçtan çıkar. Adlandırılmış borularla, şuna benzer bir yapı kurulabilir:

process1 > named_pipe

ve bu:

process2 < named_pipe

ve şu şekilde davranacaktır:

süreç1 | süreç2

Adlandırılmış Boru Kurma

İlk olarak, bir named pipe oluşturmalıyız. Bu, `mkfifo` komutu kullanılarak yapılır.

```
[me@linuxbox ~]$ mkfifo pipe1
[me@linuxbox ~]$ ls -l pipe1
prw-r--r-- 1 me    me     0 2009-07-17 06:41 pipe1
```

Burada `mkfifo komutunu` kullanarak `pipe1` adlı bir adlandırılmış boru oluşturuyoruz. `ls komutunu` kullanarak dosyayı inceliyoruz ve öznitelik alanındaki ilk harfin "p" olduğunu görüyoruz, bu da dosyanın adlandırılmış bir boru olduğunu gösteriyor.

Adlandırılmış Boruların Kullanımı

Adlandırılmış borunun nasıl çalıştığını göstermek için iki terminal penceresine (veya alternatif olarak iki sanal konsola) ihtiyacımız olacak. İlk terminalde basit bir komut girip çıktısını adlandırılmış boruya yönlendiriyoruz.

```
[me@linuxbox ~]$ ls -l > pipe1
```

ENTER tuşuna bastıktan sonra, komut askıda kalmış gibi görünecektir. Bunun nedeni, borunun diğer ucundan henüz veri alan hiçbir şeyin olmamasıdır. Bu durumda, borunun *tıkalı* olduğu söylenir. Diğer uca bir işlem eklediğimizde ve bu işlem borudan girişi okumaya başladığında bu durum ortadan kalkacaktır. İkinci terminal penceresini kullanarak şu komutu giriyoruz:

```
[me@linuxbox ~]$ cat < pipe1
```

İlk terminal penceresinden üretilen dizin listesi, ikinci terminalde `cat` komutunun çıktısı olarak görünür. İlk terminaldeki `ls` komutu, engellenmediğinde başarıyla tamamlanır.

Özet

İşte, yolculuğumuz burada sona eriyor. Şimdi geriye kalan tek şey pratik yapmak, pratik yapmak, pratik yapmak. Yolculuğumuzda pek çok konuyu ele aldık, ancak keşfedilecek çok daha fazla şey var. Keşfedilmeyi ve kullanılmayı bekleyen binlerce komut satırı programı var. `/usr/bin` dizininde araştırmaya başlayın ve görün!

Daha fazlasını öğrenmek isteyen okuyucular, LinuxCommand.org adresinden ücretsiz olarak indirebilecekleri devam kitabı "*Linux Komut Satırı ile Maceralar*"a göz atabilirler.

Daha Fazla Okuma

- Bash man sayfasının "Bileşik Komutlar" bölümü, grup komutu ve alt kabuk notasyonlarının tam açıklamasını içerir.
- Bash man sayfasının EXPANSION bölümü, işlem ikamesi ile ilgili bir alt bölüm içerir.
- *Gelişmiş Bash Komut Dosyası Yazma Kılavuzu'nda* da işlem ikamesi hakkında bir tartışma bulunmaktadır: https://tldp.org/LDP/abs/html/process-sub.html
- *Linux Journal'da* adlandırılmış borular hakkında iki iyi makale bulunmaktadır. İlki, Eylül

1997 tarihli:
https://www.linuxjournal.com/article/2156

- ikincisi ise Mart 2009'da yayınlanmıştır:
https://www.linuxjournal.com/content/using-named-pipes-fifos-bash

Dizin

A

a2ps komutu....362
mutlak yol adları....9
alias komutu....52, 136
takma adlar....44, 52, 134
Amerikan Ulusal Standartlar Enstitüsü (bkz. ANSI)
....175
Amerikan Bilgi Değişim Standart Kodu (bkz. ASCII)
....17
çapalar....27
anonim FTP sunucuları....221
ANSI....175
ANSI kaçış kodları....175, 179
ANSI.SYS....175
Apache web sunucusu....126
AppImage paket biçimi....190
apropos komutu....50
apt komutu....185
apt komutu....185
apt-get komutu....185
aptitude komutu....185
Arch....182
arşivleme....253
aritmetik genişletme....74, 79, 395, 500, 509
aritmetik ifadeler....74, 497, 509, 511
aritmetik operatörler....74, 510
aritmetik doğruluk testleri....422, 509
diziler....
  sonuna değerler ekle....529
  değerleri atama....524
  ilişkisel....534, 540
  oluşturma....523
  silme....533
  eleman sayısını belirleme....528
  kullanılan alt simgeleri bulma....528
  dizin....523
  çok boyutlu....523
  değişkenleri okuma....432
  sıralama....532
  alt simge....523
  iki boyutlu....523
ASCII....82, 86, 244, 275, 288, 362
  zil karakteri....172
  satır başı....29
  sıralama düzeni....275, 277, 419
  kontrol kodları....82, 275, 356
  groff çıktı sürücüsü....34
  satır besleme karakteri....29
  boş karakter....244
  yazdırılabilir karakterler....275
  metin....17
aspell komutu....327
assembler....369
montaj dili....369
atama operatörleri....512
ilişkisel diziler....534, 540
asenkron yürütme....555
ses CD'leri....211
Otomatik Özel IP Adresleme....221
AWK programlama dili....327, 518

B

geriye dönük referanslar....288, 322 f.
geriye eğik çizgi kaçış dizileri....82
geriye eğik çizgi ile kaçış özel karakterler....17
yedeklemeler, artımlı....257
basename komutu....480
bash....2, 134
  man sayfası....50
temel düzenli ifadeler 278, 287 f., 320, 324, 334 bc
komutu....518
Berkeley Yazılım Dağıtımı....360
bg komutu....124
ikili....98 f., 103, 369, 510
bit maskesi....102

bit operatörleri 514
Bourne, Steve 2, 6
köşeli parantez genişletme 75, 79, 494
dallanma 41
break komutu 445, 488
bozuk bağlantılar 41
BSD stili 11
tamponlama 199
hatalar .....462 f.
yapı ortamı 374
bzip2 komutu 252

C

C programlama dili 369, 496, 513, 516
C 369
iptal komutu 367
satır başı. .18, 82, 172, 275 f., 291, 326 f., 359 durum
bileşik komut 469
büyük/küçük harf dönüştürme 507
cat komutu 61, 291
cd komutu 9, 11
CD-ROM'lar 211
cdrecord komutu 212
karakter sınıfları .....27 f., 272, 274, 276 f., 281, 317, 327
karakter aralıkları .....28, 273 f., 327
sağlama toplamı dosyası 213
chgrp komutu 109
alt işlem 115
chmod komutu 97, 110, 384
chown komutu 108, 110
Chrome 389
kronolojik sıralama 29
açık metin 221, 225
istemci-sunucu mimarisi 557
COBOL programlama dili 36
kod enjeksiyon saldırısı 547
karşılaştırma sırası 137, 275, 277, 317, 419
  ASCII 277, 419
  sözlük 276
  geleneksel 277
comm komutu 311
komut geçmişi 3, 88
komut satırı .....
  argümanlar 47
  düzenleme 3, 84
  genişletme 71
  tarih 3, 89
  arayüzler 29
komut seçenekleri 14
komut ikamesi 77, 79, 494
komutlar .....
  argümanlar 14, 476
  tipi belirleme 45
  belgeler 46
  yürütülebilir program dosyaları 44, 369
  başka bir kullanıcı olarak yürütme 105
  uzun seçenekler 14
  seçenekler 14
yorum 139, 147, 326, 383, 464
Ortak Unix Yazdırma Sistemi 358, 367
karşılaştırma operatörleri 51
derleyici 369
derleme 368
tamamlamalar 86
bileşik komutlar .....
  durum 469
  için 493
  eğer 412
  kadar 450
  süresince 443
  (( )) 422, 438, 509
  [[ ]] 421, 438
sıkıştırma algoritmaları 250
koşullu ifadeler 428, 457
yapılandırma dosyaları 18, 21, 134
configure komutu 374
sabitler 394
devam komutu 445
kontrol karakterleri 172, 291
kontrol kodları 82, 275
kontrol operatörleri .....
  && 426, 438
  || 426
kontrol terminali 115
KOPYALAMA 373
kopyalama ve yapıştırma .....
  vim'de 160
  komut satırında 85
  X Pencere Sistemi ile 3
coreutils paketi .....48, 51 f., 305, 331
bir dosyadaki kelimeleri sayma 66
cp komutu 30, 36, 144, 229
CPU .....114 f., 368
cron işi 234
bulmaca 272
csplit komutu 332
CUPS 358, 367
curl komutu 224
curl programı 371
geçerli çalışma dizini 8

imleç hareketi....84
kes komutu....302, 505

D

daemon programları....115, 126
veri sıkıştırma....249
veri yedekliliği....249
veri doğrulama....421
tarih komutu....4
tarih biçimleri....298
dd komutu....210
Debian....182
Debian Stili (.deb)....183
hata ayıklama....408, 463
deklare komutu....507
savunma amaçlı programlama....458, 463
sınırlayıcılar....80, 296, 299
bağımlılıklar....184, 377
tasarım....462 f.
aygıt sürücüleri....191, 369
aygıt adları....199
aygıt düğümleri....20
df komutu....4, 410
sözlük....370
sözlük sıralama düzeni....27
diff komutu....312
Dijital Kısıtlama Yönetimi (DRM)....184
dizinler....
arşivleme....253
değiştirme....9
kopyalama....30
oluşturma....29, 36
mevcut çalışma....8
silme....33, 41
hiyerarşik....7
ana....21, 95, 410
listeleme....13
taşınma....31, 38
gezinti....7
OLD_PWD değişkeni....13
ana....8
PATH değişkeni....137
PWD değişkeni....137
kaldırma....33, 41
yeniden adlandırma....31, 38
kök....7
paylaşılan....110
yapışkan bit....104
senkronize etme....261
ağ üzerinden aktarma....261
içeriği görüntüleme....9
disk bölümleri....194
DISPLAY değişkeni....136
dnf komutu....185
Dolphin....29
dos2unix komutu....292
çift tırnak....79
dpkg komutu....185
du komutu....294, 410
Dinamik Ana Bilgisayar Yapılandırma Protokolü (DHCP) 220

E

echo komutu....71, 136, 390
-e seçeneği....8
-n seçeneği....43
kenar ve köşe durumları....463
EDITOR değişkeni....137
etkin grup kimliği....104
etkin kullanıcı kimliği....104, 115
elif ifadesi....41
e-posta....290
gömülü sistemler....369
boş değişkenler....501
şifreli tüneller....22
şifreleme....317
dosya sonu....62, 398
sonsuz döngü....446
enscript komutu....365
ortam....106, 134, 436
takma adlar....134
kurmak....137
inceleme....134
giriş kabuğu....137
kabuk işlevleri....134
kabuk değişkenleri....134
başlangıç dosyaları....137
alt kabuklar....539
değişkenler....134
eqn komutu....347
ERR tuzağı....553
eval yerleşik....545
yürütülebilir dosyalar....37
yürütülebilir program dosyaları....44, 369
yürütülebilir programlar....
konum belirleme....45
PATH değişkeni....137
çıkış komutu....5, 417, 440
çıkış durumu....413, 417
ÇIKIŞ tuzağı....553
genişletme komutu....30

genişletmeler 71
aritmetik 74, 79, 395, 500, 509
köşeli parantez 75, 79, 494
komut ikamesi....77, 79, 494
sınırlayıcılar 80
sonuçlanan hatalar 455
geçmiş 89, 91
parametre....76, 79, 393, 400, 500
yol adı 72, 79, 494
tilde 73, 79
kelime bölme....79
ifadeler....
aritmetik 74, 497, 509, 511, 524
koşullu....428, 457
ext4....207
genişletilmiş düzenli ifadeler 27
Genişletilebilir İşaretleme Dili 290

F

yanlış komut....414
fg komutu....124
FIFO....557
dosya komutu....16
dosya tanımlayıcı 59
dosya sistemi bozulması 199
Dosya Aktarım Protokolü (FTP)....221
dosya adları 244
büyük/küçük harf duyarlı....12
gömülü alanlar 12, 285
uzantılar 12
gizli 11
öncü tireler 460
POSIX Taşınabilir Dosya Adı Karakter Kümesi 461
sorunlar 460
dosyalarıyla ilgili sorunlar....
erişim 94
arşivleme....253, 259
özellikler 95
blok özel....96
blok özel aygıt....235
dosya modunu değiştirme 97
sahip ve grup sahibi değiştirme....108
özel karakter....96
özel karakter aygıtı....235
sıkıştırma....249
yapılandırma 18, 134, 289
kopyalama 30, 36
ağ üzerinden kopyalama 221
boş oluşturma....58
deb....182
silme....33, 41, 241
içeriği belirleme....16
cihaz düğümleri 20
yürütme erişimi....95
ifadeler 415
bulma....232
gizli 11
ISO görüntüsü....211
liste 9, 13
mod....96
hareket....31, 37
sahibi....98
izinler....94
okuma erişimi 95
normal 235
kaldırma....33, 41
yeniden adlandırma 31, 37
rpm 182
paylaşımlı kütüphane....21
başlangıç....137
yapışkan bit 104
sembolik bağlantılar....235
senkronize etme....261
geçici....554
metin....17
ağ üzerinden aktarma 221, 258, 261
kesme....58
tipi....95
içeriği görüntüleme....17
yazma erişimi....95
görüntü....212
filtreler....65
bul komutu....234, 257
findutils paketi 248
Firefox....389
güvenlik duvarları....217
ilk giren ilk çıkar 557
flatpak paket formatı 190
disketler....200
akış kontrolü....
dal....412
durum bileşik komut 469
elif ifadesi....419
sonsuz döngü....446
bileşik komut için....493
for döngüsü....493
fonksiyon ifadesi 403
if bileşik komut 412
döngü....442
menü odaklı....438
çoktan seçmeli kararlar....469

while ve until döngüleri ile dosya okuma 451
döngüyü sonlandırma 445
tuzaklar.....551
döngüye kadar.....450
while döngüsü.....443
fmt komutu 338
odak politikası.....4
katlama komutu.....337
bileşik komut için 4
for döngüsü.....493
Fortran programlama dili.....369, 496
free komutu 5, 198
Özgür Yazılım Vakfı .....xix, xxii
fsck komutu 209
ftp komutu.....221, 230, 370, 399
FTP sunucuları 221, 399
FUNCNAME değişkeni.....48
fonksiyon ifadesi 403

G

gcc 370
gedit komutu 121, 143
genisoimage komutu.....211
Gentoo 182
getopts yerleşik komutu.....485
Ghostscript.....35
gid.....94
küresel değişkenler 405
genelleştirme.....27
GNOME 2, 29, 42, 101, 143, 230
gnome-terminal.....2
GNU binutils paketi 495
GNU C Derleyicisi 37
GNU coreutils paketi.....48, 51 f., 305, 331
GNU findutils paketi 248
GNU Projesi .....14, xix, xxii, 248, 331, 370, 372
info komutu 50
GNU/Linux.....xix, xxii
grafiksel kullanıcı arayüzleri.....xvii
grep komutu.....66, 267, 435
groff.....347
grup komutu 427
grup komutları 59, 407, 538
gruplar 94
etkin grup kimliği 104
gid.....94
birincil grup kimliği.....94
setgid 104
GUI.....3, xvii, 29, 42, 84, 101, 138
gunzip komutu.....250
gzip komutu 52, 250

H

halt komutu 13
sabit diskler 193
sabit bağlantılar 24, 35
oluşturma 38
listeleme 39
başlık komutu.....67
başlık dosyaları 373
merhaba dünya programı 383
yardım komutu.....46
burada belgeler.....397
burada dizeler.....436
onaltılık 98, 509
gizli dosyalar.....11, 28, 73
hiyerarşik dizin yapısı.....7
yüksek seviyeli programlama dilleri.....369
tarihçe.....
genişleme.....89, 91
arama.....89
geçmiş komutu.....89
ana dizinler.....21
root hesabı.....22
/etc/passwd.....95
ana dizin 8, 11, 73, 106, 137
HOME değişkeni 137
ana bilgisayar adı.....172
HTML 290, 327, 347, 389, 400, 402
Köprü Metni Biçimlendirme Dili 290

I

G/Ç yeniden yönlendirme (yeniden yönlendirme bölümüne bakın)
.....56
ICMP ECHO_REQUEST 217
id komutu.....94
IDE.....200
if bileşik komut.....139, 455, 469
ifconfig komutu.....219
IFS değişkeni 434
artımlı yedeklemeler 257
bilgi dosyaları 51
init 114
başlangıç komut dosyaları.....115
inode 39
YÜKLEME.....373
kurulum sihirbazı 183
tamsayılar.....
aritmetik 74, 518
bölme 75, 510

ifadeler 419
etkileşim 429
Dahili Alan Ayırıcı 434
yorumlanan diller 369
yorumlanan programlar 370
tercüman 369
ip komutu 219
iso görüntüleri ....211 f.
iso9660 212

J

iş kontrolü 122
iş numaraları 122
iş özellikleri 123
katıl komutu 307
Joliet uzantıları 211
Joy, Bill 151

K

kate komutu 143
KDE 2, 29, 42, 101, 143, 230
çekirdek. .xvi, xix f., 49, 114, 127, 191, 200, 315, 378
anahtar alanlar 296
kill komutu 125
killall komutu 129
metni öldürme 85
Knuth, Donald 347
konsol 2
kwrite komutu 121, 143

L

LANG değişkeni 137, 276, 278
less komutu 17, 64, 261, 286
lftp komutu 22
kütüphaneler 36
LibreOffice Writer ....18, xx, 573
satır devam karakteri 387
satır düzenleyiciler 151
satır devam karakteri 326
bağlayıcı 369
bağlama 369
bağlantılar ....
..
bozuk 41
oluşturma 34
zor 24, 35
sembolik 24, 35
Linux topluluğu 182
Linux dağıtımları 182
Arch 182
CentOS 183
Debian ....182 f., 368
Fedora ...., 95, 183
Gentoo 182
Linux Mint 183
OpenSUSE ...., 183
paketleme sistemleri 182
Raspbian 183
Red Hat Enterprise Linux 183
Slackware 182
Ubuntu ....xix, 182 f.
Linux Dosya Sistemi Hiyerarşi Standardı .19, 25, 386
Linux çekirdeği ...xvi, xix f., 49, 114, 127, 191, 200, 315, 378
aygıt sürücüleri 191
literal karakterler 270
canlı CD'ler xix
ln komutu 34, 38
yerel değişkenler 405
yerel ayar 275, 277, 317, 419
yerel ayar komutu 277
localhost 226
lokalize komutu 232, 285
mantıksal hatalar 457
mantıksal işlemler 423
mantıksal operatörler 238
mantıksal ilişkiler 237, 241
giriş istemcisi 5, 222
giriş kabuğu 95, 106, 137
uzun seçenekler 14
döngü arayüzü 22
döngü 442
döngüler 457, 511, 514, 537, 544
kayıpsız sıkıştırma 250
kayıplı sıkıştırma 250
kayıp+bulunan dizin 209
küçük harfleri büyük harfe dönüştürme 508
lp komutu 361
lpq komutu 366
lpr komutu 360
lprm komutu 367
lpstat komutu 365
ls komutu 9, 13
uzun biçim 15
dosya özniteliklerini görüntüleme 95
lsblk komutu 20
Lukyanov, Alexander 224
LVM (Mantıksal Birim Yöneticisi) 193

M

makine dili....36
bakım....386, 390, 392, 401
make komutu....375
make dosyası....375
man komutu....48
man sayfaları....48, 348
mapfile komutu....529
Markdown....290
işaretleme dilleri....290, 347
bellek....
  her bir işleme atanan....115
  serbest alan görüntüleme....5
  Yerleşik set boyutu....118
  bölümleme ihlali....12
  kullanım....11
  kullanım görüntüleme....131
  sanal....118
menü odaklı programlar....438
meta tuşu....86
meta diziler....270
meta karakterler....270
meta veri....183, 185
mkdir komutu....29, 36
mkfifo komutu....557
mkfs komutu....207
mkisofs komutu....212
mktemp komutu....554
mlocate....233
mnemonics....369
modal editör....154
tek aralıklı yazı tipleri....358
Moolenaar, Bram....151
daha fazla komut....19
mount komutu....195, 212
bağlantı noktaları....21, 195, 197
bağlama....194
MP3....110
çoklu kullanıcı sistemleri....93
çoktan seçmeli kararlar....469
çoklu görev....93, 114, 555
mv komutu....31, 37

N

adlandırılmış borular....55
nano komutu....15
Nautilus....29
ağ....216
  anonim FTP sunucuları....221
  varsayılan yol....221
  Dinamik Ana Bilgisayar Yapılandırma Protokolü (DHCP)....220
  şifreli tüneller....229
  ağ ayarlarını ve istatistiklerini inceleyin....219
  Dosya Aktarım Protokolü (FTP)....221
  güvenlik duvarları....217
  FTP sunucuları....221
  yerel alan ağı....220
  döngü geri besleme arabirimi....220
  ortadaki adam saldırıları....225
  yönlendiriciler....219
  uzaktaki ana bilgisayarlarla güvenli iletişim....225
  ana bilgisayarın çalışıp çalışmadığını test etme 217
  bir ana bilgisayara giden yolu izleme....218
  dosya aktarımı....261
  dosya aktarımı....221
  sanal özel ağ....229
satır sonu karakteri....172
satır sonları....80
HABER....373
güzel komut....124
nl komutu....333
nohup komutu....129
nroff komutu....347
null karakteri....244
sayı tabanları....509

O

sekizlik....98, 509, 527
Ogg Vorbis....110
OLD_PWD değişkeni....137
OpenOffice.org Writer....., 573
OpenSSH....225
operatörler....
  aritmetik....74, 510
  atama....512
  ikili....456
  karşılaştırma....515
  üçlü....516
OPTARG değişkeni....485
OPTIND değişkeni....485
dosya sahibi olmak....94

P

paket dosyaları....183
paket bakımcıları....183
paket yönetimi....182
  deb....182
  Debian Stili (.deb)....183

paket bulma....185
yüksek seviyeli araçlar....184
paketleri yükleme....186
Düşük seviyeli araçlar....184
paket depoları....183
Red Hat Stili (.rpm)....183
paketleri kaldırma....187
RPM....182
paketleri güncelleme....187
paketleme sistemleri....182
sayfa açıklama dili....290, 349, 357
PAGER değişkeni....137
çağrı cihazları....19
parametre genişletme....76, 79, 500, 531
ana dizin....8
üst süreç....115
parted komutu....203
passwd komutu....111
şifreler....111
yapıştır komutu....306
PATA....200
yama komutu....315
yamalar....312
PATH değişkeni....137, 139, 384, 403
yol adı genişletme....72, 79, 494
yol adları....285
mutlak....9
tamamlama....86
göreceli....9
desen....508
PDF....350, 360
Perl programlama dili. 44, 267, 327, 369, 518 izinler....382
PHP programlama dili....369
ping komutu....21
boru hatları....63, 436, 543
in komut ikamesi....78
plocate....23
taşınabilirlik....374, 411, 426
taşınabilir....411
Taşınabilir Belge Formatı....350, 360
Taşınabilir İşletim Sistemi Arayüzü....279
konumsal parametreler....476, 501 f., 504
POSIX....275, 278 f., 426
karakter sınıfları....27 f., 274, 276 f., 281, 317, 327
POSIX-....211
PostScript....290, 349, 357, 362, 367
güç kapatma komutu....130
pr komutu....341, 358
birincil grup kimliği....94
yazdırılabilir karakterler....27
printenv komutu....77, 134
yazıcı tamponları....198
yazıcılar....198, 200
çıktı tamponlama....198
kontrol kodları....35
daisy-wheel....356
cihaz adları....200
sürücüler....358
grafiksel....357
etki....356
lazer....357
printf komutu....342, 499
yazdırma....
sistem durumunu belirleme....365
geçmişi....355
İnternet Yazdırma Protokolü....365
tek aralıklı yazı tipleri....356
metin hazırlama....358
güzel....362
yazdırma kuyrukları....365
orantılı yazı tipleri....357
kuyruk....366
yervir....365
yazdırma işlerini sonlandırma....367
işleri görüntüleme....366
işlem kimliği....115
işlem ikamesi....543
işlemler....114
arka plan....122
çocuk....115
kontrol....120
ön plan....122
kesintiye uğratmak....122
iş kontrolü....122
güzel....116
ebeveyn....115
PID....115
işlem kimliği....115
SIGINT....552
sinyaller....126
SIGTERM....552
uyku....116
durum....116
durma....123
görüntüleme....115, 118
zombi....116
üretim kullanımı....462
programlanabilir tamamlama....88
ps komutu....115
PS1 değişkeni....137, 171

PS2 değişkeni....391
ps2pdf komutu....35
PS3 değişkeni....447
PS4 değişkeni....466
sözde kod....412, 442
pstree komutu....13
PuTTY....231
pwd komutu....8
PWD değişkeni....13
Python programlama dili....369

Q

alıntı....78
  çift tırnak....79
  kaçış karakteri....81
  eksik tırnak....454
  tek tırnak....81

R

RAID (Bağımsız Disklerin Yedekli Dizisi)
....193
RANDOM değişkeni....531
raster görüntü işlemcisi....35
okuma komutu....430, 440, 451, 461, 529, 543
Readline....84
README....52, 373
yeniden başlatma komutu....13
yönlendirme....
  engellenen boru....558
  grup komutları ve alt kabuklar....538
  burada belgeler....397
  burada dizeler....436
  standart hata....59
  standart giriş....61, 399
  standart çıktı....57
yönlendirme operatörleri....
  &>....60
  &>>....61
  <....63
  <(liste)....543
  <<....397, 399
  <<-....399
  <<<....436
  >....57
  >(liste)....543
  >>....58
  |....63
düzenli ifadeler....66, 267, 323, 421, 435
  bağlantılar....271
  geri referanslar....288, 322 f.
  temel....278, 287 f., 320, 324, 334
  genişletilmiş....278
ilişkisel veritabanları....308
göreceli yol adları....9
erken yayınla, sık sık yayınla....46
bir dosyadaki yinelenen satırları kaldırma....65
renice komutu....125
REPLY değişkeni....430, 447, 543
rapor oluşturucu....38
depolar....183
geri dönüş komutu....404, 417
yeniden kullanılabilir....411
rev komutu....310
RIP....358
rlogin komutu....225
rm komutu....33
Rock Ridge uzantıları....211
roff....347
ROT13 kodlaması....317
RPM....182
rpm komutu....185
rsync komutu....261
rsync uzaktan güncelleme protokolü....261
Ruby programlama dili....369

S

skaler değişkenler....52
Schilling, Jorg....21
scp komutu....229
script komutu....92
komut dosyası dilleri....44, 369
sdiff komutu....332
bir dosyada desen arama....66
geçmişi arama....89
Güvenli Kabuk....225
sed komutu....318, 351, 505
select kabuk yerleşik komutu....447
set -e....459, 553
set -o pipefail....459
set -u....459
set komutu....134, 465
setgid....104
setuid....104, 416
Seward, Julian....252
sftp komutu....230
sha256sum komutu....213
paylaşımlı kütüphaneler....21, 184
shebang....383, 388
kabuk yerleşikleri....44

kabuk işlevleri 44, 134, 403, 481
kabuk istemleri 2, 9, 90, 106, 121, 137, 171, 226, 391
kabuk betikleri 382
SHELL değişkeni 137
kabuk değişkenleri 13
shellcheck 460
shift komutu 479, 485
shuf komutu 531
kapatma komutu 13
SIGINT 552
sinyaller 551
tek tırnak 81
Slackware 182
uyku komutu 445
dilim 531
snap paket biçimi 190
yumuşak bağlantı 24
sıralama komutu 65, 293
sıralama anahtarları 296
kaynak kodu .....182 f., 191, 290, 368
kaynak komutu 148, 385
kaynak ağacı 372
özel parametreler 481, 502
bölme komutu 332
SSH 22
ssh komutu 226, 258
ssh programı 93
Stallman, Richard .....xix, xxii, 144, 279, 370
standart hata .....56, 58 f.
  imha etmek 61
  bir dosyaya yönlendirme 59
standart giriş 57, 399, 430
  yönlendirme 61
standart çıktı 56
  dosyaya ekleme 58
  imha etme 61
  standart hatayı yeniden yönlendirmek 60
  bir dosyaya yönlendirme 57
başlangıç dosyaları 137
stat komutu 246
statik bağlama 190
yapışkan bit 104
depolama aygıtları 193
  ses CD'leri 211
  CD-ROM 211
  cihaz adı 199
  disk bölümleri 194
  disketler 200
  bağlantı noktaları 195, 197
  bölümler 204
  doğrudan okuma ve yazma 210
  dosya sistemlerini onarma 209
  bağlantıyı kesme 198
  USB flash sürücüler 210
akış düzenleyici 318
diziler .....
  ifadeler 418
  bir kısmını ayıklama 504
  uzunluk 503
  arama ve değiştirme işlemi gerçekleştir 505
  başındaki kısmı kaldır 504
  son kısmı kaldır 505
  ${parameter:offset:length} 504
  ${parameter:offset} 504
strings komutu 495
stubs 408, 462
stil 373
su komutu 105
alt kabuklar 436, 538
sudo komutu 105, 107
Sun Microsystems 151
süper kullanıcı 3, 95, 106, 130
sembolik bağlantılar 24, 35, 39
  oluşturma 39, 42
  listeleme 40
sembolik bağlantı 24
sözdizimi hataları 453
sözdizimi vurgulaması 382, 387
systemd 114

T

tablolar 308
tablo verileri 296, 345
tac komutu 310
tail komutu 67
bant arşivi 254
tar komutu 254
tarball 372
hedefler 376
Görev Yöneticisi 120
Tatham, Simon 231
tbl komutu 347, 350
tee komutu 69
teletype 115
telnet komutu 225
TERM değişkeni 137
terminal emülatörleri 2
terminal oturumları .....
  kontrol terminali 115
  .bashrc'nin etkisi 385
  ortam 106

çıkış....5
giriş kabuğu....106, 137
TERM değişkeni....137
adlandırılmış borular kullanma....55
sanal....5
uzaktan sistemlerle....93
terminaller....86, 92 f., 175, 347, 356
üçlü operatör....516
test durumları....463
test komutu....415, 421, 456
test kapsamı....463
test....462 f.
TEX....347
metin....17
satır uzunluğunu ayarlama....337 f.
ASCII....17
satır başı....292
karşılaştırma....311
MS-DOS'u Unix'e dönüştürme....317
kelime sayma....66
kesme....302
yinelenen satırları silme....30
birden fazla boş satırı silme....292
farklılıkları algılama....312
ortak satırları görüntüleme....311
kontrol karakterlerini görüntüleme....291
DOS formatı....292
EDITOR değişkeni....137
editörler....143, 289
genişleyen sekmeler....305
dosya....17
filtreleme....65
katlama....337 f.
biçimlendirme....333
dizgiciler için biçimlendirme....347
tabloların biçimlendirilmesi....350
birleştirme....307
satır besleme karakteri....292
küçük harfleri büyük harfe dönüştürme....316
satır numaralandırma....292, 333
sayfa numaralandırma....341
yapıştırma....306
baskıya hazırlık....358
yinelenen satırları kaldırma....65
PostScript'te görüntüleme....349
ROT13 kodlu....317
desen arama....66
sıralama....65, 293
yazım denetimi....327
değiştirme....322
boşlukları sekmelerle değiştirme....305
sekme ile ayrılmış....304
karakterleri çevirmek....316
Unix formatı....292
less ile görüntüleme....17, 64
metin editörleri....143, 289, 316
emacs....144
kabuk betikleri yazmak için....382
gedit....143, 382
etkileşimli....316
kate....143, 382
kwrite....143
satır....151
nano....143, 150
piko....143
akış....318
sözdizimi vurgulaması....382, 387
vi....143
vim....143, 382, 387
görsel....151
tilde genişletme....73, 79
tload komutu....131
top komutu....118
yukarıdan aşağıya tasarım....401
Torvalds, Linus....xvi, xxi
touch komutu....245 f., 263, 377, 489 tr
komutu....316
traceroute komutu....218
izleme....464
karakterlerin transliterasyonu....316
tuzaklar....551
troff komutu....347
true komutu....414
TTY....115
komut yaz....45
dizgi makineleri....347, 357
TZ değişkeni....137

U

Ubuntu....95, 108, 182, 275, 385
umask komutu....102, 111
umount komutu....19
unalias komutu....54
tekli işleç bekleniyor....45
tekli operatörler....510
unexpand komutu....305
beklenmeyen belirteç....455
uniq komutu....65, 300
Unix....xvii
Unix Sistem V....360
unix2dos komutu....292

unset komutu....53
until bileşik komutu....450
until döngüsü....450
unzip komutu....259
updatedb komutu....234
yukarı akış sağlayıcıları....183
çalışma süresi....402
çalışma süresi komutu....4, 409
USB flash sürücüler....193, 210
Usenet....31
KULLANICI değişkeni....135, 137
kullanıcılar....
..
hesaplar....94
kimlik değiştirme....105
şifre değiştirme....111
etkin kullanıcı kimliği....104, 115
ana dizin....95
kimlik....94
şifre....95
varsayılan izinleri ayarlama....102
setuid....104
süper kullanıcı....95, 98, 104 f., 112
/etc/passwd....95
/etc/shadow....95
UUID....195

V

girişi doğrulama....437
değişken....77, 392, 500
değer atama....395, 511
sabitler....394
bildirme....393, 395
ortam....134
küresel....405
yerel....405
isim....394, 503
skaler....523
kabuk....134
vi komutu....150
vi tarzı komut satırı düzenleme....169
vim komutu....287, 387
sanal konsollar....5
sanal özel ağ....229
sanal terminaller....5
görsel editörler....151
vmstat komutu....131

W

wait komutu....55
wc komutu....6
web sayfaları....290
wget komutu....224
Gördüğünüz Neyse O....356
whatis komutu....50
which komutu....45, 77
while bileşik komut....443
joker karakterler....27, 62, 71, 267, 274
wodim komutu....213
kelime bölme....79 f.
Wordle....547
dünya....94
WYSIWYG....356

X

X Pencere Sistemi....229
xargs komutu....243
xload komutu....131
xlogo komutu....120
XML....290

Y

metin çekme....85
yum komutu....185

Z

zgrep komutu....287
zip komutu....259
zless komutu....52

mkfs komutu....207

-

--help seçeneği....47

.

./configure....374
.bash_history....88
.bash_login....138
.bash_profile....138 f.
.bashrc....138, 143, 385, 411, 481
.profile....138
.ssh/bilinen_ana_bilgisayarlar....228

(

(( )) bileşik komut 509, 515

[

[komut 455
[[]] bileşik komut 443

/

/ 20
/bin 20
/boot 20
/boot/grub/grub.cfg 20
/boot/vmlinuz 20
/dev 20
/dev/cdrom 200
/dev/dvd 200
/dev/floppy 200
/dev/null 61
/etc 21
/etc/bash.bashrc 138
/etc/crontab 21
/etc/fstab 21, 194, 209
/etc/group 95
/etc/passwd 21, 95, 299, 305, 435
/etc/profile 138
/etc/shadow 95
/etc/sudoers 105
/lib 21
/lost+found 21
/media 21
/mnt 21
/opt 21
/proc 22
/root 22, 106
/run 22
/sbin 22
/sys 22
/tmp 22, 555
/usr 22
/usr/bin 22
/usr/lib 22
/usr/local 23
/usr/local/bin 23, 378, 386
/usr/local/sbin 386
/usr/sbin 23
/usr/share 23
/usr/share/dict 272
/usr/share/doc 23, 52
/var 23
/var/log 23
/var/log/mesajlar 23, 200
/var/log/syslog 23, 68, 200

~

~/.config 23
~/.local 23

$

$! 556
$((ifade)) 509
${!dizi[@]} 528
${!dizi[*]} 528
${!prefix@} 503
${!prefix*} 503
${#parameter} 503
${parameter,,pattern} 508
${parametre,desen} 508
${parametre:-kelime} 501
${parameter:?word} 502
${parameter:+kelime} 503
${parameter:=kelime} 502
${parameter//pattern/string} 505
${parameter/#pattern/string} 505
${parametre/%desen/dize} 505
${parameter/pattern/string} 505
${parameter##pattern} 504
${parameter#pattern} 504
${parameter%%pattern} 505
${parameter%pattern} 505
${parameter^^pattern} 508
${parameter^pattern} 508
$@ 482, 492
$* 482, 492
$# 477
0 481

Kolofon

Bu kitap, Ubuntu 8.04 ile fabrika ayarlarında yapılandırılmış bir Dell Inspiron 530N bilgisayarda OpenOffice.org Writer kullanılarak Liberation Serif ve Sans yazı tipleriyle yazılmıştır. Metnin PDF versiyonu doğrudan OpenOffice.org Writer tarafından oluşturulmuştur. İkinci İnternet Sürümü, Ubuntu 12.04 üzerinde LibreOffice Writer kullanılarak aynı bilgisayarda üretilmiştir. Üçüncü ve sonraki İnternet Sürümleri, Ubuntu LTS (sürüm 14.04-24.04) çalıştıran bir System76 Ratel Pro bilgisayarda LibreOffice Writer ile üretilmiştir. %100 geri dönüştürülmüş elektronlarla basılmıştır.