BAŞLANGIÇ EĞİTİMİ · 3 GÜN · 18 SAAT

Linux Kavramları

Nazım KOÇ
nazim.koc@gmail.com
https://UCanLinux.Com
Sürüm 2026.07.23

Bu Eğitim Hakkında

01 Tamamen **terminal** üzerinde ilerler; grafik arayüz (GUI) hiç kullanılmaz.

02 Belirli bir **dağıtımdan bağımsızdır**; kavramlar her Linux'ta geçerlidir.

03 Örnekler çok **basit C ve bash** programlarıyla verilir.

04 Amaç: sistemin nasıl çalıştığını **kavramsal** olarak anlamaktır.

Nazım KOÇ · nazim.koc@gmail.com

GNU

GNU, "GNU's Not Unix" ifadesinin kısaltmasıdır (özyineli kısaltma). 1983'te Richard Stallman tarafından başlatılan, tamamen **özgür yazılımdan** oluşan bir işletim sistemi projesidir.

Amacı Unix ile uyumlu ama özgür bir sistem üretmekti. Bu proje çekirdek dışındaki tüm parçaları üretti:

- Derleyici: `gcc`
- Kabuk: `bash`
- Temel araçlar: `ls, cp, mv, cat, grep, coreutils`
- C kütüphanesi: `glibc`

GNU/Linux

Linux aslında yalnızca **çekirdektir** (kernel). 1991'de Linus Torvalds tarafından yazılmıştır. Tek başına çekirdek bir iş yapmaz.

GNU projesinin araçları ile Linux çekirdeği birleştiğinde kullanılabilir bir işletim sistemi ortaya çıkar. Bu yüzden doğru ad çoğu zaman **GNU/Linux**'tur.

`GNU araçları` + `Linux çekirdeği` = `Sistem`

Nazım KOÇ · nazim.koc@gmail.com

login

login, sisteme kullanıcı adı ve parola ile giriş yapma işlemidir. Kullanıcının kimliği doğrulanır ve ona bir oturum açılır.

- Kullanıcı bilgileri `/etc/passwd` içinde tutulur.
- Parolalar `/etc/shadow` içinde şifreli saklanır.
- Giriş başarılı olunca kullanıcının kabuğu (shell) başlatılır.

```
login: nazim
Password: ********
nazim@linux:~$ whoami
nazim
```

mod (erişim izinleri)

Her dosyanın bir **erişim modu** vardır: okuma (r), yazma (w), çalıştırma (x). Bu izinler üç grup için ayrı tutulur: **sahip**, **grup**, **diğerleri**.

```
$ ls -l deneme.sh
 -rwxr-xr--  1 nazim nazim  42  deneme.sh
 |└┘└┘└┘
 | |  |  └─ diğerleri: r--
 | |  └──── grup:      r-x
 | └─────── sahip:     rwx
 └────────── dosya tipi
```

Sayısal karşılık: r=4, w=2, x=1. Yani `rwxr-xr--` = **754**.

owner (sahiplik)

Her dosyanın bir **sahibi** (owner) ve bir **grubu** vardır. Sistem bunları isimle değil sayılarla tutar: `UID` (kullanıcı no) ve `GID` (grup no).

- Sahiplik, izinlerin kime uygulanacağını belirler.
- Sahibi değiştirmek için `chown` komutu kullanılır.
- `root` kullanıcısının UID değeri her zaman 0'dır.

```
$ id
uid=1000(nazim) gid=1000(nazim) groups=1000(nazim),27(sudo)
```

console (konsol)

Konsol, çekirdeğin doğrudan bağlı olduğu fiziksel giriş/çıkış birimidir: klavye ve ekran. Sistem mesajları buraya yazılır.

Linux birden çok **sanal konsol** sunar. Bunlar arasında geçiş yapılabilir:

```
Ctrl + Alt + F1   → tty1
Ctrl + Alt + F2   → tty2
...
$ tty
/dev/tty1
```

terminal

Terminal, kullanıcı ile sistem arasında metin tabanlı giriş/çıkışı sağlayan araçtır. Eskiden ayrı bir cihazdı; bugün çoğunlukla bir yazılımdır (terminal emülatörü).

- Terminal sadece metin gösterir ve klavye girişini iletir.
- İçinde çalışan program genellikle bir **kabuktur** (shell, örn. bash).
- Konsol donanımsal, terminal ise mantıksal bir kavramdır.

```
Kullanıcı ⇄ Terminal ⇄ Kabuk ⇄ Çekirdek
```

Çekirdek (kernel)

Çekirdek, donanım ile yazılım arasındaki köprüdür. Programlar donanıma doğrudan değil, çekirdek üzerinden erişir.

Başlıca görevleri:

- Süreç (process) yönetimi
- Dosya sistemi yönetimi
- Ağ iletişimi
- Bellek yönetimi
- Aygıt sürücüleri
- Sistem çağrıları (system call)

Kök Dosya Sistemi

Linux'ta tüm dosyalar **tek bir ağaç** altında toplanır. Bu ağacın tepesi **kök** dizinidir: /

Windows'taki gibi C:, D: sürücü harfleri yoktur. Diğer diskler ve aygıtlar bu ağaca **bağlanır** (mount).

```
/
├── bin/    ├── etc/    ├── home/
├── dev/    ├── proc/   ├── usr/
├── var/    ├── tmp/    └── lib/
```

Standart Akışlar ve Yönlendirme

Her programın üç standart akışı vardır:

`0 stdin` standart giriş (klavye)

`1 stdout` standart çıkış (ekran)

`2 stderr` standart hata (ekran)

Yönlendirme ile bu akışlar dosyalara bağlanır:

```
ls > liste.txt      # çıkışı dosyaya yaz (üzerine)
ls >> liste.txt     # çıkışı dosyaya ekle
wc -l < liste.txt   # dosyadan giriş oku
```

Hata Yönlendirme ve Borular

Hata akışı (2) ayrıca yönlendirilebilir; iki akış birleştirilebilir:

```
komut 2> hata.txt        # sadece hatayı dosyaya
komut  > cikti.txt 2>&1  # çıkış+hata aynı dosyaya
komut &> hepsi.txt       # kısayolu (bash)
komut  > /dev/null 2>&1  # hiçbir şey görmek istemiyorum
```

Boru (pipe) bir programın çıkışını diğerinin girişine bağlar:

```
cat /etc/passwd | grep bash | wc -l
```

Önemli Dizinler ve Amaçları

`/bin`	Temel çalıştırılabilir komutlar (ls, cp, cat). Sistem açılışında gerekir.
`/etc`	Sistem geneli yapılandırma dosyaları. Tamamı metindir.
`/var`	Değişken veriler: kayıtlar (log), kuyruklar, önbellek.
`/dev`	Aygıt dosyaları. Donanım burada dosya gibi görünür.
`/proc`	Çekirdek ve süreç bilgisi. Diskte değil, bellekte üretilir.

Man Sayfaları Nedir?

man (manual), Linux'un yerleşik kılavuz sistemidir. Neredeyse her komut ve sistem çağrısının bir man sayfası vardır. İnternete gerek kalmadan belgeye erişilir.

```
$ man ls
```

Açılan sayfada gezinme: `Boşluk` ileri, b geri, `/kelime` arama, q çıkış.

Bölümler (Sections)

Man sayfaları numaralı bölümlere ayrılır. Aynı isim farklı bölümlerde olabilir:

1 Kullanıcı komutları (ls, cp)

2 Sistem çağrıları (open, read)

3 Kütüphane fonksiyonları (printf)

5 Dosya biçimleri (/etc/passwd)

8 Sistem yönetimi komutları (mount)

```
$ man 2 write     # write sistem çağrısı
$ man 1 write     # write komutu
```

Sayfa Bölümleri ve Arama

Her man sayfası benzer başlıklar içerir: **NAME, SYNOPSIS, DESCRIPTION, OPTIONS, EXAMPLES, SEE ALSO**.

Doğru sayfayı bulmak için:

```
$ man -k kopyala       # anahtar kelimeyle ara (apropos)
$ whatis ls            # tek satırlık özet
$ man -f printf        # hangi bölümlerde var?
```

Kullanıcı ve Kimlik Dosyaları

`passwd` Kullanıcı hesapları: ad, UID, GID, ev dizini, kabuk.

`shadow` Şifrelenmiş parolalar (sadece root okur).

`group` Grup tanımları ve üyeleri.

```
$ cat /etc/passwd
nazim:x:1000:1000:Nazim Koc:/home/nazim:/bin/bash
|     | |    |    |         |           └ kabuk
|     | |    |    |         └ ev dizini
|     | |    |    └ açıklama (GECOS)
|     | |    └ GID
|     | └ UID
|     └ parola (x → /etc/shadow)
└ kullanıcı adı
```

Sistem ve Ağ Dosyaları

`fstab`	Açılışta hangi dosya sistemlerinin nereye bağlanacağı.
`hostname`	Makinenin adı.
`hosts`	İsim → IP eşleşmeleri (yerel).
`resolv.conf`	DNS sunucu adresleri.
`crontab`	Zamanlanmış görevler.
`os-release`	Dağıtım adı ve sürüm bilgisi.

Açılış Zinciri ve init

Çekirdek yüklendikten sonra ilk kullanıcı sürecini başlatır. Bu sürecin **PID değeri her zaman 1**'dir ve tüm diğer süreçlerin atasıdır.

```
Güç → BIOS/UEFI → Bootloader → Çekirdek → init (PID 1) → servisler
```

init sistemi, hangi programların otomatik başlatılacağını belirler. En yaygını bugün `systemd`; klasik olan `SysV init`'tir.

Shell Nedir?

Kabuk (shell), kullanıcının yazdığı komutları okuyup çalıştıran bir programdır. Sistemin özel bir parçası değildir; sadece başka bir programdır (örn. `bash`).

Görevleri:

- Komut satırını okumak ve ayrıştırmak
- Programı bulup çalıştırmak
- Yönlendirme ve boruları kurmak
- Değişkenleri ve ortamı yönetmek
- Betik (script) çalıştırmak

PATH

PATH, kabuğun bir komutu ararken bakacağı dizinlerin listesidir. Dizinler iki nokta üst üste (`:`) ile ayrılır.

```
$ echo $PATH
/usr/local/bin:/usr/bin:/bin

$ which ls        # ls hangi dizinde bulundu?
/bin/ls
```

Kendi programınızın çalışması için ya PATH'e ekleyin ya da tam yolunu yazın (`./program`).

ENV — Ortam Değişkenleri

Ortam (environment), çalışan her programa aktarılan ad=değer çiftleridir. Çocuk süreçler bu ortamı miras alır.

```
$ env                # tüm ortam değişkenleri
$ echo $HOME
/home/nazim

$ AD=deneme          # sadece kabukta kalır
$ export AD          # çocuk süreçlere de geçer
```

Yaygın değişkenler: `HOME`, `USER`, `PATH`, `LANG`, `SHELL`.

PS1 — Komut İstemi

PS1, kabuğun gösterdiği komut isteminin (prompt) biçimini belirleyen değişkendir. Özel kaçış dizileri kullanır.

`\u` kullanıcı adı

`\h` makine adı

`\w` çalışılan dizin

`\$` $ (root için #)

```
$ PS1='\u@\h:\w\$ '
nazim@linux:~$
```

$? — Çıkış Kodu

Her komut bittiğinde bir **çıkış kodu** döndürür. $? son komutun kodunu tutar. **0 = başarı**, sıfırdan farklı = hata.

```
$ ls /var
$ echo $?
0

$ ls /yok
ls: /yok: No such file or directory
$ echo $?
2
```

Betiklerde koşul denetimi bu değere dayanır: `if komut; then ... fi`

Diğer Özel Değişkenler

`$$`	Çalışan kabuğun süreç no'su (PID).
`$0`	Betiğin/kabuğun adı.
`$1 $2 …`	Betiğe verilen konumsal argümanlar.
`$#`	Argüman sayısı.
`$@`	Tüm argümanlar (liste).
`$!`	Arka planda başlatılan son sürecin PID'i.
`PWD`	Çalışılan dizin; OLDPWD bir önceki.

~/.bash_profile

Kullanıcı sisteme **giriş yaptığında** (login shell) bir kez çalıştırılan dosyadır. Genellikle ortam değişkenlerini ayarlamak için kullanılır.

```
# ~/.bash_profile
export PATH="$HOME/bin:$PATH"
export EDITOR=vi

# .bashrc'yi de yükle (yaygın alışkanlık)
[ -f ~/.bashrc ] && . ~/.bashrc
```

Bir kez, giriş anında çalışır — her yeni terminalde değil.

~/.bashrc

Etkileşimli her yeni kabuk (yeni terminal penceresi) açıldığında çalışır. Takma adlar (alias), fonksiyonlar ve prompt burada tanımlanır.

```
# ~/.bashrc
alias ll='ls -alF'
alias ..='cd ..'
PS1='\u@\h:\w\$ '

selamla() { echo "Merhaba $1"; }
```

Özet: giriş için **.bash_profile**, her terminal için **.bashrc**.

Sinyaller Nedir?

Sinyal, bir sürece gönderilen kısa, asenkron bir bildirimdir. Çekirdek veya başka bir süreç, "bir olay oldu" demek için sinyal gönderir. Yazılımsal bir kesme gibidir. Bak $ man 7 signal

Süreç bir sinyal aldığında üç şey olabilir:

- Varsayılan davranışı uygular (çoğunlukla sonlanma)
- Sinyali görmezden gelir (yok sayar)
- Kendi **işleyicisini** (handler) çalıştırır

Sinyal Göndermek

Komut satırından `kill` ile sinyal gönderilir; klavyeden de bazı sinyaller üretilir.

```
$ kill -TERM 1234     # nazikçe sonlan iste
$ kill -9 1234        # zorla öldür (SIGKILL)
$ kill -l             # sinyal listesi

Ctrl+C → SIGINT     Ctrl+Z → SIGTSTP
```

`SIGKILL(9)` yakalanamaz, hemen öldürür

`SIGSTOP(19)` durdurur (yakalanamaz)

Sinyaller Nereden Gelir?

Klavye	Ctrl+C (SIGINT), Ctrl+Z (SIGTSTP), Ctrl+\ (SIGQUIT).
Çekirdek	Donanım/yazılım hatası: geçersiz bellek → SIGSEGV, sıfıra bölme → SIGFPE.
Başka süreç	kill() sistem çağrısı ile (uygun izin gerekir).
Zamanlayıcı	alarm()/timer → SIGALRM.
Boru	Kapalı boruya yazma → SIGPIPE.
Çocuk	Çocuk süreç bitince → SIGCHLD.

Sık Kullanılan Sinyaller

SIGHUP	1	Terminal koptu / yeniden yükle
SIGINT	2	Kesme (Ctrl+C)
SIGKILL	9	Zorla öldür (yakalanamaz)
SIGSEGV	11	Geçersiz bellek erişimi
SIGTERM	15	Nazik sonlandırma (varsayılan kill)
SIGCHLD	17	Çocuk süreç durdu/bitti

C ile Sinyal Yakalamak

`sigaction()` ile SIGINT (Ctrl+C) yakalanır: Bak signal.c

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

volatile sig_atomic_t calisiyor = 1;

void handler(int sig) {       /* sinyal işleyici */
    calisiyor = 0;            /* döngüyü durdur   */
}

int main(void) {
    struct sigaction sa = {0};
    sa.sa_handler = handler;
    sigaction(SIGINT, &sa, NULL);   /* Ctrl+C'yi bağla */

    while (calisiyor) { printf("calisiyor...\n"); sleep(1); }
    printf("temiz cikis\n");
    return 0;
}
```

Derleme ve Davranış

```
$ gcc -o yakala yakala.c
$ ./yakala
calisiyor...
calisiyor...
^C           ← Ctrl+C basıldı
temiz cikis  ← program aniden ölmedi
```

Önemli noktalar:

- İşleyici içinde yalnızca **async-signal-safe** işler yapılmalı.
- Paylaşılan bayrak `volatile sig_atomic_t` olmalı.
- SIGKILL ve SIGSTOP asla yakalanamaz.

bash ile Sinyal Yakalamak: trap

bash'te sinyaller `trap` yerleşik komutuyla yakalanır: Bak yakala.sh

```
#!/bin/bash

temizle() {
    echo "Sinyal alindi, temizlik yapiliyor..."
    rm -f /tmp/kilit
    exit 0
}

trap temizle SIGINT SIGTERM   # bu sinyallerde temizle() çalışsın

touch /tmp/kilit
echo "Calisiyorum (PID $$). Durdurmak icin Ctrl+C."
while true; do
    sleep 1
done
```

trap Ayrıntıları

```
trap 'echo cikiliyor' EXIT   # betik biterken çalışır
trap '' SIGINT               # Ctrl+C'yi yok say
trap - SIGINT                # varsayılana geri dön
trap -p                      # kurulu trap'leri listele
```

Kullanım amacı: betik yarıda kesilse bile geçici dosyaları silmek, kilitleri kaldırmak, kaynakları düzgün serbest bırakmak. Bu, güvenilir betik yazmanın temelidir.

nohup Nedir?

Terminal kapandığında, o terminalde çalışan programlara **SIGHUP** gönderilir ve genelde sonlanırlar. **nohup**, bir programı bu sinyalden koruyarak başlatır.

```
$ nohup ./uzun_islem &
[1] 4567
nohup: ignoring input and appending output to 'nohup.out'
```

Program artık terminal kapansa bile çalışmaya devam eder. Çıktı otomatik olarak `nohup.out` dosyasına yazılır.

Pratik Kullanım

```
# Çıktıyı kendi dosyana yönlendir
$ nohup ./sunucu > sunucu.log 2>&1 &

# Hangi işler arka planda?
$ jobs
$ ps -ef | grep sunucu
```

İlgili kavramlar:

- & — programı arka planda başlatır.

tree

Dizin yapısını **ağaç görünümünde** listeler. İç içe klasörleri bir bakışta görmek için kullanılır.

```
$ tree -L 2 proje/
proje/
├── src/
│   ├── main.c
│   └── util.c
├── Makefile
└── README

$ tree -d       # sadece dizinler
$ tree -a       # gizli dosyalar dahil
```

dmesg

Çekirdek mesajlarını gösterir. Açılış olayları, aygıt takılması/çıkarılması ve donanım hataları buradan izlenir.

```
$ sudo dmesg | tail            # son mesajlar
$ sudo dmesg -w                # canlı izle (yeni gelenler), usb memory tak ve çıkar
$ sudo dmesg | grep -i usb     # USB olaylarını süz
[  12.3] usb 1-1: new high-speed USB device
[  12.4] sd 0:0:0:0: [sda] Attached SCSI disk
$ sudo dmesg -l err,warn

$ ls -l /dev/kmsg
$ sudo -s
echo "merhaba" > /dev/kmsg
```

Yeni bir cihaz taktığınızda ne olduğunu anlamanın en hızlı yoludur. Çekirdek mesajları, /dev/kmsg dosyası ile iletilir.

tail

Bir dosyanın **son satırlarını** gösterir. Kayıt (log) dosyalarını izlemek için vazgeçilmezdir.

```
$ tail dosya.log          # son 10 satır
$ tail -n 50 dosya.log    # son 50 satır
$ tail -f /var/log/syslog # canlı takip (follow)
```

`-f` seçeneği dosyaya yeni eklenen satırları anlık gösterir; kardeşi `head` ilk satırları verir.

chmod

Dosya **izinlerini değiştirir**. İki yazım vardır: sembolik ve sayısal (octal).

```
$ chmod +x betik.sh        # çalıştırma izni ekle
$ chmod u+w,o-r dosya       # sahibe yazma, diğerine -okuma
$ chmod 755 program        # rwxr-xr-x
$ chmod -R 644 dizin/      # alt dosyalara da uygula
```

Sayısal: sahip-grup-diğer sırasıyla, r=4 w=2 x=1 toplamı.

chown

Bir dosyanın **sahibini ve grubunu** değiştirir. Genellikle root yetkisi gerekir.

```
$ chown nazim dosya          # sahibi nazim yap
$ chown nazim:gelistirici d  # sahip + grup
$ chown :gelistirici dosya   # sadece grup
$ chown -R nazim /srv/uyg    # dizin ağacına uygula
```

Sadece grubu değiştirmek için `chgrp` de kullanılabilir.

ifconfig

Ağ arayüzlerinin adreslerini gösterir ve ayarlar. (Modern sistemlerde yerini `ip` komutu almıştır.)

```
$ ifconfig
eth0: inet 192.168.1.20  netmask 255.255.255.0
      ether 08:00:27:1a:2b:3c

$ ifconfig eth0 up               # arayüzü aç
$ ifconfig eth0 192.168.1.50     # IP ata

# Modern eşdeğeri:
$ ip addr show
$ ip link set eth0 up
```

cpio

Dosya listelerini bir **arşive** paketler/çıkarır. Dosya adlarını standart girişten okur; bu yüzden çoğu zaman `find` ile eşleşir. Linux'un **initramfs** imajları cpio biçimindedir.

-o: copy out
-i: copy in
t: content list

```
# Arşiv oluştur (create)
$ find . | cpio -o > arsiv.cpio

# İçeriği listele (list)
$ cpio -it < arsiv.cpio

# Çıkart (extract)
$ cpio -id < arsiv.cpio
```

strace

Bir programın yaptığı **sistem çağrılarını** canlı olarak gösterir. "Program neden çalışmıyor / hangi dosyayı açamadı?" sorularının cevabı için birebirdir.

```
$ strace ./program
open("/etc/ayar", O_RDONLY) = -1 ENOENT (yok)
write(1, "merhaba\n", 8)        = 8

$ strace -e trace=open,read ./program   # sadece bunları
$ strace -p 1234                         # çalışan sürece bağlan
$ strace -c ./program                    # çağrı özeti/istatistik
```

awk — Alan Tabanlı İşleme

awk satırları **alanlara** böler (varsayılan ayraç boşluk). Her alana `$1`, `$2` … ile erişilir; `$0` tüm satır, NF alan sayısı, NR satır no.

```
$ cat veri.txt

# 1. alanı yazdır
$ awk '{ print $1 }' veri.txt

# 1. ve 3. alan birlikte
$ awk '{ print $1, $3 }' veri.txt

# ayracı ':' yap → /etc/passwd
$ awk -F: '{ print $1 }' /etc/passwd
```

awk — BEGIN, END ve Hesap

BEGIN bloğu ilk satırdan önce, END bloğu son satırdan sonra çalışır. Değişkenlerle hesap yapılır.

```
# 2. sütunun toplamı
$ awk '{ toplam += $2 } END { print toplam }' veri.txt

# başlık + satırları numarala
$ awk 'BEGIN { print "SIRA AD" }
       { print NR, $1 }' veri.txt
```

awk küçük ama tam bir programlama dilidir: koşul, döngü ve dizileri vardır.

awk — Felsefesi

```
BEGIN { bir kez işler, ilk değerler atanır }


+--> line= read_from_file();                  while(line= read_file(veri.txt){
|    $1, $2, ... $N = line.parse();              $1, $2, $3..., $N= line.parse();
|    NR: number of record atanır                 NR= 1
|    NF: number of field atanır                  NR= 5
|
|    kullanıcı kodu                              print { $1,$3 }
+-------                                        }

END { bir kez işler, rapor yazılır }

örnek:
awk '{ print $1, $3 }' veri.txt
```

sed — Akış Editörü

sed, metni satır satır işleyip dönüştürür. En sık kullanılan işlem **yerine koymadır** (substitute): `s/eski/yeni/`

```
# ilk eşleşmeyi değiştir
$ sed 's/kirmizi/mavi/' dosya.txt

# satırdaki TÜM eşleşmeleri (g = global)
$ sed 's/ /_/g' dosya.txt

# dosyanın kendisini düzenle (in-place)
$ sed -i 's/hata/HATA/g' kayit.log

$ sed 's/000/,000/' veri.txt
```

sed — Satır Seçme ve Silme

```
# 3. satırı sil
$ sed '3d' dosya.txt

# 2 ile 5 arası satırları sil
$ sed '2,5d' dosya.txt

# sadece 10-20 arası satırları yaz (-n = sessiz)
$ sed -n '10,20p' dosya.txt

# 'HATA' geçen satırları sil
$ sed '/HATA/d' kayit.log
```

Adres (satır no veya desen) + komut (d sil, p yaz, s değiştir) yapısı sed'in temelidir.

Regular Expression — Temeller

Düzenli ifade (regex), metin içinde **desen** aramanın dilidir. grep, sed ve awk hepsi bunu kullanır.

`.`	herhangi tek karakter
`*`	öncekinden 0 veya daha fazla
`^ $`	satır başı / satır sonu
`[abc]`	a, b veya c
`[0-9]`	bir rakam
`[^x]`	x olmayan karakter

Genişletilmiş Regex ve Örnekler

`grep -E` ile **+** (bir/çok), **?** (0/1), **{n}** (tam n kez) ve **|** (veya) kullanılabilir.

```
# 'hata' veya 'uyari' içeren satırlar
$ grep -E 'hata|uyari' kayit.log

# basit e-posta deseni
$ grep -E '[a-z]+@[a-z]+\.[a-z]+' liste.txt

# tam IP benzeri desen
$ grep -E '([0-9]{1,3}\.){3}[0-9]{1,3}' ag.log
```

Kaçış: özel karakteri düz almak için önüne \ konur (örn. `\.` gerçek nokta).

dd Nedir?

dd, veriyi bir kaynaktan bir hedefe **blok blok, ham bayt** düzeyinde kopyalar. Dosya sistemini umursamaz; diskin/dosyanın içeriğini olduğu gibi taşır.

`if=`	girdi dosyası (input file)
`of=`	çıktı dosyası (output file)
`bs=`	blok boyutu (block size)
`count=`	kaç blok kopyalanacak

Dikkat: yanlış `of=` tüm diski siler. dd affetmez. Device Destroy olmasın.

Neden Ham Kopya?

Normal cp dosyaları kopyalar; dd ise **her baytı**, boş alan ve bölümleme tablosu dahil kopyalar. Bu yüzden:

- Bir diskin **bire bir kopyasını** (klon) çıkarabilir.
- Önyükleme sektörünü ve bölümleri korur.
- Silinmiş veri kurtarma / adli inceleme için kullanılır.

```
# diskin tamamını başka diske klonla
$ dd if=/dev/sda of=/dev/sdb bs=4M status=progress
```

Temel Örnekler

```
# USB'yi imaja yedekle
$ dd if=/dev/sdb of=yedek.img bs=4M status=progress

# imajı USB'ye geri yaz
$ dd if=yedek.img of=/dev/sdb bs=4M

# 100 MB'lik boş dosya üret
$ dd if=/dev/zero of=bos.img bs=1M count=100

# diskin ilk 512 baytını (MBR) al
$ dd if=/dev/sda of=mbr.bin bs=512 count=1
```

İnce Ayarlar

`bs=4M`	büyük blok → daha hızlı
`skip=N`	girişte N blok atla
`seek=N`	çıkışta N blok atla
`conv=sync`	eksik blokları sıfırla doldur
`conv=noerror`	hatada durma (bozuk disk)

Boş Alandan Dosya Sistemi

dd ile üretilen ham alan başta boştur. Kullanmak için üzerine bir **dosya sistemi** kurulur (`mkfs`).

```
# 1) 64 MB'lik boş dosya
$ dd if=/dev/zero of=disk.img bs=1M count=64

# 2) içine ext4 dosya sistemi kur
$ mkfs.ext4 disk.img
mke2fs 1.46
Creating filesystem with 65536 1k blocks...

# artık bağlanmaya (mount) hazır

# örnek için bak dd.txt
```

Farklı Dosya Sistemleri

```
$ mkfs.vfat disk.img        # FAT (taşınabilir, USB)
$ mkfs.ext2 disk.img        # basit, günlüksüz
$ mkfs.ext4 -L VERI disk.img # etiketli ext4

# dosya sistemini denetle
$ fsck.ext4 -f disk.img
```

Doğrulama: `file disk.img` komutu içindeki dosya sistemi tipini söyler.

Bir Cihazdan İmaj Almak

Bir SD kart veya diskin tam kopyası (imajı) tek komutla alınır. Boyutu diskin tamamı kadar olur.

```
# kartı imaja al
$ dd if=/dev/mmcblk0 of=kart.img bs=4M status=progress

# yer kazanmak için sıkıştır
$ dd if=/dev/mmcblk0 bs=4M | gzip > kart.img.gz
```

İmajı geri yazarken hedef diskin en az kaynak kadar büyük olması gerekir.

İmaj Dosyasını Bağlamak

Bir imaj dosyası, **loop** aygıtı üzerinden gerçek disk gibi bağlanabilir:

```
# tek dosya-sistemli imaj
$ mkdir /mnt/imaj
$ mount -o loop disk.img /mnt/imaj
$ ls /mnt/imaj

# işiniz bitince
$ umount /mnt/imaj
```

İmaj içinde bölümler varsa `offset=` ya da `losetup -P` gerekir (sonraki bölümde).

Master (Altın) İmaj Nedir?

Bir cihaz kusursuz kurulup ayarlandıktan sonra, o durumun imajı alınır. Bu **master imaj**, aynı donanıma sahip onlarca cihaza **hızlıca çoğaltılır**.

```
# 1) örnek cihazı hazırla ve kapat
# 2) master imajını al
$ dd if=/dev/mmcblk0 of=master.img bs=4M

# 3) yeni kartlara yaz (seri üretim)
$ dd if=master.img of=/dev/mmcblk0 bs=4M
```

Pratik İpuçları

- İmaj almadan önce **geçici dosyaları ve kayıtları temizleyin** (küçük ve temiz kalsın).
- Sıkıştırılmış master (`.img.gz`) saklama ve dağıtımda yer kazandırır.
- Her cihazda benzersiz olması gerekenler (makine adı, SSH anahtarları) ilk açılışta **yeniden üretilmelidir**.

```
# sıkıştırılmış master'ı doğrudan karta yaz
$ gunzip -c master.img.gz | dd of=/dev/mmcblk0 bs=4M
```

losetup Komutu

losetup, sıradan bir dosyayı bir **loop blok aygıtına** (`/dev/loop0`) bağlar. Böylece dosya, gerçek bir disk gibi işlem görür.

```
# boş loop aygıtı bul ve bağla
$ losetup -f --show disk.img
/dev/loop0

# bağlı loop aygıtlarını listele
$ losetup -a

# bağlantıyı kaldır
$ losetup -d /dev/loop0
```

İmaj Kurma

Sıfırdan kullanılabilir bir imaj üç adımda hazırlanır:

```
# 1) boş alan
$ dd if=/dev/zero of=disk.img bs=1M count=128

# 2) dosya sistemi
$ mkfs.ext4 disk.img

# 3) bağla, doldur, çöz
$ mount -o loop disk.img /mnt/imaj
$ cp -r icerik/* /mnt/imaj/
$ umount /mnt/imaj
```

Sonuçta `disk.img` taşınabilir, kendi kendine yeten bir birimdir.

Sıkıştırma

İmajlar çoğunlukla büyük ama **boş alanı çok** olur. Boşluk sıfırlarla dolu olduğu için sıkıştırma çok etkilidir.

```
# önce boş alanı sıfırla (daha iyi sıkışır)
$ dd if=/dev/zero of=/mnt/imaj/bosluk bs=1M; rm /mnt/imaj/bosluk

# sıkıştır
$ gzip disk.img            # disk.img.gz
$ xz disk.img              # daha küçük, daha yavaş

# geri aç
$ gunzip disk.img.gz
```

Mount Etmek

```
# basit loop mount
$ mount -o loop disk.img /mnt/imaj

# salt-okunur (imajı değiştirme riskini önler)
$ mount -o loop,ro disk.img /mnt/imaj

# dosya sistemi tipini açıkça belirt
$ mount -o loop -t vfat disk.img /mnt/imaj
```

Bağlıyken imajın içindeki dosyalar normal dosya gibi okunur/yazılır. Değişiklikler doğrudan imaj dosyasına işlenir.

İmaj İçinde Bölüm (Partition) Kurmak

Gerçek disk gibi, bir imaj birden çok bölüm içerebilir:

```
# 1) boş imaj + bölüm tablosu
$ dd if=/dev/zero of=disk.img bs=1M count=256
$ fdisk disk.img        # n ile bölüm ekle, w ile yaz

# 2) bölümleri aygıt olarak aç (-P)
$ losetup -Pf --show disk.img
/dev/loop0              # bölümler: loop0p1, loop0p2

# 3) her bölüme fs kur ve bağla
$ mkfs.ext4 /dev/loop0p1
$ mount /dev/loop0p1 /mnt/imaj
```

tmpfs Nedir?

tmpfs, verileri diske değil **RAM'e** (ve gerekirse takas alanına) yazan bir dosya sistemidir. Çok hızlıdır ama **geçicidir**: yeniden başlatınca içerik kaybolur.

- Kalıcı depolama gerektirmeyen geçici veri için idealdir.
- Boyutu esnektir; sadece kullanılan kadar RAM tutar.
- Diski yormadığı için gömülü sistemlerde çok kullanılır.

Sistemde Nerede Kullanılır?

Çoğu dağıtım bazı dizinleri zaten tmpfs olarak bağlar. `df -h` ile görülebilir:

```
$ df -h -t tmpfs
Filesystem   Size  Used  Mounted on
tmpfs        1.6G  1.2M  /run
tmpfs        3.9G  0     /dev/shm
tmpfs        788M  0     /run/user/1000
```

`/run` çalışma zamanı verileri, `/dev/shm` paylaşımlı bellek içindir.

tmpfs'i Elle Bağlamak

```
# 100 MB'lik tmpfs oluştur ve bağla
$ mkdir /mnt/hizli
$ mount -t tmpfs -o size=100M tmpfs /mnt/hizli

# doğrula
$ df -h /mnt/hizli
tmpfs  100M  0  /mnt/hizli

# çöz (içerik kaybolur!)
$ umount /mnt/hizli
```

`size=` üst sınırdır; alanın tamamı baştan RAM'de yer kaplamaz.

Açılışta Otomatik Bağlamak

Kalıcı olması için `/etc/fstab` dosyasına bir satır eklenir:

```
# /etc/fstab
# aygıt  bağlama-noktası  tip    seçenekler       dump pass
tmpfs    /mnt/hizli       tmpfs  size=100M,mode=1777  0    0
```

```
# fstab'ı test et (yeniden başlatmadan)
$ mount /mnt/hizli
```

/dev/shm Nedir?

`/dev/shm`, sistemin varsayılan bir tmpfs bağlama noktasıdır. **POSIX paylaşımlı bellek** (shared memory) burada dosya olarak tutulur.

Farklı süreçler aynı belleği paylaşarak çok hızlı veri alışverişi yapar. Program `shm_open()` çağırdığında burada bir giriş oluşur.

```
$ ls /dev/shm
$ df -h /dev/shm
```

/dev/shm'i Doğrudan Kullanmak

`/dev/shm` sıradan bir dizin gibi de kullanılabilir; çok hızlı, geçici çalışma alanıdır:

```
# büyük veriyi RAM'de işle (disk yormadan)
$ cp buyuk.veri /dev/shm/
$ sort /dev/shm/buyuk.veri > /dev/shm/sirali
$ mv /dev/shm/sirali sonuc.txt
$ rm /dev/shm/buyuk.veri
```

Uyarı: burada tutulan veri RAM kullanır ve yeniden başlatınca silinir.

Neden SD Kartlar Yıpranır?

SD kartlar ve USB bellekler **flash** tabanlıdır. Her hücrenin sınırlı sayıda **yazma/silme** ömrü vardır. Sürekli yazılan dizinler kartı erken bitirir.

Sık yazılan tipik yerler:

`/var/log`	sistem kayıtları
`/tmp`	geçici dosyalar
`/var/tmp`	önbellek, kuyruk

Ömrü Uzatmak: tmpfs Çözümü

Sık yazılan dizinleri RAM'e (tmpfs) taşıyarak flash'a yazma en aza indirilir:

```
# /etc/fstab
tmpfs  /tmp       tmpfs  defaults,size=50M    0  0
tmpfs  /var/log   tmpfs  defaults,size=30M    0  0
```

- Kalması gereken kayıtlar için ara sıra diske özet yazılabilir.
- Dosya sistemini **salt-okunur** bağlamak da ömrü ciddi uzatır.
- Birçok gömülü sistemin standart tasarım kalıbıdır.

/dev/null

"Kara delik" aygıtıdır. Buraya **yazılan her şey yok olur**; buradan **okuma anında dosya sonu** (EOF) verir. İstenmeyen çıktıyı atmak için kullanılır.

```
# çıktıyı tamamen sustur
$ komut > /dev/null

# hata mesajlarını da at
$ komut > /dev/null 2>&1

# sadece hataları görmek için stdout'u at
$ komut 2>&1 >/dev/null
```

/dev/zero

Okundukça **sonsuz sayıda sıfır bayt** (`0x00`) üretir. Boş dosya/alan oluşturmak veya belleği sıfırlamak için kullanılır.

```
# 10 MB'lik sıfır dolu dosya
$ dd if=/dev/zero of=bos.img bs=1M count=10

# tmpfs'i sıfırla (test)
$ head -c 1M /dev/zero > /tmp/deneme
```

Kardeşi `/dev/null` yutar, `/dev/zero` üretir.

/dev/random

Rastgele bayt üretir. Kaynağı sistemin topladığı **entropidir** (donanım gürültüsü, olay zamanlamaları). Klasik davranışta entropi yetersizse **bekler** (bloklar).

```
# 16 rastgele baytı onaltılık yaz
$ head -c 16 /dev/random | xxd
```

Yüksek güvenlik gerektiren anahtar üretiminde tercih edilir; ama bekleme yapabileceği için dikkatli kullanılmalıdır.

xxd geri dönüştürebilir, hexdump dönüştüremez.

/dev/urandom

`/dev/random` gibi rastgele bayt üretir ama **asla beklemez** (unblocked). Entropi havuzunu tohum alıp kriptografik olarak genişletir.

```
# rastgele parola/dosya adı
$ head -c 12 /dev/urandom | base64

# diski rastgele veriyle doldur (silme)
$ dd if=/dev/urandom of=/dev/sdb bs=4M
```

Pratikte önerilen kaynak budur: hızlı ve modern sistemlerde yeterince güvenlidir.

/dev/full

Her zaman "**disk dolu**" hatası (ENOSPC) döndüren aygıttır. Programların "disk doldu" durumunu **doğru ele alıp almadığını test etmek** için kullanılır.

```
$ echo deneme > /dev/full
bash: echo: write error: No space left on device

# okuma ise sıfır verir (/dev/zero gibi)
$ head -c 4 /dev/full | xxd
00000000: 0000 0000
```

C ile /dev/full Örneği

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>

int main(void) {
    int fd = open("/dev/full", O_WRONLY);
    ssize_t n = write(fd, "deneme", 6);      /* yazmayı dene */
    if (n == -1)                             /* hata oldu mu? */
        printf("write hatasi: %s\n", strerror(errno));
    close(fd);
    return 0;
}

// bak full.c
```

vfat (FAT)

Microsoft'un FAT biçiminin Linux desteğidir. Basit ve **her yerde okunur** (Windows, kamera, USB). Ama Unix izinleri, sahiplik ve büyük dosya desteği **yoktur** (FAT32'de dosya sınırı 4 GB).

- Kullanım: USB bellek, SD kart, EFI önyükleme bölümü.
- Kurulum: `mkfs.vfat disk.img`
- Bağlama: `mount -t vfat ...`

ext2

Linux'un klasik dosya sistemidir. Tam Unix izinleri ve sahiplik vardır ama **günlük (journal) yoktur**. Ani güç kesintisinde tutarlılık için `fsck` gerekir.

- Az yazma yaptığı için **flash belleklerde** hâlâ tercih edilir.
- Sadelik ve düşük ek yük ister.
- ext3 ve ext4'ün atasıdır.

ext3

ext2'ye **günlükleme (journaling)** eklenmiş halidir. Yazma işlemleri önce bir günlüğe kaydedilir; kesinti olursa sistem bu günlükten **hızla tutarlı hale gelir**, uzun fsck taramasına gerek kalmaz.

- ext2 ile geriye/ileriye uyumludur.
- Güvenilirliği artırır, biraz yazma yükü ekler.
- Uzun süre masaüstü/sunucu varsayılanı oldu.

ext4

Bugün pek çok dağıtımın **varsayılan** dosya sistemidir. ext3'ün üzerine performans ve kapasite iyileştirmeleri getirir.

- **Extent** tabanlı depolama → büyük dosyalarda daha verimli.
- Çok büyük hacim ve dosya desteği (TB–EB ölçeği).
- Gecikmeli tahsis, daha hızlı fsck, journal.
- Dengeli seçim: hız + güvenilirlik.

NFS

Network File System: bir sunucudaki dizini **ağ üzerinden** başka makinelere bağlatır. Uzak dizin yerel bir klasör gibi görünür.

```
# sunucudaki /veri paylaşımını bağla
$ mount -t nfs 192.168.1.10:/veri /mnt/veri
```

Merkezi dosya paylaşımı ve disksiz istemcilerin köke ağ üzerinden erişmesi için kullanılır.

cramfs

Compressed ROM filesystem: sıkıştırılmış, **salt-okunur** ve çok küçük bir dosya sistemidir. Basit gömülü cihazların ilk açılış imajlarında kullanılırdı.

- Yer kazandırır; belleğe sığdırmak kolaydır.
- Kısıtları çoktur (boyut sınırları, özellik azlığı).
- Bugün büyük ölçüde **squashfs** tarafından değiştirilmiştir.

squashfs

Yüksek oranda **sıkıştırılmış, salt-okunur** bir dosya sistemidir. Live CD/USB, kurtarma ortamları ve paketleme (snap gibi) için standarttır.

```
# dizinden squashfs imajı üret
$ mksquashfs kok/ kok.sqfs -comp xz

# salt-okunur olarak bağla
$ mount -t squashfs -o loop kok.sqfs /mnt/kok
```

Değiştirilebilir üst katman için genellikle **overlayfs** ile birlikte kullanılır.

ubifs

Doğrudan **ham NAND flash** yongaları için tasarlanmış dosya sistemidir (UBI katmanı üzerine kurulur). SD kart/SSD gibi bir denetleyicisi olmayan çıplak flash için kullanılır.

- Yerleşik **aşınma dengeleme** (wear leveling) ve hata düzeltme.
- Sıkıştırma ve güç kesintisine dayanıklılık.
- Gömülü sistemlerde (yönlendirici, IoT) yaygındır.

/dev

Sözde dosya sistemi diskte durmaz; çekirdek tarafından anlık üretilir. /dev altında donanım aygıtları **dosya gibi** görünür (devtmpfs).

```
$ ls -l /dev/sda /dev/tty
brw-rw---- 1 root disk 8, 0 /dev/sda    # b = blok aygıt
crw-rw-rw- 1 root tty  5, 0 /dev/tty    # c = karakter aygıt
```

Baştaki **b**/**c** aygıt türünü; sayı çifti (8,0) sürücü ve alt-aygıt numarasını gösterir.

/sys

sysfs, çekirdeğin aygıt ve sürücü ağacını dosya olarak sunar. Donanım özellikleri okunur; bazıları yazılarak **ayar değiştirilebilir**.

```
# pil durumu
$ cat /sys/class/power_supply/BAT0/capacity
87

# LED'i yak (yazılabilir öznitelik)
$ echo 1 > /sys/class/leds/led0/brightness
```

GPIO, LED, ağ arayüzü gibi pek çok donanım buradan kontrol edilir.

/proc

procfs, çalışan süreçler ve çekirdek durumu hakkında bilgi verir. Her süreç için `/proc/<PID>/` dizini vardır.

```
$ cat /proc/cpuinfo       # işlemci bilgisi
$ cat /proc/meminfo        # bellek durumu
$ cat /proc/1234/status    # 1234 nolu sürecin durumu
$ cat /proc/mounts         # bağlı dosya sistemleri
$ echo 1 > /proc/sys/net/ipv4/ip_forward  # ayar değiştir
```

C ile sysfs GPIO Kullanımı

GPIO pini sysfs'te birer dosyadır; açıp yazmak yeterlidir:

```
#include <stdio.h>

/* pin'i dosyaya yaz: value=1 (yüksek), 0 (düşük) */
void gpio_yaz(const char *yol, const char *deger) {
    FILE *f = fopen(yol, "w");
    if (f) { fprintf(f, "%s", deger); fclose(f); }
}

int main(void) {
    gpio_yaz("/sys/class/gpio/export", "17");            /* pini aç */
    gpio_yaz("/sys/class/gpio/gpio17/direction", "out"); /* çıkış   */
    gpio_yaz("/sys/class/gpio/gpio17/value", "1");        /* yak     */
    return 0;
}

// not: her seferinde fopen()/fclose() yapmaya gerek yoktur.
//      bir kez fopen() ile açtıktan sonra, rewind() ile işlemler yapılabilir
```

Kütüphane Nedir?

Kütüphane (library), birçok program tarafından yeniden kullanılabilen, önceden derlenmiş **fonksiyon topluluğudur**. Her programın tekerleği yeniden icat etmesini önler.

Örneğin `printf`, dosya açma, matematik fonksiyonları hep kütüphanelerden gelir.

```
program.c ──derle──▶ program ──çağırır──▶ libc (printf, malloc...)
```

Neden Kütüphane Kullanılır?

- **Kod paylaşımı:** aynı fonksiyon onlarca programda tekrar yazılmaz.
- **Küçük ikili dosya:** ortak kod bir yerde durur, programlar ince kalır.
- **Merkezi güncelleme:** kütüphanedeki bir güvenlik yaması, tüm programları düzeltir.
- **Bellek tasarrufu:** paylaşımlı kütüphane RAM'e bir kez yüklenip birçok süreçte kullanılır.

Kütüphane Cinsleri

.a — statik

Derleme sırasında programın içine kopyalanır.
Program tek başına çalışır, dış bağımlılık yok.
Dosya büyür.

.so — paylaşımlı

Çalışma anında yüklenir (shared object).
Program ince kalır, kütüphane sistemde ortaktır.

```
libm.a   → statik matematik   |   libm.so → paylaşımlı matematik
```

Kütüphaneler Nerede Aranır?

Paylaşımlı kütüphaneler standart dizinlerde aranır: `/lib`, `/usr/lib` ve `/etc/ld.so.conf` içindekiler.

```
# önbelleği güncelle (yeni .so ekleyince)
$ ldconfig

# geçici olarak ek dizin ekle
$ export LD_LIBRARY_PATH=/opt/lib:$LD_LIBRARY_PATH

# önbellekte bir kütüphaneyi ara
$ ldconfig -p | grep libm
```

libc.so ve libdl.so

`libc.so` Standart C kütüphanesi. Neredeyse her program buna bağlıdır: `printf`, `malloc`, `open`, `read` ve sistem çağrısı sarmalayıcıları burada.

`libdl.so` Çalışma anında kütüphane **elle yüklemek** için: `dlopen`, `dlsym`, `dlclose`. Eklenti (plugin) sistemlerinin temelidir.

```
void *k = dlopen("./eklenti.so", RTLD_NOW);
void (*f)() = dlsym(k, "calistir");   /* fonksiyonu bul */
f();                                   /* çağır          */
```

Statik ve Dinamik Karşılaştırma

	Statik (.a)	Dinamik (.so)
Dosya boyutu	Büyük	Küçük
Bağımlılık	Yok, taşınabilir	.so gerekir
Güncelleme	Yeniden derle	Kütüphaneyi değiştir yeter
Bellek	Her süreçte kopya	Paylaşımlı, tasarruflu
Başlangıç	Hızlı	Yükleme + çözümleme

Plugin (Eklenti) Kütüphaneler

Plugin, program çalışırken **istenirse yüklenen** paylaşımlı kütüphanedir. Program baştan bilmediği yeteneklerle sonradan genişletilebilir.

- Ana program `dlopen()` ile `.so` dosyasını yükler.
- Belirli bir isimdeki fonksiyonu `dlsym()` ile bulur.
- Yeni eklenti = yeni `.so` dosyası; ana programı yeniden derlemek gerekmez.

Örnek: tarayıcı eklentileri, ses/video codec'leri, editör uzantıları.

İçerik Analizi: Bağımlılıklar

Bir programın hangi paylaşımlı kütüphanelere ihtiyaç duyduğunu `ldd` gösterir:

```
$ ldd ./program
    linux-vdso.so.1
    libm.so.6 => /lib/libm.so.6
    libc.so.6 => /lib/libc.so.6
    /lib/ld-linux.so.2   (dinamik yükleyici)

# hangi .so tipini kullanıyor?
$ file ./program
```

İçerik Analizi: Semboller

Bir kütüphanenin içindeki fonksiyonlar (semboller) incelenebilir:

```
# dışa açılan (dynamic) sembolleri listele
$ nm -D /lib/libm.so.6 | head
0000... T sin
0000... T cos

# bölümleri ve başlıkları dök
$ objdump -T /lib/libm.so.6

# içindeki okunabilir metinleri gör
$ strings /lib/libm.so.6 | grep -i version
```

Dinamik Kütüphanelerde Sürüm

Sürümleme, farklı sürümlerin bir arada yaşamasını sağlar. Üç isim düzeyi vardır:

```
libfoo.so        → linker adı  (derlemede kullanılır, symlink)
libfoo.so.2      → soname      (çalışmada aranan, symlink)
libfoo.so.2.3.1  → gerçek dosya (asıl kod)

$ ls -l /usr/lib/libswipl.so*
lrwxrwxrwx 1 root root      13 Mar  1  2022 libswipl.so -> libswipl.so.8
lrwxrwxrwx 1 root root      17 Mar  1  2022 libswipl.so.8 -> libswipl.so.8.4.2
-rw-r--r-- 1 root root 1557296 Mar  1  2022 libswipl.so.8.4.2
```

- Baştaki sayı (**2**) uyumu bozan değişiklikte artar (major).
- Program soname'e (`.so.2`) bağlanır; ara sürümler sorunsuz güncellenir.

syslogd

syslogd, sistemdeki programlardan gelen kayıt mesajlarını toplayıp `/var/log` altındaki dosyalara yazan bir servistir.

Her mesajın bir **kaynağı** (facility) ve **önem düzeyi** (level) vardır:

`facility`	auth, cron, daemon, kern, user...
`level`	emerg, alert, crit, err, warning, notice, info, debug

Yapılandırma: `/etc/rsyslog.conf`. Modern sistemlerde eşdeğeri `journald`.

klogd

klogd, **çekirdeğin** ürettiği mesajları (dmesg'in kaynağı) okuyup syslogd'ye ileten servistir. Böylece çekirdek olayları da kalıcı kayıt dosyalarına düşer.

```
# çekirdek mesajları buradan gelir
$ dmesg
# klogd bunları /var/log/kern.log'a yazar
$ tail /var/log/kern.log
```

Modern sistemlerde bu işi de `systemd-journald` üstlenir; klogd ayrı bir süreç olmayabilir.

C'den syslog Kullanımı

```
#include <syslog.h>

int main(void) {
    /* günlüğü aç: etiket, seçenek, kaynak */
    openlog("denemeuyg", LOG_PID, LOG_USER);

    syslog(LOG_INFO,    "program basladi");
    syslog(LOG_WARNING, "disk %d%% dolu", 92);
    syslog(LOG_ERR,     "baglanti kurulamadi");

    closelog();
    return 0;
}

// bak: mylog.c
// $ tail /var/log/syslog
// $ journalctl -t denemeuyg
```

bash'ten syslog: logger

Kabuktan kayıt yazmak için `logger` komutu kullanılır:

```
#!/bin/bash

# basit mesaj
logger "yedekleme betigi basladi"

# etiket ve önem düzeyi ile
logger -t yedek -p user.info "kopyalama tamam"
logger -t yedek -p user.err  "hedef disk bulunamadi"

# çıktıyı kontrol et
tail /var/log/syslog
journalctl -t yedek
```

fd Nedir?

VFS (Virtual File System), çekirdeğin farklı dosya sistemlerini **tek bir ortak arayüzle** sunmasıdır. Program dosyanın ext4'te mi NFS'te mi olduğunu bilmeden aynı çağrıları kullanır.

fd (file descriptor), açılan her dosya/soket/borunun program içindeki **küçük tam sayı numarasıdır**. Tüm G/Ç bu numara üzerinden yapılır.

```
int fd = open("dosya.txt", O_RDONLY);  /* fd = 3 döner */
read(fd, tampon, 100);                  /* numara ile oku */
close(fd);                              /* serbest bırak  */
```

Ayrılmış (Rezerve) fd'ler

Her program açılışta üç fd ile başlar; bunlar standart akışlardır:

0	`stdin`	standart giriş
1	`stdout`	standart çıkış
2	`stderr`	standart hata

Yeni açılan dosyalar 3'ten başlar. Bir sürecin açık fd'leri incelenebilir:

```
$ ls -l /proc/1234/fd
```

Paket Yöneticisi Olmadan Derleme

Kaynak kodu genellikle sıkıştırılmış bir arşiv (`.tar.gz`) olarak gelir. Klasik derleme üç adımdır:

```
# 1) arşivi aç
$ tar xzf uygulama-1.2.tar.gz
$ cd uygulama-1.2

# 2) sistemi tanı ve Makefile üret
$ ./configure

# 3) derle
$ make
```

`configure` gerekli kütüphaneleri kontrol eder; eksikse hata verir.

Nasıl Install Edilir?

```
# dosyaları sisteme kopyala (genelde root gerekir)
$ sudo make install

# nereye kurulacağını seç
$ ./configure --prefix=/usr/local

# sistemi kirletmeden bir dizine topla (paketleme)
$ make install DESTDIR=/tmp/paket
```

- Varsayılan hedef çoğunlukla `/usr/local`'dır.
- Kaldırma için (varsa) `make uninstall`.
- Yeni kütüphane kurulduysa `ldconfig` çalıştırın.

Örnek: libzip

```
# 1) kaynağı aç ve derle (libzip cmake kullanır)
$ tar xzf libzip-1.10.tar.gz && cd libzip-1.10
$ mkdir build && cd build
$ cmake .. && make
$ sudo make install && sudo ldconfig
```

```
# 2) libzip'e bağlanan bir programı derle
$ gcc program.c -lzip -o program
#          kütüphaneyi bağla ─────┘
# başlık dosyası: #include <zip.h>
```

Bağlama hatası olursa `pkg-config --cflags --libs libzip` doğru bayrakları verir.

Sık Kullanılan Aygıt Adları

`/dev/sda`	1. SATA/SCSI/USB disk; bölümleri sda1, sda2…
`/dev/sdb`	2. disk (örn. takılan USB bellek).
`/dev/mmcblk0`	SD/eMMC kart; bölümleri mmcblk0p1, p2…
`/dev/nvme0n1`	NVMe SSD (1. sürücü, 1. namespace).
`/dev/ttyUSB0`	USB-seri dönüştürücü (seri port).
`/dev/ttyS0`	Yerleşik seri port (COM1).
`/dev/tty1`	Sanal konsol; /dev/pts/0 sözde terminal.

Fonksiyon İşaretçisi Nedir?

Bir **fonksiyon işaretçisi**, bir fonksiyonun adresini tutar. Böylece hangi fonksiyonun çağrılacağı **çalışma anında** seçilebilir; tablolar, geri çağırmalar (callback) ve eklentiler bununla yapılır.

```
/* iki int alıp int döndüren fonksiyonlara işaretçi */
int (*islem)(int, int);

int topla(int a, int b) { return a + b; }

islem = topla;          /* adresi ata */
int s = islem(3, 4);    /* 7 : işaretçi üzerinden çağır */
```

Örnek: 4 İşlemli Hesap

```
#include <stdio.h>

int topla(int a,int b){ return a+b; }
int cikar(int a,int b){ return a-b; }
int carp (int a,int b){ return a*b; }
int bol  (int a,int b){ return b ? a/b : 0; }

int main(void) {
    /* fonksiyon işaretçisi dizisi: 4 işlem */
    int (*islem[4])(int,int) = { topla, cikar, carp, bol };
    const char *ad[4] = { "+", "-", "*", "/" };
    int a = 12, b = 4;

    for (int i = 0; i < 4; i++)
        printf("%d %s %d = %d\n", a, ad[i], b, islem[i](a, b));
    return 0;
}
// bak ptr.c
// sonuçlar:  12 + 4 = 16   12 - 4 = 8   12 * 4 = 48   12 / 4 = 3
```

Statik Derleme

Kütüphaneler ikili dosyanın **içine kopyalanır**; sonuç kendi kendine yeten, taşınabilir bir dosyadır.

```
$ gcc -static fptr.c -o fptr
$ ldd fptr
    not a dynamic executable      # dış .so yok

$ ls -lh fptr
-rwxr-xr-x  880K  program          # büyük ama bağımsız
```

Artı: her yerde çalışır. **Eksi:** büyük dosya, güncelleme için yeniden derleme gerekir.

Dinamik Derleme

Varsayılan yöntemdir. Kütüphaneler kopyalanmaz; çalışma anında sistemdeki `.so` dosyalarından yüklenir.

```
$ gcc program.c -o program        # dinamik (öntanımlı)
$ ldd program
    libc.so.6 => /lib/libc.so.6

$ ls -lh program
-rwxr-xr-x  16K  program           # küçük, .so'lara bağlı
```

Artı: küçük dosya, ortak bellek, kolay güncelleme. **Eksi:** doğru `.so` sistemde bulunmalı.

Kod Boyunun Kısaltılması

```
# hata ayıklama sembollerini at
$ strip program

# boyut için optimize et
$ gcc -Os program.c -o program
```

- `strip` sembol tablosunu kaldırır, dosyayı küçültür.
- `-Os` hız yerine boyutu önceler.
- Gömülü sistemlerde `size program` ile bölüm boyutları izlenir.

Bağlı Kütüphaneleri Bulma

```
# çalışan/çözümlenmiş bağımlılıklar
$ ldd program

# ikilinin istediği .so listesi (NEEDED)
$ readelf -d program | grep NEEDED

# derleme için gerekli bayrakları öğren
$ pkg-config --cflags --libs libzip
```

`ldd` "hangi dosya nerede bulundu"yu, `readelf -d` ise ikilinin ham olarak istediği kütüphaneleri gösterir. Eksik bağımlılık burada ortaya çıkar.

I/O Multiplexing Kavramı

Bir program aynı anda birçok bağlantı/dosyayı beklemek zorunda kalabilir. Her biri için ayrı iş parçacığı açmak pahalıdır.

I/O multiplexing, tek bir noktadan **çok sayıda fd'yi izlemeyi** sağlar: "bunlardan hangisi okumaya/yazmaya hazır?" diye sorulur ve yalnızca hazır olanlar işlenir.

- Meşgul beklemeden (busy-wait) kaçınılır.
- Çekirdek çağrıları: `select`, `poll`, `epoll`.

poll Çağrısı

`poll`, izlenecek fd'lerin bir dizisini alır. Her fd için hangi olayın beklendiği (`events`) yazılır; dönüşte hangi olayların gerçekleştiği (`revents`) okunur.

```
struct pollfd {
    int   fd;         /* izlenecek tanıtıcı        */
    short events;     /* beklenen: POLLIN, POLLOUT */
    short revents;    /* gerçekleşen (çekirdek doldurur) */
};

int n = poll(fds, sayi, timeout_ms);
/* n>0: hazır fd sayısı, 0: zaman aşımı, -1: hata */
```

Çok Basit poll Örneği

```
#include <stdio.h>
#include <poll.h>
#include <unistd.h>

int main(void) {
    struct pollfd f = { .fd = 0, .events = POLLIN };  /* stdin */
    char tampon[128];

    while (1) {
        int n = poll(&f, 1, 5000);           /* 5 sn bekle */
        if (n == 0) { printf("bes saniye sessizlik\n"); continue; }
        if (f.revents & POLLIN) {             /* veri hazir */
            int k = read(0, tampon, sizeof tampon);
            if (k <= 0) break;                /* EOF: cik    */
            write(1, tampon, k);             /* geri yaz    */
        }
    }
    return 0;
}

// bak poll.c
```

popen Ne İşe Yarar?

popen, C programından bir **kabuk komutu çalıştırıp** onun çıktısını bir boru üzerinden dosya gibi okumayı (veya girişine yazmayı) sağlar. `pclose` ile kapatılır.

```
FILE *p = popen("ls -l", "r");   /* çıktısını oku */
/* ... fgets(...) ile satır satır oku ... */
pclose(p);

FILE *q = popen("wc -l", "w");   /* girişine yaz  */
```

Kısa yol olarak pratiktir; ama komut kabuktan geçtiği için kullanıcı girdisiyle oluşturulan komutlarda **güvenlik** riskine dikkat edilmelidir.

Basit Bir C Kodu

```
#include <stdio.h>

int main(void) {
    /* 'date' komutunu çalıştır, çıktısını oku */
    FILE *p = popen("date +%H:%M", "r");
    if (!p) return 1;

    char satir[64];
    if (fgets(satir, sizeof satir, p))
        printf("Su anki saat: %s", satir);

    pclose(p);
    return 0;
}

// Çıktı:  Su anki saat: 14:07
// ayrıca bak mytop.c
```

Web Scraping Nasıl Yapılır?

Terminalde web scraping = sayfayı **indir** + istenen kısmı **süz**. İndirme için `curl`/`wget`, süzme için grep/sed/awk kullanılır.

```
# sayfayı indir ve başlığı çek
$ curl -s https://ornek.test \
    | grep -oE '<title>[^<]+' \
    | sed 's/<title>//'

# C içinden: popen ile aynı boru
FILE *p = popen("curl -s https://ornek.test", "r");

// bak doviz.c
```

Not: Sadece izin verilen sitelerde yapılmalıdır.

Process (Süreç) Nedir?

Süreç, çalışan bir programın canlı örneğidir. Diskteki program pasif dosyadır; çalıştırıldığında bir süreç olur. Her sürecin kendi **bellek alanı**, açık dosyaları ve durumu vardır.

`PID` sürecin benzersiz numarası

`PPID` ata (parent) sürecin numarası

`durum` çalışıyor, uyuyor, durmuş, zombi

```
$ ps -ef        $ top        $ pstree
```

Süreç Nasıl Oluşur: fork + exec

`fork()` mevcut sürecin kopyasını (çocuk) üretir; `exec()` o kopyanın üzerine yeni program yükler.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main(void) {
    pid_t p = fork();              /* süreci ikiye ayır */
    if (p == 0) {                  /* çocuk             */
        execlp("ls", "ls", "-l", NULL);   /* ls'e dönüş  */
    } else {                       /* ebeveyn           */
        wait(NULL);                /* çocuğu bekle      */
        printf("cocuk bitti\n");
    }
    return 0;
}
```

Thread ile Process Farkı

	Process	Thread
Bellek	Ayrı, izole adres alanı	Aynı adres alanını paylaşır
Oluşturma	Ağır (fork)	Hafif
İletişim	IPC gerekir	Doğrudan paylaşılan bellek
Hata etkisi	Biri çökse diğeri yaşar	Biri çökerse tüm süreç düşer
Kütüphane	fork/exec	pthread

Thread'ler veriyi kolay paylaşır ama **eşzamanlılık** (kilit, yarış durumu) sorumluluğu programcıya kalır.

Joinable ve Detachable

Bir thread iki biçimde çalışabilir:

`joinable` Bittiğinde başka bir thread onu `pthread_join()` ile **beklemeli** ve sonucunu toplamalıdır. Aksi halde kaynak sızar.

`detached` Kimse beklemez; bitince kaynaklarını **kendi** serbest bırakır. `pthread_detach()` ile ayrılır.

```
pthread_create(&t, NULL, is, NULL);
pthread_join(t, NULL);        /* joinable: bekle ve topla */
pthread_detach(t);            /* detached: kendi temizler */
```

Zombi Süreçler

Zombi, işi bitmiş ama ata süreci henüz **çıkış kodunu toplamamış** süreçtir. Bellek kullanmaz, sadece süreç tablosunda bir giriş tutar. Çok birikirse tablo dolar.

```
# tespit: durumu Z (defunct) olanlar
$ ps aux | grep 'Z'
$ ps -eo pid,ppid,stat,cmd | awk '$3 ~ /Z/'
```

Engelleme: ata süreç çocuğunu toplamalıdır:

```
wait(&durum);                /* çocuğu topla       */
signal(SIGCHLD, SIG_IGN);    /* ya da otomatik topla */
```

IPC Nedir?

IPC (Inter-Process Communication), ayrı bellek alanlarına sahip süreçlerin birbiriyle **veri alışverişi** yapması ve **eşgüdüm** sağlamasıdır. Süreçler izole olduğundan, iletişim için çekirdeğin sunduğu mekanizmalar gerekir.

Neden gerekir: bir süreç veri üretir, diğeri tüketir; bir servis isteği alır, işçi süreçlere dağıtır; birden çok süreç ortak bir kaynağı paylaşır.

IPC Yöntemleri Haritası

`Pipe / FIFO`	Tek yönlü bayt akışı (boru).
`Message Queue`	Sıralı, mesaj tabanlı iletişim.
`Shared Memory`	Ortak bellek: en hızlı veri paylaşımı.
`Semaphore / Mutex`	Eşgüdüm ve karşılıklı dışlama.
`Socket`	Yerel veya ağ üzerinden iki yönlü.
`Signal`	Basit olay bildirimi.

POSIX'in Desteklediği IPC

İki ana aile vardır: eski **System V** ve daha yeni, daha temiz **POSIX**. POSIX arayüzleri isimle (bir yol gibi) açılır.

`shm_open`	POSIX paylaşımlı bellek
`mq_open`	POSIX mesaj kuyruğu
`sem_open`	POSIX isimli semafor
`pthread_mutex`	iş parçacığı karşılıklı dışlama

Bağlama için genellikle `-lrt` ve `-lpthread` gerekir.

Yaşam Döngüsü ve Kalıcılık

POSIX IPC nesneleri isimlidir ve `/dev/shm` altında görünür. Açan süreç kapansa bile **çekirdek yeniden başlayana kadar** yaşarlar; bu yüzden açıkça silinmeleri gerekir.

```
/* aç → kullan → kapat → sil */
int fd = shm_open("/veri", O_CREAT|O_RDWR, 0600);
/* ... */
close(fd);
shm_unlink("/veri");     /* ismi kaldır (kalıcılığı bitir) */

$ ls /dev/shm            # açık POSIX nesnelerini gör
```

mmap — Belleğe Eşleme

`mmap`, bir dosyayı (veya boş alanı) doğrudan sürecin **bellek adres alanına** bağlar. Artık read/write yerine, diziye erişir gibi **işaretçiyle** okunup yazılır.

- Büyük dosyalarda kopyalama yükünü azaltır.
- Aynı dosya birden çok süreçte eşlenirse **paylaşımlı** hale gelir.
- Çekirdek sayfa sayfa, ihtiyaç oldukça yükler (demand paging).

Paylaşımlı Bellek

İki veya daha çok süreç **aynı fiziksel belleği** kendi adres alanlarına eşler. Bir süreç yazar, diğeri anında görür. Veri kopyalanmadığı için **en hızlı IPC** yöntemidir.

Adımlar:

- `shm_open` ile isimli bellek nesnesi aç.
- `ftruncate` ile boyut ver.
- `mmap` ile adres alanına eşle.

Dikkat: hızlıdır ama **senkronizasyon yoktur**; semafor/mutex ile korunmalı.

POSIX shm Örneği (yazan)

```
#include <fcntl.h>
#include <sys/mman.h>
#include <string.h>
#include <unistd.h>

int main(void) {
    int fd = shm_open("/mesaj", O_CREAT|O_RDWR, 0600);
    ftruncate(fd, 128);                       /* boyut ayarla */
    char *p = mmap(NULL, 128, PROT_WRITE,
                   MAP_SHARED, fd, 0);        /* eşle          */

    strcpy(p, "merhaba paylasimli bellek");   /* ortak alana yaz */

    /* okuyan süreç: shm_open("/mesaj") + mmap ile aynı metni görür */
    munmap(p, 128);  close(fd);
    return 0;
}
/*  Derleme:  gcc shm.c -lrt -o shm  */
```

Mesaj Kuyruğu

Süreçler birbirine **ayrık mesajlar** (bayt akışı değil, paketler) gönderir. Çekirdek mesajları bir kuyrukta tutar; alıcı hazır olduğunda okur. Gönderen ve alan aynı anda çalışmak zorunda değildir.

- Mesaj sınırları korunur (her okuma bir mesaj).
- Mesajlara **öncelik** verilebilir.
- Kuyruk dolu/boşsa süreç bekletilebilir (bloklama).

POSIX mq Örneği

```
#include <mqueue.h>
#include <string.h>
#include <stdio.h>

int main(void) {
    /* gönderen taraf */
    mqd_t q = mq_open("/kuyruk", O_CREAT|O_WRONLY, 0600, NULL);
    mq_send(q, "isle:42", 8, 0);      /* mesaj, boyut, öncelik */
    mq_close(q);

    /* alan taraf */
    mqd_t r = mq_open("/kuyruk", O_RDONLY);
    char tampon[256];
    mq_receive(r, tampon, sizeof tampon, NULL);
    printf("gelen: %s\n", tampon);
    mq_close(r);  mq_unlink("/kuyruk");
    return 0;
}

// bak que_reader.c, que_writer.c
```

Neden Senkronizasyon?

İki süreç/thread aynı veriyi aynı anda değiştirirse sonuç bozulur (**yarış durumu**). Bunu önlemek için erişim sıraya sokulur.

`mutex` Kilit: aynı anda **tek** kişi kritik bölgeye girer (kilitle → kullan → çöz).

`semafor` Sayaç: en fazla **N** erişime izin verir. mutex, N=1 olan özel bir semafor gibidir.

İsimsiz Borular (pipe)

Tek yönlü bayt akışıdır; genellikle **akraba süreçler** (fork ile) arasında kullanılır. Kabuktaki | operatörü de budur.

```
int fd[2];
pipe(fd);              /* fd[0]=okuma, fd[1]=yazma */

if (fork() == 0) {                /* çocuk: yazar */
    close(fd[0]);
    write(fd[1], "selam", 5);
} else {                          /* ebeveyn: okur */
    close(fd[1]);
    char b[16];
    int n = read(fd[0], b, 16);
    write(1, b, n);
}
```

İsimli Borular (FIFO)

FIFO, dosya sisteminde bir **ada** sahip borudur. Böylece **akraba olmayan** süreçler de aynı boruyu açıp haberleşebilir.

```
# kabuktan FIFO oluştur
$ mkfifo /tmp/boru

# terminal 1: oku (yazan gelene kadar bekler)
$ cat /tmp/boru

# terminal 2: yaz
$ echo "merhaba" > /tmp/boru

# C'den:  mkfifo("/tmp/boru", 0600); open(...);
```

Diğer IPC Yöntemleri

`Unix soketi`	Yerel, iki yönlü, güvenilir; fd bile aktarabilir. Aynı makinedeki servisler için standarttır.
`eventfd`	Süreçler/thread'ler arası basit sayaç tabanlı olay bildirimi (tek fd).
`signalfd`	Sinyalleri handler yerine bir fd üzerinden okumaya çevirir.
`signal`	En basit bildirim: veri taşımaz, sadece "olay oldu" der.

Hangi Yöntemi Seçmeli?

En hızlı veri	Shared memory (+ semafor ile korunmuş).
Basit akış	Pipe / FIFO.
Yapılandırılmış mesaj	Message queue.
Ağ / makineler arası	Socket (TCP/UDP).
Sadece bildirim	Signal / eventfd.

Kural: ihtiyaç kadar karmaşık olanı seç. Basit bir bildirim için paylaşımlı bellek kurmak gereksiz yüktür.

Netlink Nedir?

Netlink, kullanıcı programları ile **çekirdek** arasında iki yönlü mesajlaşma sağlayan özel bir soket ailesidir (`AF_NETLINK`). ioctl'e göre daha esnek ve olay tabanlıdır.

Yaygın kanalları:

`NETLINK_ROUTE`	ağ arayüzü, rota, adres olayları
`NETLINK_KOBJECT_UEVENT`	aygıt takma/çıkarma olayları

Netlink Soketi Açmak

```
#include <linux/netlink.h>
#include <sys/socket.h>
#include <string.h>

int main(void) {
    /* uevent kanalına bağlan */
    int s = socket(AF_NETLINK, SOCK_DGRAM, NETLINK_KOBJECT_UEVENT);

    struct sockaddr_nl sa = {0};
    sa.nl_family = AF_NETLINK;
    sa.nl_groups = 1;                 /* çekirdek olay grubu */
    bind(s, (void*)&sa, sizeof sa);

    char tampon[4096];
    recv(s, tampon, sizeof tampon, 0);  /* olay mesajını al */
    return 0;
}

// bak netlink.c
```

Olayları Nasıl İzleriz?

Çekirdek, donanım ve durum değişikliklerini **uevent** mesajları olarak yayınlar. Kabuktan izlemek en kolay yoldur:

```
# tüm aygıt olaylarını canlı izle
$ udevadm monitor

# çekirdek + udev olaylarını ayrı göster
$ udevadm monitor --kernel --udev

# bir aygıtın tüm özniteliklerini dök
$ udevadm info -a -n /dev/sdb
```

Bir uevent Mesajı

Her olay ad=değer satırlarından oluşur. USB bellek takıldığında gelen tipik mesaj:

```
ACTION=add                  # add / remove / change
DEVNAME=/dev/sdb            # aygıt düğümü
SUBSYSTEM=block             # alt sistem
DEVTYPE=disk
ID_VENDOR=Kingston
SEQNUM=2481
```

`ACTION` alanı ne olduğunu, `SUBSYSTEM` hangi tür aygıt olduğunu söyler. Kurallar bu alanlara bakarak eşleşir.

udevd Nasıl Çalışır?

udevd (systemd-udevd), çekirdekten gelen uevent olaylarını dinleyen bir servistir. Bir aygıt takıldığında:

1. Çekirdek bir **uevent** yayınlar (netlink üzerinden).
2. udevd olayı alır ve `/etc/udev/rules.d/` altındaki **kurallarla** karşılaştırır.
3. `/dev` altındaki **aygıt düğümünü** oluşturur, izin/isim verir.
4. Kuralda tanımlıysa bir **program çalıştırır**.

udev Kuralları

Kurallar **eşleşme** (==) ve **atama** (=) ifadelerinden oluşur:

```
# /etc/udev/rules.d/99-usb.rules

# bu üretici/model USB takılınca sabit isim ver
SUBSYSTEM=="block", ATTRS{idVendor}=="0951", \
  SYMLINK+="benim_usb"

# kuralları yeniden yükle ve tetikle
$ udevadm control --reload
$ udevadm trigger
```

Böylece aygıt her takıldığında `/dev/benim_usb` ile erişilir.

Aygıt Takılınca Program Çalıştırmak

udev kuralına RUN eklenir. Belirtilen aygıt takıldığında (ACTION=="add") çekirdek adına o program çalıştırılır.

```
# /etc/udev/rules.d/90-usb-tak.rules

ACTION=="add", SUBSYSTEM=="block", \
  ATTRS{idVendor}=="0951", \
  RUN+="/usr/local/bin/usb_takildi.sh"
```

Not: RUN ile çalışan program **kısa sürmeli**; uzun işler için servis tetiklenmelidir (udev programı bloklamaz).

Örnek Betik ve Test

```
#!/bin/bash
# /usr/local/bin/usb_takildi.sh
logger -t usbtak "USB takildi: $DEVNAME ($(date))"
```

```
# izin ver, kuralı yükle
$ chmod +x /usr/local/bin/usb_takildi.sh
$ sudo udevadm control --reload

# USB tak, kaydı kontrol et
$ sudo journalctl -t usbtak

# 0951:1666 Kingston Technology
$ lsub
```

udev, betiğe olay bilgisini `$DEVNAME`, `$ACTION` gibi ortam değişkenleriyle geçirir.

POSIX Nedir, Ne İşe Yarar?

POSIX (Portable Operating System Interface), Unix benzeri sistemlerin uyması gereken ortak bir **standarttır**. Sistem çağrılarını, kabuk davranışını ve temel komutları tanımlar.

Amacı **taşınabilirliktir**: POSIX'e uygun yazılan bir program, Linux'ta da diğer POSIX sistemlerde de (çoğunlukla) yeniden derlenerek çalışır.

Örnek POSIX arayüzleri: `open`, `read`, `write`, `fork`, `pthread_create`, `sem_open`.

POSIX Özellikleri Nasıl Elde Edilir?

Bazı POSIX arayüzleri, bir **özellik makrosunun** (feature test macro) tanımlanmasıyla açılır:

```
/* başlıklardan ÖNCE tanımla */
#define _POSIX_C_SOURCE 200809L
#include <unistd.h>

$ gcc -D_POSIX_C_SOURCE=200809L program.c
```

```
# sistemin POSIX sürüm/limit değerleri
$ getconf _POSIX_VERSION
$ getconf -a | grep POSIX
```

Sistem Çağrısı Nedir?

Sistem çağrısı (system call), bir programın çekirdekten **hizmet istediği** denetimli giriş noktasıdır. Dosya açma, bellek isteme, süreç oluşturma gibi ayrıcalıklı işler ancak böyle yapılır.

```
Kullanıcı alanı  ──syscall──▶  Çekirdek alanı
   printf()  →  write(1,...)  →  çekirdek ekrana yazar

# bir programın syscall'larını gör
$ strace -c ./program
```

Kütüphane fonksiyonları (`printf`) çoğu zaman altta bir sistem çağrısını (`write`) sarmalar.

time ile Analiz

`time`, bir komutun ne kadar sürdüğünü üç açıdan gösterir:

```
$ time ./program
real    0m2.104s     # geçen gerçek (duvar, kol saati) süre
user    0m1.980s     # kullanıcı alanında CPU süresi
sys     0m0.090s     # çekirdek (syscall) CPU süresi
```

- **user** yüksekse: hesap ağırlıklı (CPU-bound).
- **sys** yüksekse: çok fazla sistem çağrısı / G/Ç.
- **real** ≫ user+sys ise: beklemede (disk, ağ) zaman geçiyor.

TCP/IP Soket Kavramı

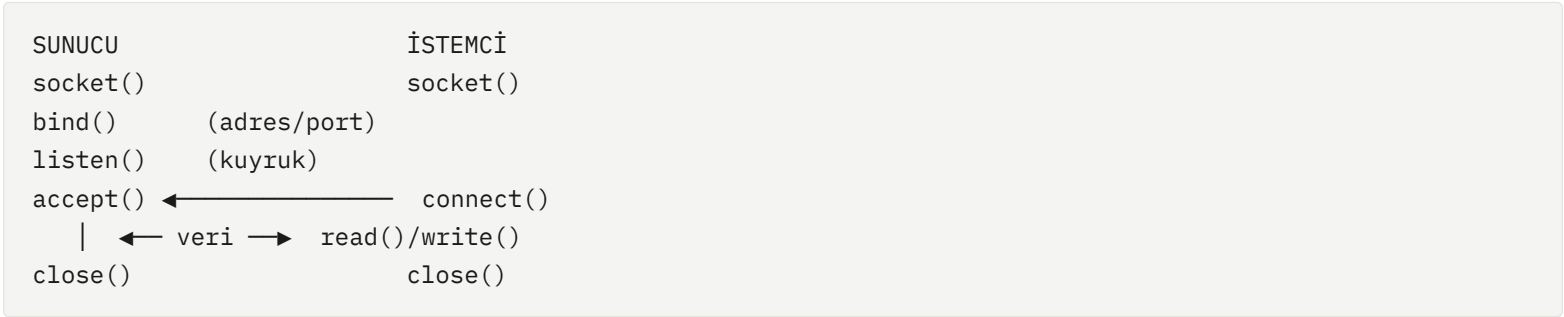
Soket, iki program arasında ağ üzerinden iletişim kuran uç noktadır. **TCP** ise bağlantı kuran, **güvenilir** ve **sıralı** bir bayt akışı sağlar: gönderilen veriler kaybolmadan, sırasıyla ulaşır.

`IP`	Hangi makine (adres).
`Port`	Makinedeki hangi servis (0–65535).
`SOCK_STREAM`	TCP soket tipi.

Sunucu bir portu **dinler**, istemci o adrese **bağlanır**; sonra iki taraf `read`/`write` ile konuşur.

Kavramsal Yaklaşım

TCP'de sunucu ve istemci farklı çağrı sırası izler:



`accept()` her istemci için ayrı bir bağlantı soketi döndürür; dinleme soketi açık kalır.

Çok Basit C Client

```
#include <stdio.h>
#include <string.h>
#include <arpa/inet.h>
#include <unistd.h>

int main(void) {
    int s = socket(AF_INET, SOCK_STREAM, 0);
    struct sockaddr_in a = {0};
    a.sin_family = AF_INET;
    a.sin_port   = htons(1234);
    inet_pton(AF_INET, "127.0.0.1", &a.sin_addr);

    connect(s, (struct sockaddr*)&a, sizeof a);
    write(s, "Merhaba\n", 8);            /* sunucuya gönder */

    char buf[128];
    int n = read(s, buf, sizeof buf);    /* yanıtı oku    */
    write(1, buf, n);
    close(s);
    return 0;
}
// bak socket_client.c
```

Çok Basit C Server

```
#include <string.h>
#include <arpa/inet.h>
#include <unistd.h>

int main(void) {
    int s = socket(AF_INET, SOCK_STREAM, 0);
    struct sockaddr_in a = {0};
    a.sin_family      = AF_INET;
    a.sin_port        = htons(1234);
    a.sin_addr.s_addr = INADDR_ANY;

    bind(s, (struct sockaddr*)&a, sizeof a);
    listen(s, 1);                       /* bağlantı bekle */

    int c = accept(s, NULL, NULL);      /* istemciyi al   */
    char buf[128];
    int n = read(c, buf, sizeof buf);
    write(c, buf, n);                   /* geri yolla (echo) */
    close(c); close(s);
    return 0;
}
// bak socket_server.c
```

Derle/çalıştır: `gcc server.c -o server && ./server` — sonra client'ı çalıştır.

UDP/IP Soket Kavramı

UDP bağlantısızdır: el sıkışma yoktur, her mesaj (**datagram**) tek başına gönderilir. Hızlıdır ama **güvenilir değildir** — paket kaybolabilir, sıra değişebilir, tekrarlanabilir.

`SOCK_DGRAM`	UDP soket tipi.
`sendto()`	Her mesajla hedef adresi belirtilir.
`recvfrom()`	Mesajı ve gönderenin adresini alır.

Kullanım: DNS, DHCP, video/ses akışı, oyunlar — hız gecikmeden önemli olduğunda.

Kavramsal Yaklaşım

UDP'de bağlantı kurma adımı yoktur; sunucu sadece bir portu **bind** eder ve mesajları bekler:

```
SUNUCU                    İSTEMCİ
socket()                  socket()
bind() (adres/port)
recvfrom() ◄── datagram ──  sendto()
sendto()  ── datagram ──►  recvfrom()
close()                   close()
```

`listen()`, `accept()`, `connect()` yoktur; adres her `sendto` ile taşınır.

Çok Basit C Client

Çok Basit C Server

mount Nedir?

mount, bir dosya sistemini (disk, bölüm, imaj, ağ paylaşımı) dizin ağacındaki bir **bağlama noktasına** ekler. O andan itibaren aygıtın içeriği o dizin altından görünür.

```
# USB'nin 1. bölümünü /mnt/usb'ye bağla
$ mount /dev/sdb1 /mnt/usb

# bağlı olanları listele
$ mount
$ findmnt

# çıkar
$ umount /mnt/usb
```

Kavramsal Yaklaşım

Linux'ta **tek bir ağaç** vardır (kök /). Her yeni depolama bu ağaca bir dala takılır. Bağlama noktası boş bir dizin olmalıdır; bağlıyken eski içeriği görünmez olur.

`-t`	dosya sistemi tipi (ext4, vfat...)
`-o ro`	salt-okunur bağla
`-o loop`	imaj dosyasını bağla
`--bind`	bir dizini başka yere yansıt

Açılışta otomatik bağlama `/etc/fstab` ile tanımlanır.

Sinyaller Nedir?

Sinyal, çekirdeğin bir sürece gönderdiği kısa, **asenkron** bir bildirimdir — "yazılımsal kesme". Süreç ne yapıyorsa durur, sinyali işler, sonra kaldığı yerden devam eder.

Bir sinyal geldiğinde süreç üç davranıştan birini seçebilir:

- Varsayılan eylemi uygula (çoğu için: sonlan).
- Sinyali **yok say** (SIG_IGN).
- Kendi **işleyicisini** (handler) çalıştır.

`SIGKILL (9)` ve `SIGSTOP (19)` yakalanamaz, yok sayılamaz.

Kesmeler ile Farkları

İkisi de "işi böl, olayı işle" mantığındadır, ama katman farklıdır:

KESME (interrupt)

- Donanım → **çekirdek** katmanı.
- Kaynağı: aygıt, zamanlayıcı, CPU.
- Handler'ı çekirdekte çalışır.
- Kullanıcıya görünmez.

SİNYAL (signal)

- Çekirdek → **süreç** katmanı.
- Kaynağı: çekirdek, kill, klavye.
- Handler'ı süreçte çalışır.
- "Kullanıcı alanı kesmesi".

Anlamı ve Kaynağı

SIGINT	2	Klavye Ctrl+C — kullanıcı durdurdu.
SIGTERM	15	Nazikçe sonlan isteği (kill varsayılanı).
SIGKILL	9	Çekirdek zorla öldürür — yakalanamaz.
SIGSEGV	11	Geçersiz bellek erişimi — çekirdek üretir.
SIGCHLD	17	Çocuk süreç sonlandı/durdu.
SIGHUP	1	Terminal koptu / yapılandırmayı yeniden yükle.
SIGALRM	14	alarm()/timer zamanlayıcısı doldu.

Kaynaklar özetle: **klavye** (Ctrl+C/Z), **çekirdek** (hata/zamanlayıcı), **başka süreç** (kill), **sürecin kendisi** (raise).

Programlar Nasıl Tedbir Almalı?

- **SIGTERM/SIGINT** yakala: kaynakları serbest bırak, dosyaları kapat, temiz çık.
- Handler içinde **sadece** async-signal-safe iş yap; bayrak kaldır, gerçek işi ana döngüde yürüt.
- **SIGCHLD** yakalayıp `wait()` çağır — zombi süreç bırakma.
- Kesilen sistem çağrılarını (`EINTR`) yeniden dene.

```
volatile sig_atomic_t dur = 0;
void isle(int s) { dur = 1; }   /* sadece bayrak */
/* ana döngü: while(!dur){...}  → temiz çıkış */
```

Signal Handler — Eski Yaklaşım

Eski, taşınabilir yöntem `signal()` çağrısıdır:

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

volatile sig_atomic_t dur = 0;

void isle(int sig) {           /* handler */
    dur = 1;
}

int main(void) {
    signal(SIGINT, isle);      /* Ctrl+C'yi bagla */
    while (!dur)
        pause();               /* sinyal gelene dek bekle */
    printf("\nTemiz cikis.\n");
    return 0;
}
```

Not: modern kod `sigaction()` tercih eder; `signal()` davranışı sistemler arası değişebilir.

Çekirdek Modülleri Nedir?

Çekirdek modülü, çekirdeğe **çalışma zamanında** eklenip çıkarılabilen kod parçasıdır (çoğu aygıt sürücüsü). Sistemi yeniden derlemeden yeni donanım/özellik desteği eklenir.

`.ko`	Kernel Object — modül dosyası.
`/lib/modules/$(uname -r)/`	Modüllerin bulunduğu dizin.

```
$ lsmod             # yüklü modülleri listele
$ uname -r          # çalışan çekirdek sürümü
```

Mevcut Modüllerin Analizi

```
# yüklü modüller: ad | boyut | kullanan sayısı | kimler
$ lsmod
Module      Size    Used by
usbcore     331776  6   usbhid,ehci_hcd,...
ext4        802816  1

# bir modülün bağımlılık zinciri
$ lsmod | grep usbhid
```

- **Used by** sütunu 0 değilse modül kullanımdadır, kaldırılamaz.
- Bağımlılıklar `/lib/modules/.../modules.dep` dosyasında tutulur.
- `/proc/modules`, `lsmod`'un ham kaynağıdır.

modinfo, modprobe ve Diğerleri

`modinfo ad`	Modülün açıklaması, yazarı, parametreleri, bağımlılıkları.
`modprobe ad`	Modülü **bağımlılıklarıyla** birlikte yükler (önerilen yol).
`modprobe -r ad`	Modülü bağımlılıklarıyla kaldırır.
`insmod dosya.ko`	Tek bir .ko dosyasını yükler (bağımlılık çözmez).
`rmmod ad`	Tek modülü kaldırır.
`depmod`	Bağımlılık haritasını (modules.dep) yeniden üretir.

```
$ sudo modprobe loop        # loop modülünü yükle
$ modinfo loop | head        # bilgilerini gör
```

crontab Nedir?

cron, komutları belirli zamanlarda **otomatik** çalıştıran zamanlayıcıdır. Her kullanıcının kendi **crontab** tablosu vardır; her satır bir görevdir: **beş zaman alanı** + çalıştırılacak komut.

```
┌──── dakika        (0-59)
│ ┌── saat          (0-23)
│ │ ┌ ayın günü     (1-31)
│ │ │ ┌ ay          (1-12)
│ │ │ │ ┌ haftanın günü (0-7, 0/7=Pazar)
│ │ │ │ │
* * * * *  calistirilacak_komut
```

`crontab -e` Tabloyu düzenle.

`crontab -l` Görevleri listele.

`crontab -r` Tüm görevleri sil.

crontab Örnekleri

```
# her gün saat 02:30'da yedek al
30 2 * * *   /opt/betik/yedekle.sh

# her 5 dakikada bir kontrol
*/5 * * * *  /usr/local/bin/kontrol.sh

# hafta içi (Pzt-Cum) 09:00'da rapor
0 9 * * 1-5  /opt/betik/rapor.sh

# her Pazar gece yarısı log temizle
0 0 * * 0    /opt/betik/log-temizle.sh
```

İpucu: `*` = "her değer"; `*/n` = "her n birimde"; `a-b` = aralık; `a,b` = liste. Çıktıyı yakalamak için komut sonuna `>> log 2>&1` ekleyin.

Konu İndeksi

Konu İndeksi

Telif Hakları

Bu belge **UCanLinux.com** kaynak gösterilerek, özgürce **güncellenebilir**, **dağıtılabilir** ve **kullanılabilir**.