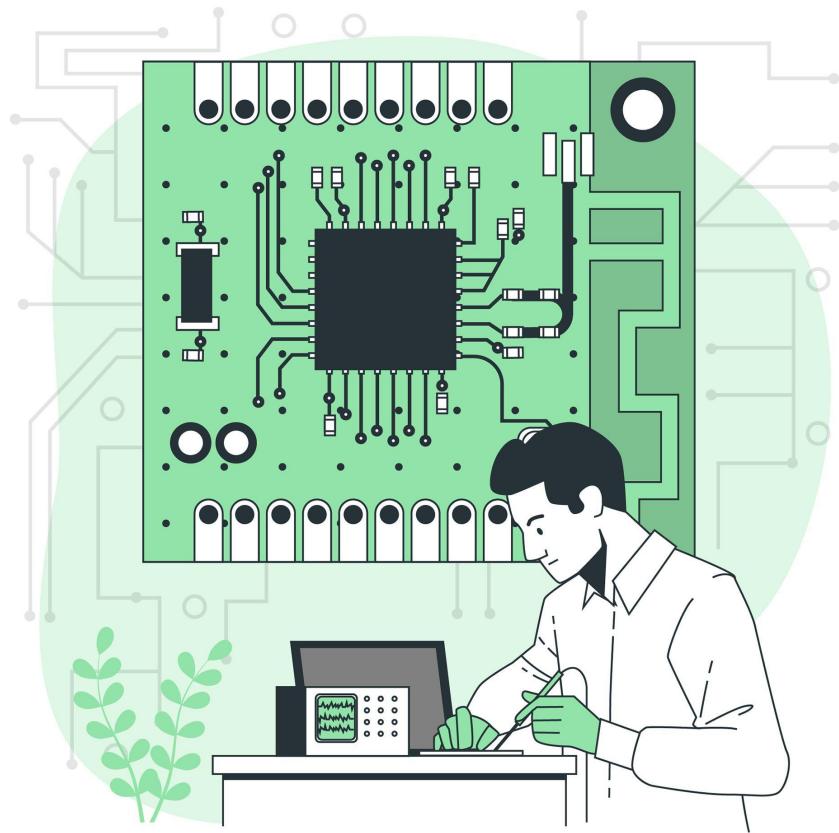


01

Giriş



Giriş

- İşletim Sistemi Nedir?
- İşletim Sistemlerinin Tarihçesi
- Bilgisayar Donanımı Genel Bakış
- İşletim Sistemi Mimarisi
- Sistem Çağrıları
- Üç Başlıkta İşletim Sistemleri



Giriş

Bilgisayar Sistemleri

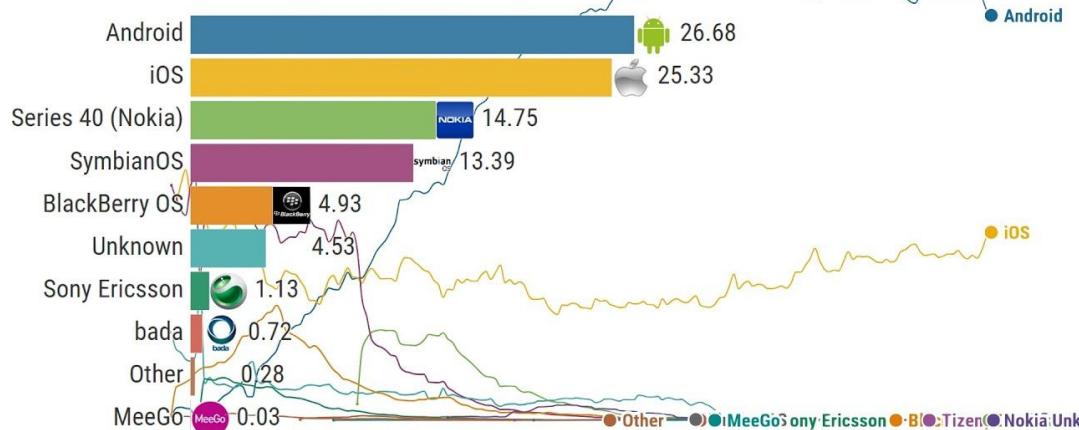


İşletim Sistemi



Giriş

Most Used Mobile OS Timelapse (2009-2023)

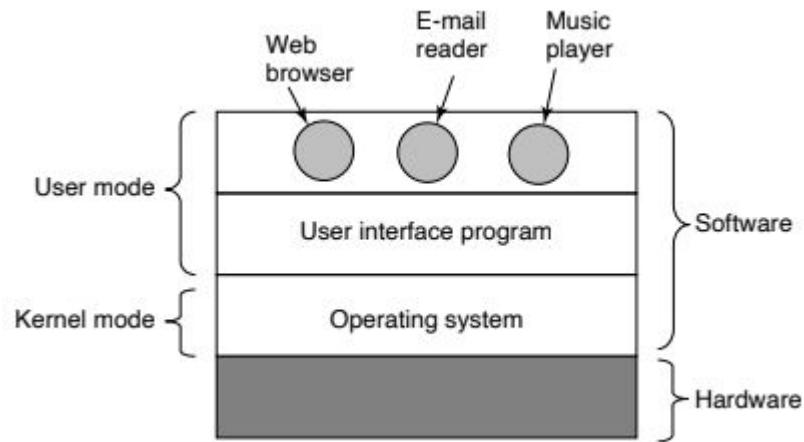


Giriş

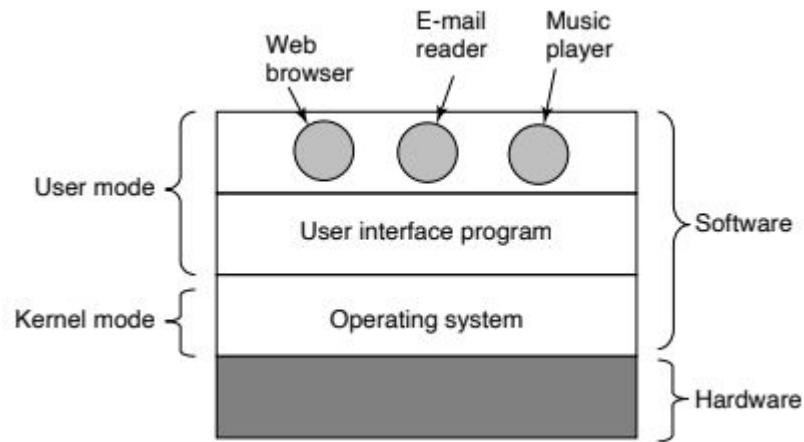


UNIX

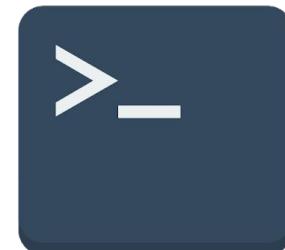
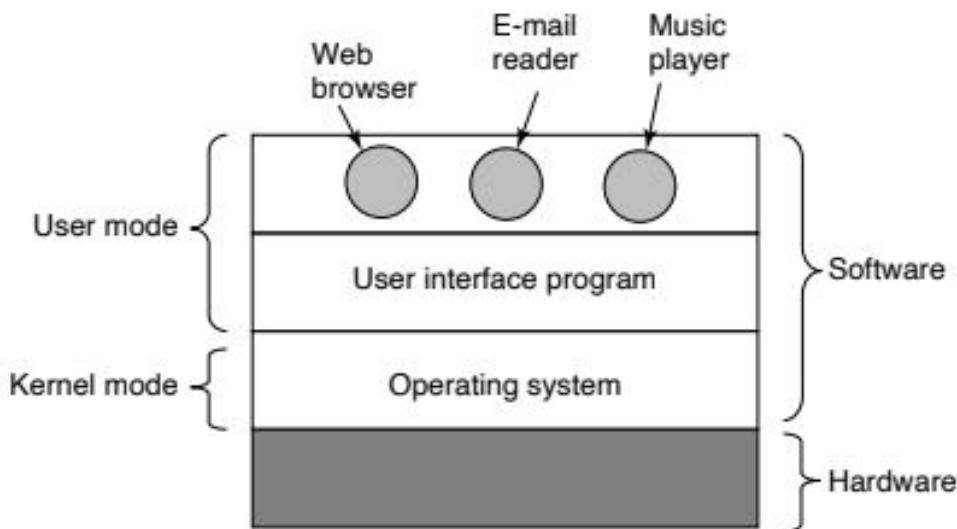
Giriş



Giriş



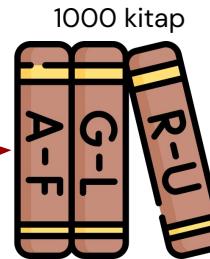
Giriş



LINES OF CODES USED



Giriş



1000 sayfa



50 satır



20 kitap x 50 raf



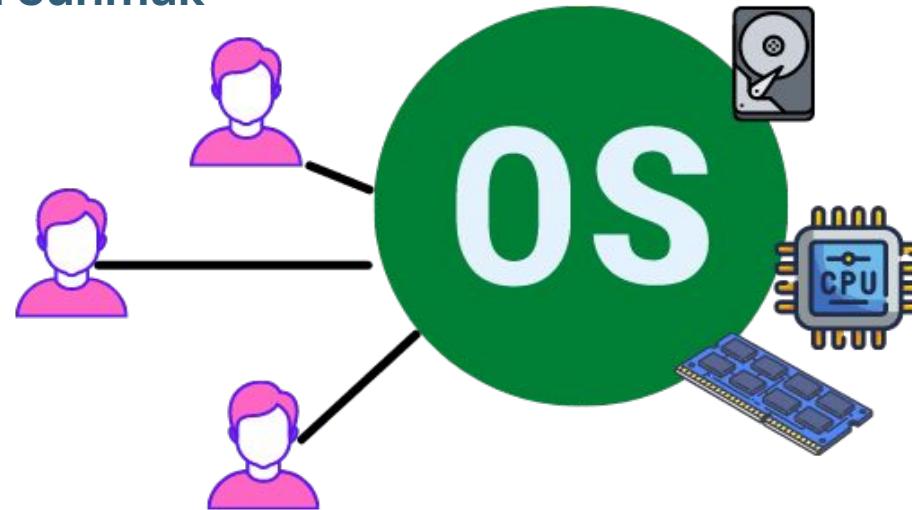
02

İşletim Sistemi Nedir?



İşletim Sistemi Nedir?

- Donanım Kaynaklarını Yönetmek
- Donanım Kaynaklarını Sunmak

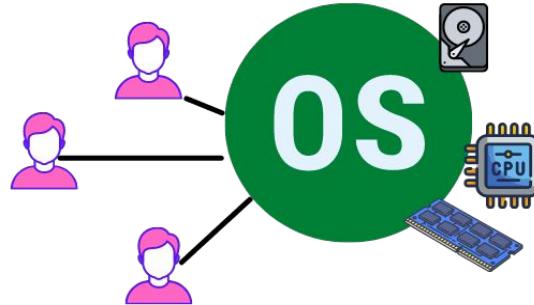


Donanım Yığını olarak İşletim Sistemi



Donanım Yığını olarak İşletim Sistemi

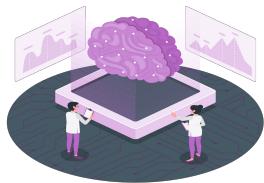
```
suhap@suhap-thinkpad-x1-yoga-gen-7:~$ googler www.kocaeli.edu.tr
1. Kocaeli Üniversitesi
https://www.kocaeli.edu.tr/
Universitemizden ÖNEMLİ GELİŞMELER - 2023-2024 Eğitim - Öğretim Yılı Üniversitemiz Akademik Takvimi - İngilizce
Veterlik Sınavı (YGS-Güz) - Kocaeli üniversitesi ...
2. Kocaeli Üniversitesi
https://www0.kocaeli.edu.tr/
3. Kocaeli Üniversitesi
https://www.turkiye.gov.tr/kocaeli-universitesi
4. Kocaeli Üniversitesi | zzzit
https://www.facebook.com/kou9zofficial/?locale=id_ID
5. Kocaeli Üniversitesi | Izmit
https://www.facebook.com/kou9zofficial/?locale=tr_TR
6. KOSTÜ | Kocaeli Sağlık ve Teknoloji Üniversitesi - Geleceği ...
https://kocaelisaglik.edu.tr/
7. kou9zofficial - Kocaeli Üniversitesi
https://www.instagram.com/kou9zofficial/
googler (? for help)
```



Kaynak Yöneticisi olarak İşletim Sistemi



İşlemci



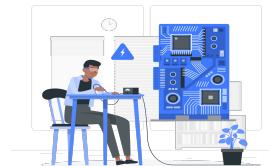
Ekran



Disk



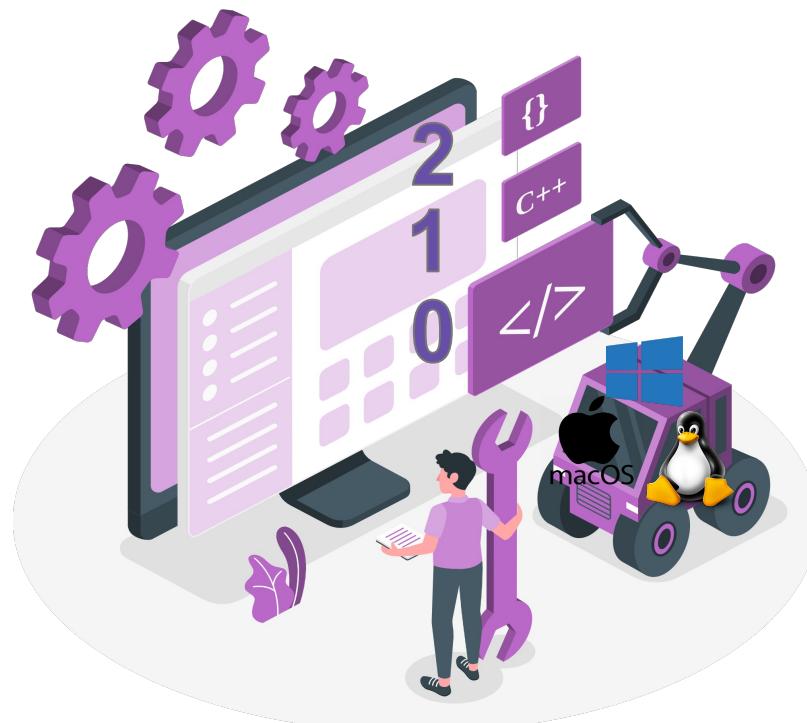
Ana Kart



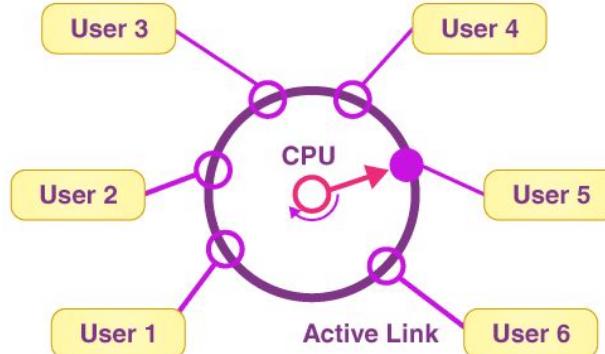
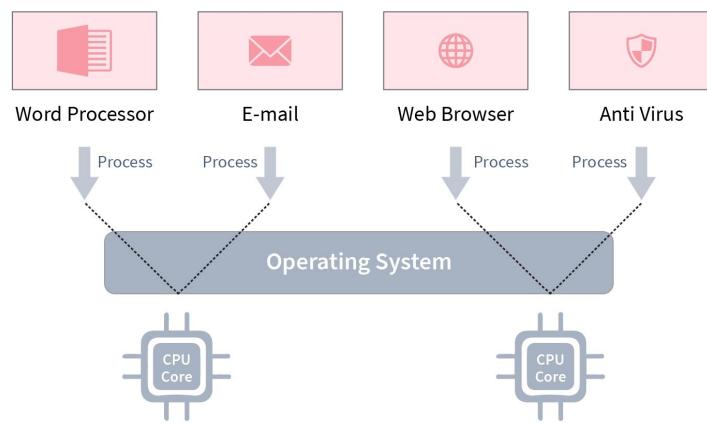
YÖNETİM



Kaynak Yöneticisi olarak İşletim Sistemi

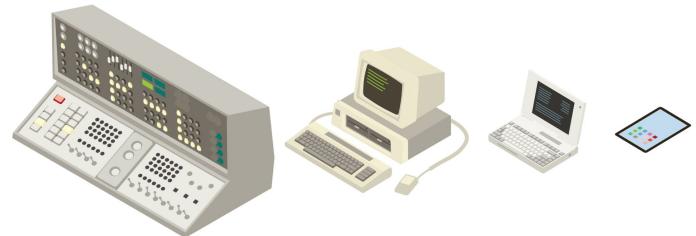


Kaynak Yöneticisi olarak İşletim Sistemi

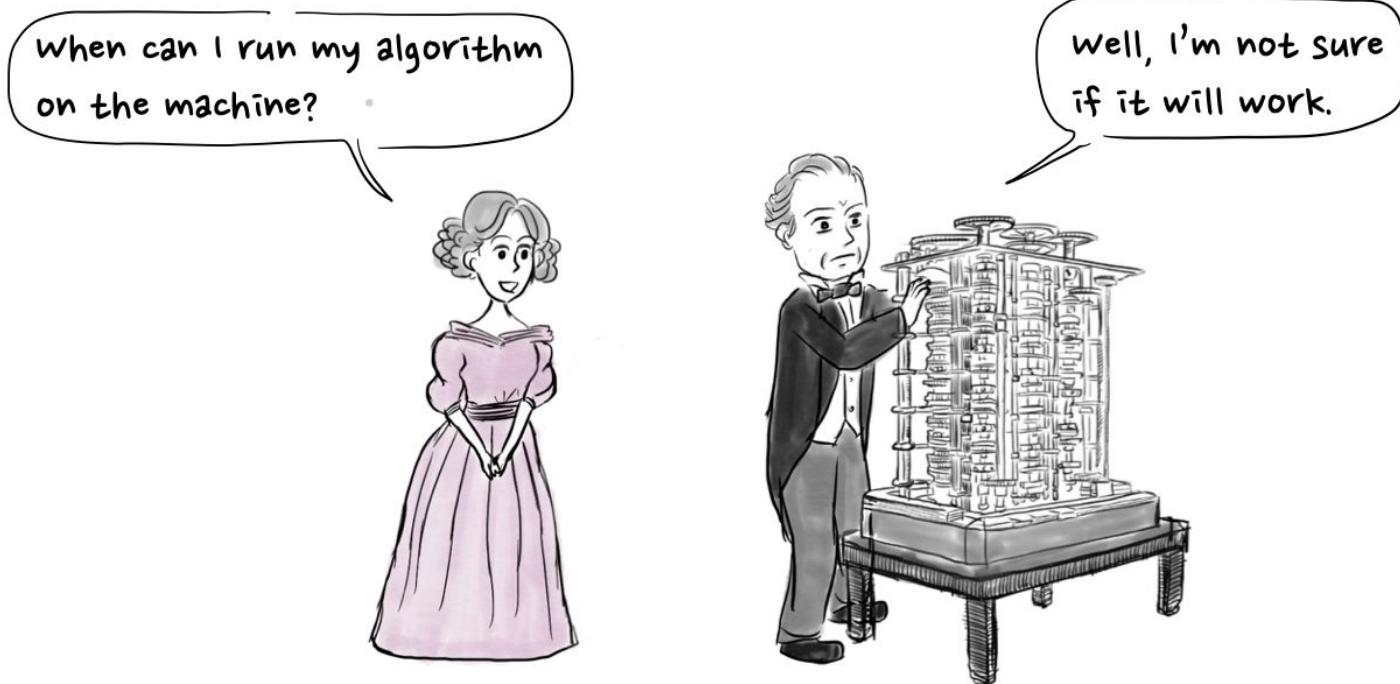


03

İşletim Sistemi Tarihçe

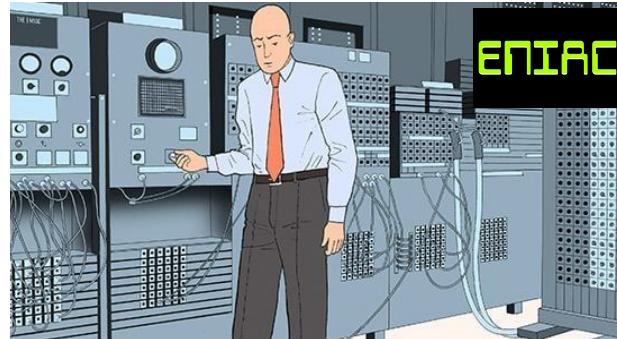
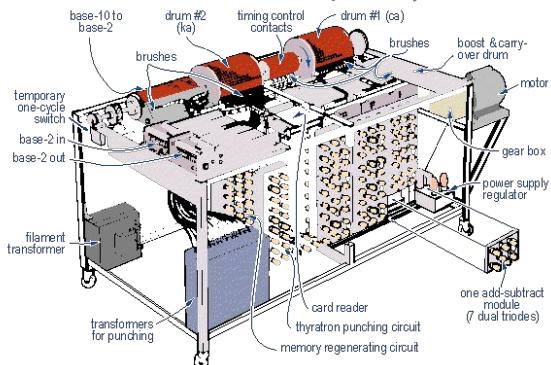


Tarihçe

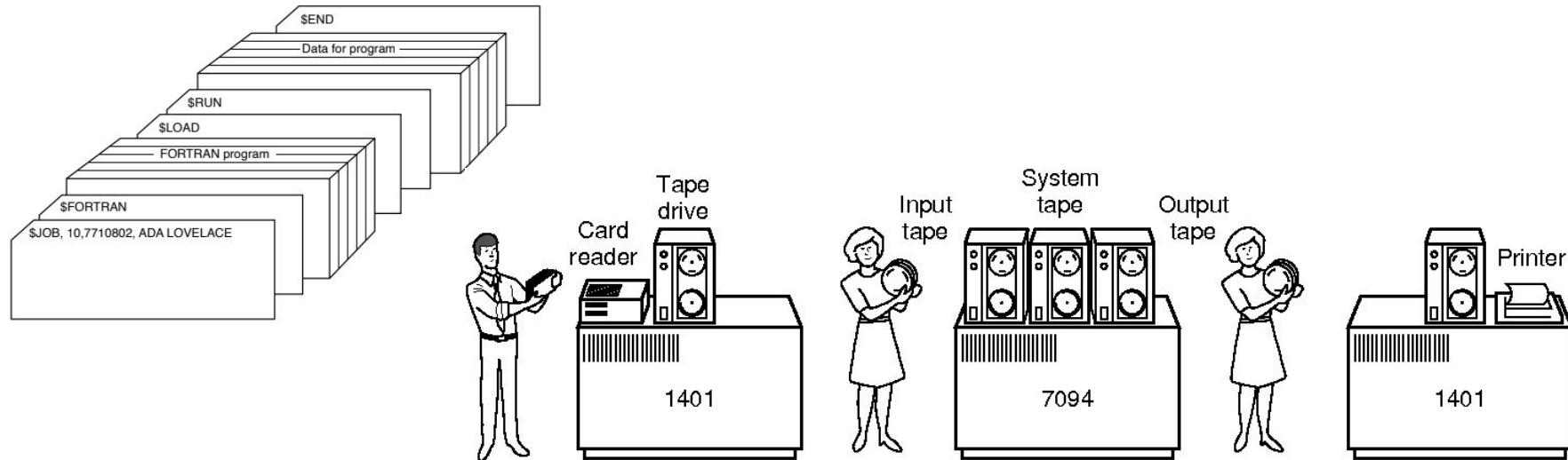


Birinci Nesil (1945-1955): Vakum Tüpler

The Atanasoff-Berry Computer



İkinci Nesil (1955-1965): Transistörler ve Toplu Sistemler



Üçüncü Nesil (1965-1980): Tümlüşik Devreler ve Çoklu Programlama

IBM 1401 Computer System



IBM, 370, 4300, 3080 ve 3090



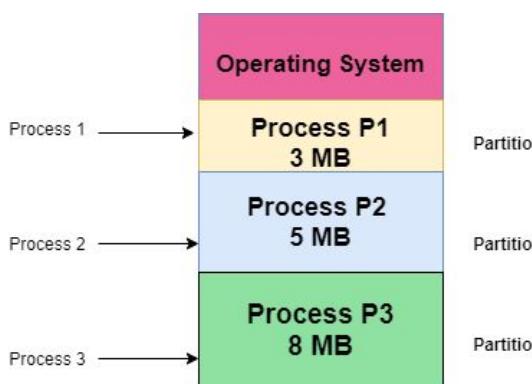
IBM 7094



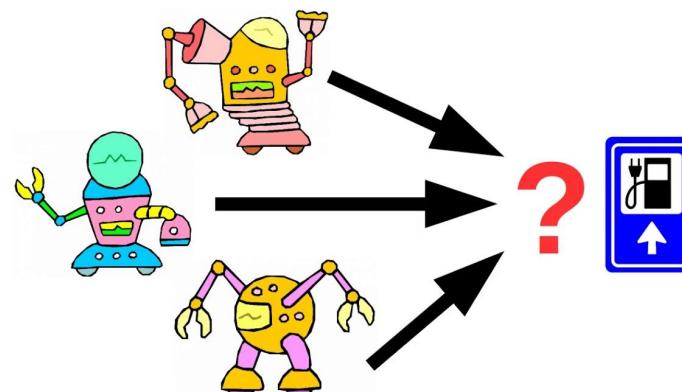
IC'leri (Integrated Circuits)



Üçüncü Nesil (1965-1980): Tümlüşik Devreler ve Çoklu Programlama



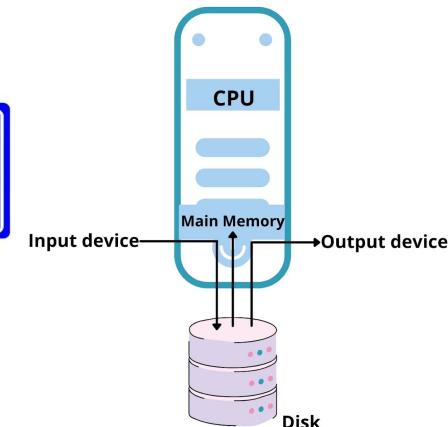
Size of Partition = Size of Process



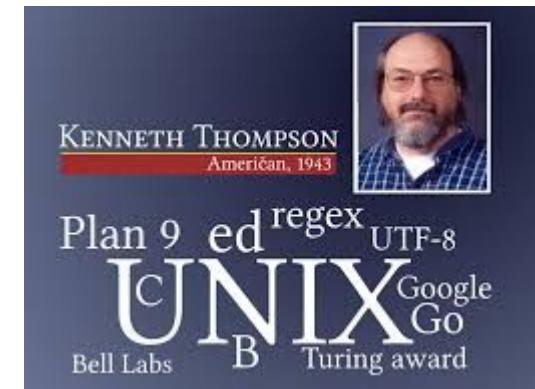
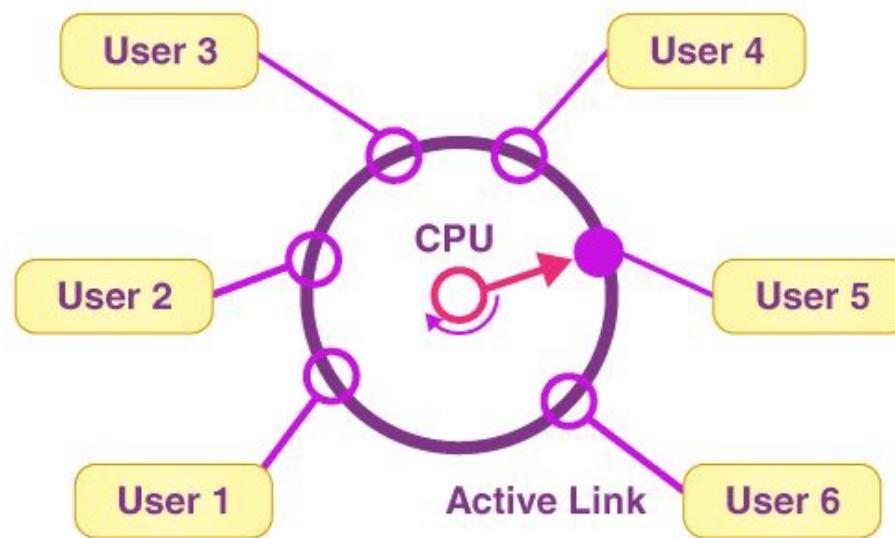
© Authors of ICRA 2018 Paper 24

Wed AM

Pod S.1



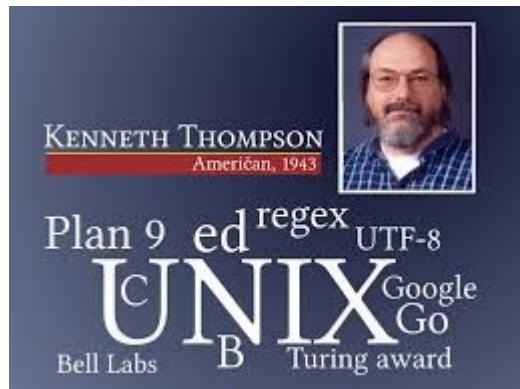
Üçüncü Nesil (1965-1980): Tümlüşik Devreler ve Çoklu Programlama



Üçüncü Nesil (1965-1980): Tümlüşik Devreler ve Çoklu Programlama



MINIX 3



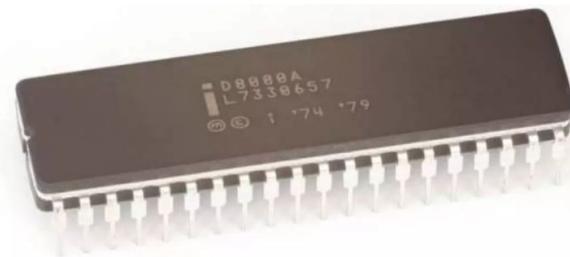
Dördüncü Nesil (1980-Günümüz): Kişisel Bilgisayarlar



Dördüncü Nesil (1980-Günümüz): Kişisel Bilgisayarlar



 DIGITAL
RESEARCH

The Digital Research logo consists of a stylized square icon containing two vertical bars of different heights, followed by the company name in a serif font.

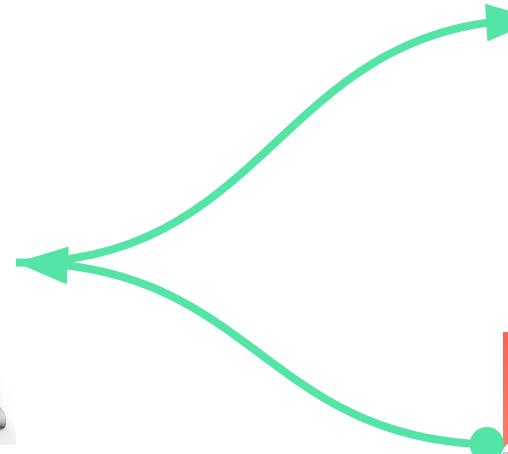
 8080 processor

The Intel logo is shown in its signature blue and white circular font, followed by the text "8080 processor".

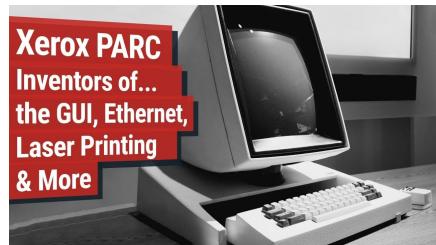
Dördüncü Nesil (1980-Günümüz): Kişisel Bilgisayarlar



GATES, BILL



Dördüncü Nesil (1980-Günümüz): Kişisel Bilgisayarlar



Dördüncü Nesil (1980-Günümüz): Kişisel Bilgisayarlar



1985 - 2001



1990 - 2001



1992 - 2001



1993 - 2001



1994 - 2001



GATES, BILL



1995 - 2001

1996 - 2004

1998 - 2006

2000 - 2006

2000 - 2010



2001 - 2014

2006 - 2017

2009 - 2020

2012 - 2016

2013 - now



2015 - now

2020 - now

2021 - now

Introducing Macintosh
January 24, 1984



Beşinci Nesil (1990-Günümüz): Mobil Bilgisayarlar

1938

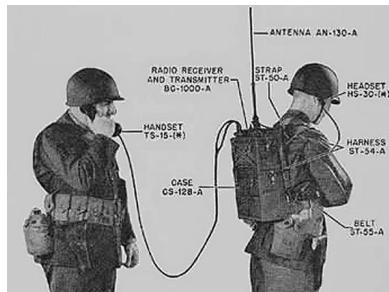
SCR-194 ve 195



ABD
Taşınabilir telsiz
11 Kg
8 Km

1940

SCR-300



Motorola
Taşınabilir radyo
17 Kg
5 Km

1942

SCR-536



Motorola
Handie Talkie
2 Kg
1,5 Km

Beşinci Nesil (1990-Günümüz): Mobil Bilgisayarlar

1946

Mobil Telefon Sistemi (MTS)



Bell
80 Kg
5 Km
Arama başına ayda 330 \$

1956

Mobil Sistem A (MTA)



Ericsson
40 Kg
5 Km

1964

Mobil Telefon Hizmetinin (IMTS)



Motorola
Mobil Telefon Hizmeti
18 Kg
Yüksek Fiyat
Kısıtlı kullanım

1982

DynaTAC
(Dinamik Uyarlanabilir Toplam Alan Kapsamı)



Motorola
10 Kg
1G
60 dak. konuşma

Beşinci Nesil (1990-Günümüz): Mobil Bilgisayarlar

1983
DynaTAC



Motorola
1 Kg
9000\$

1984
Mobira Talkman



Mobira
Uzun Konuşma Süresi

1992
International 3200



Motorola
2G

1993
IBM Simon



IBM
Akıllı telefon
-> Çağrı cihazı,
-> Faks makinesi
-> PDA

1997
Nokia Communicator



Nokia
Akıllı telefon
-> LCD ekranı,
-> QWERTY klavye

Beşinci Nesil (1990-Günümüz): Mobil Bilgisayarlar

2007

Apple iPhone



Apple
Dokunmatik ekran
3G

2008

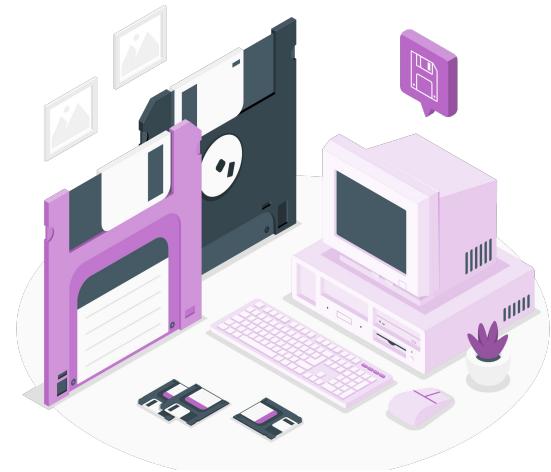
HTC Dream



HTC
Android OS

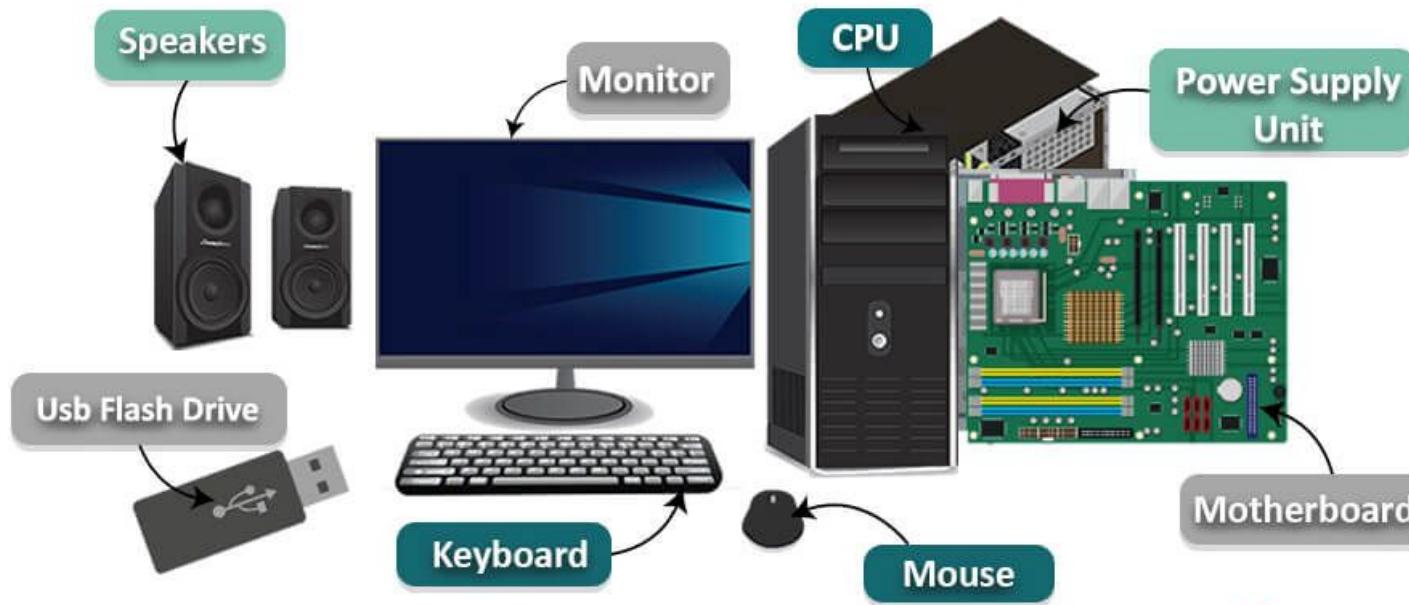
04

Bilgisayar Donanımı

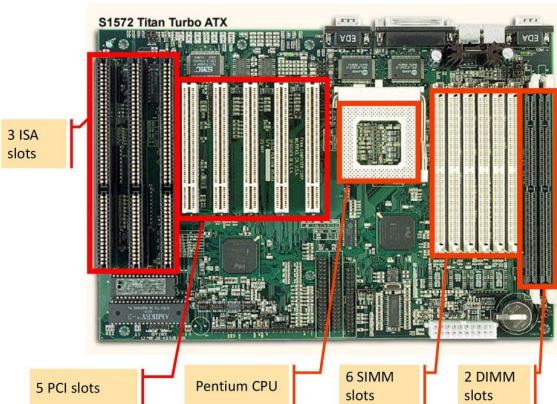


Giriş

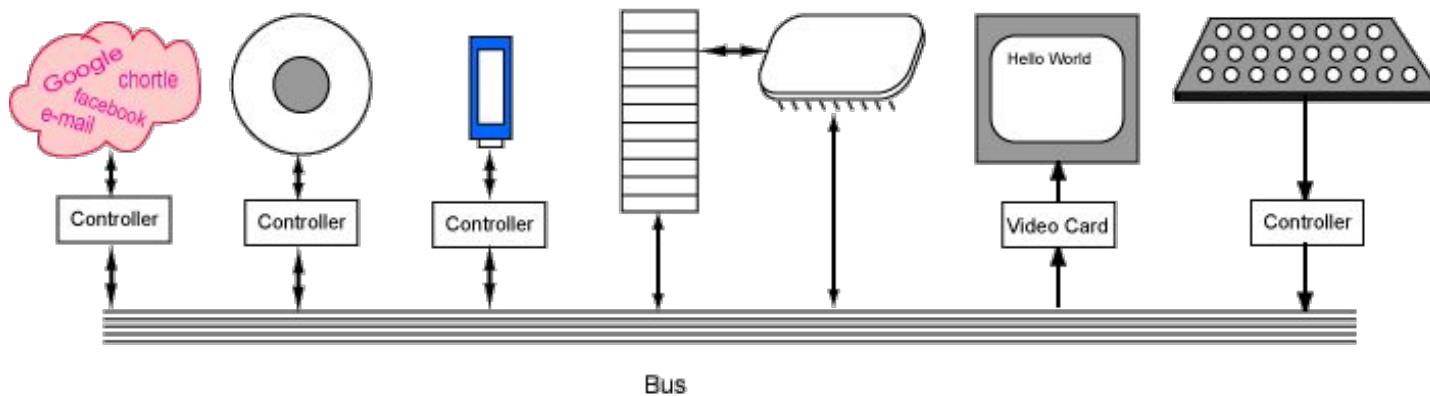
Components of Computers



Giriş

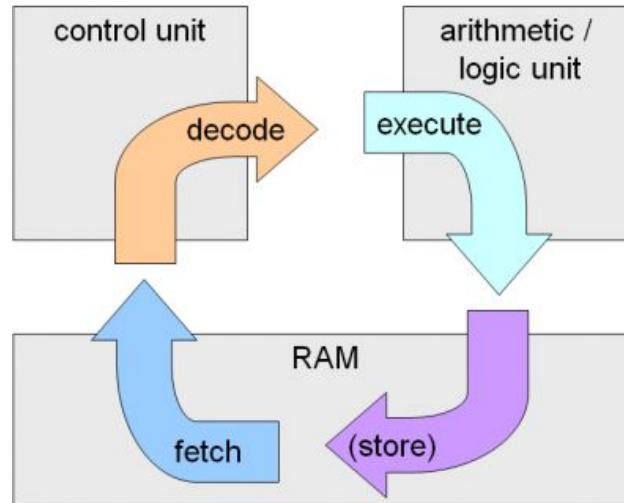
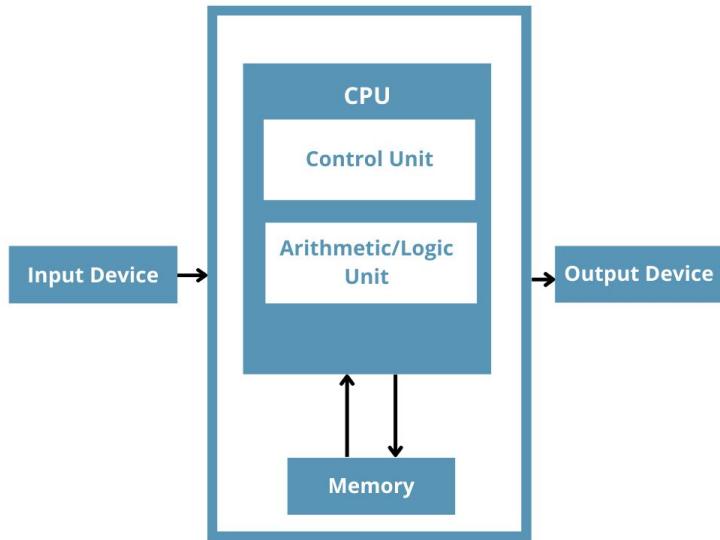
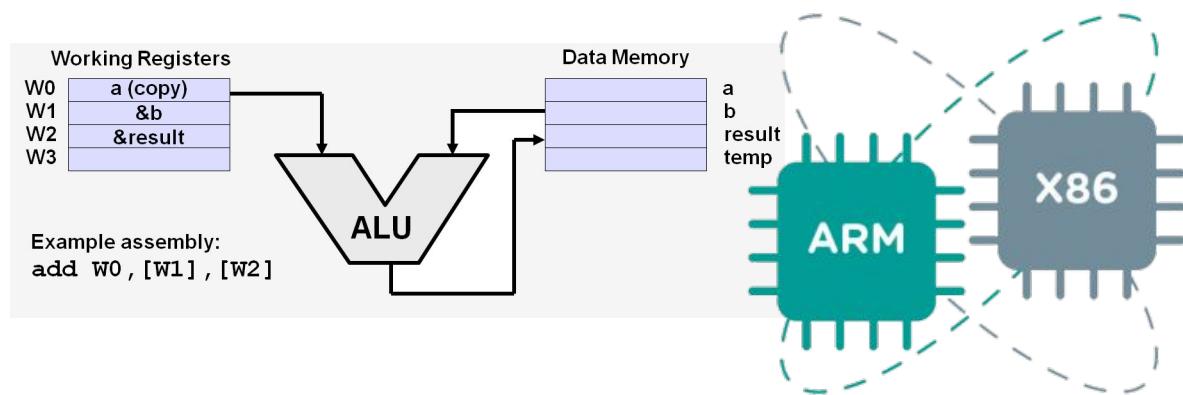


Network Hard Disk USB Drive Main Memory Processor Monitor Keyboard



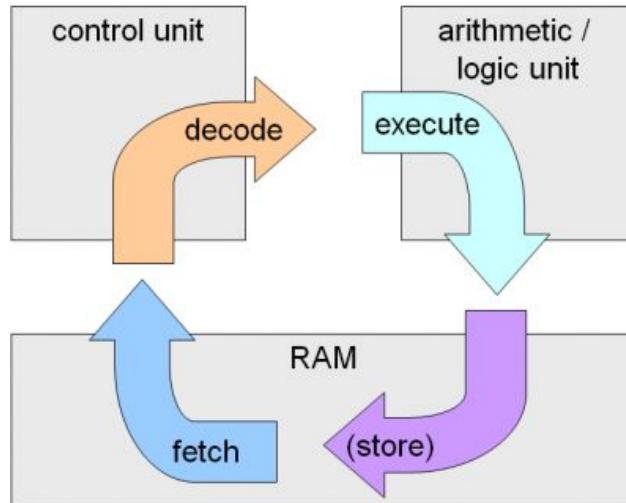
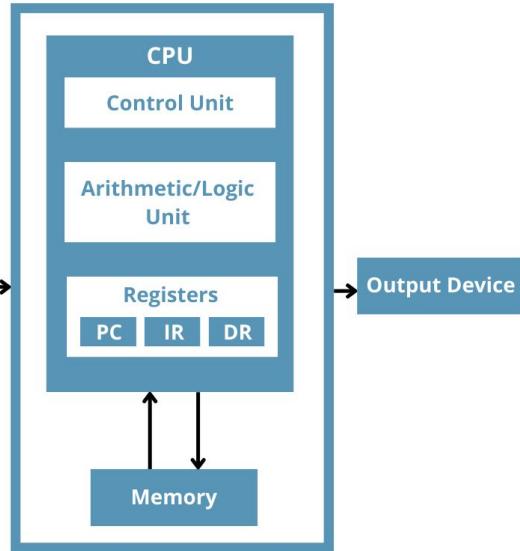
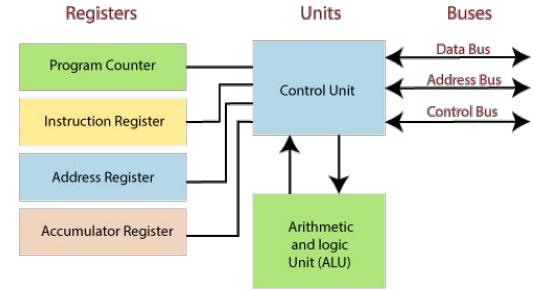
Main Components of a Computer System

İşlemci CPU

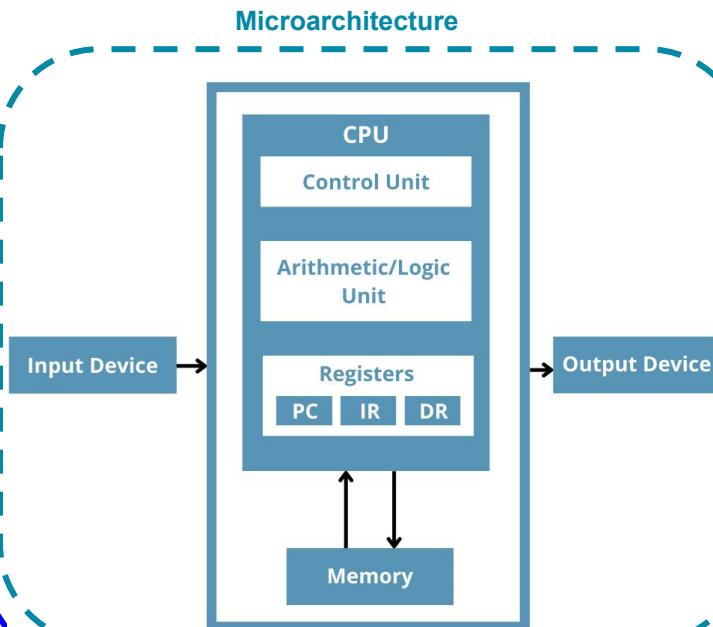


İşlemci CPU

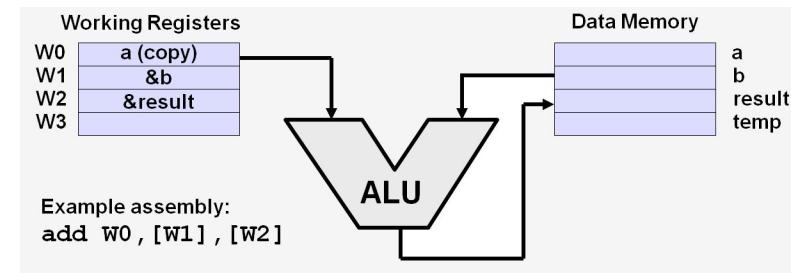
$t_{\text{fetch}} > t_{\text{execute}}$



İşlemci CPU



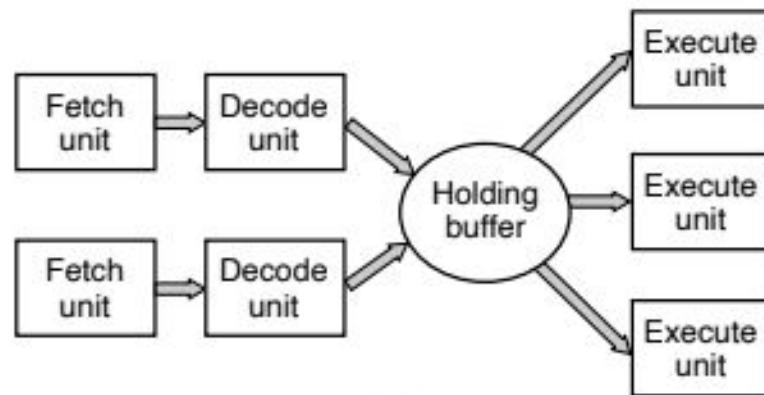
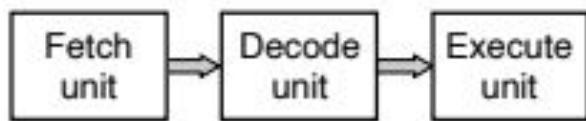
Architecture



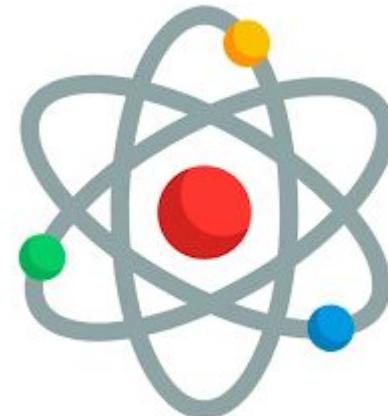
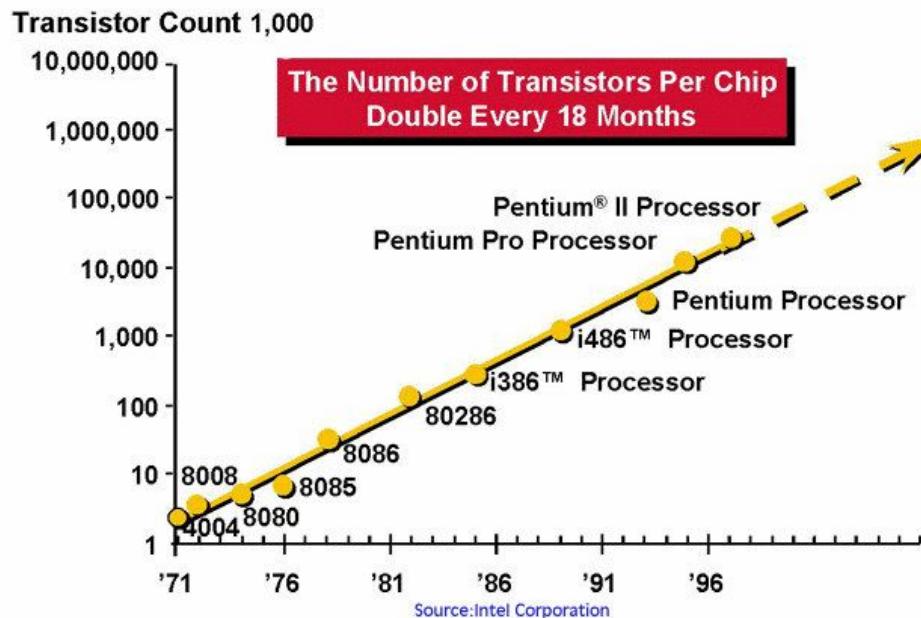
İşlemci CPU



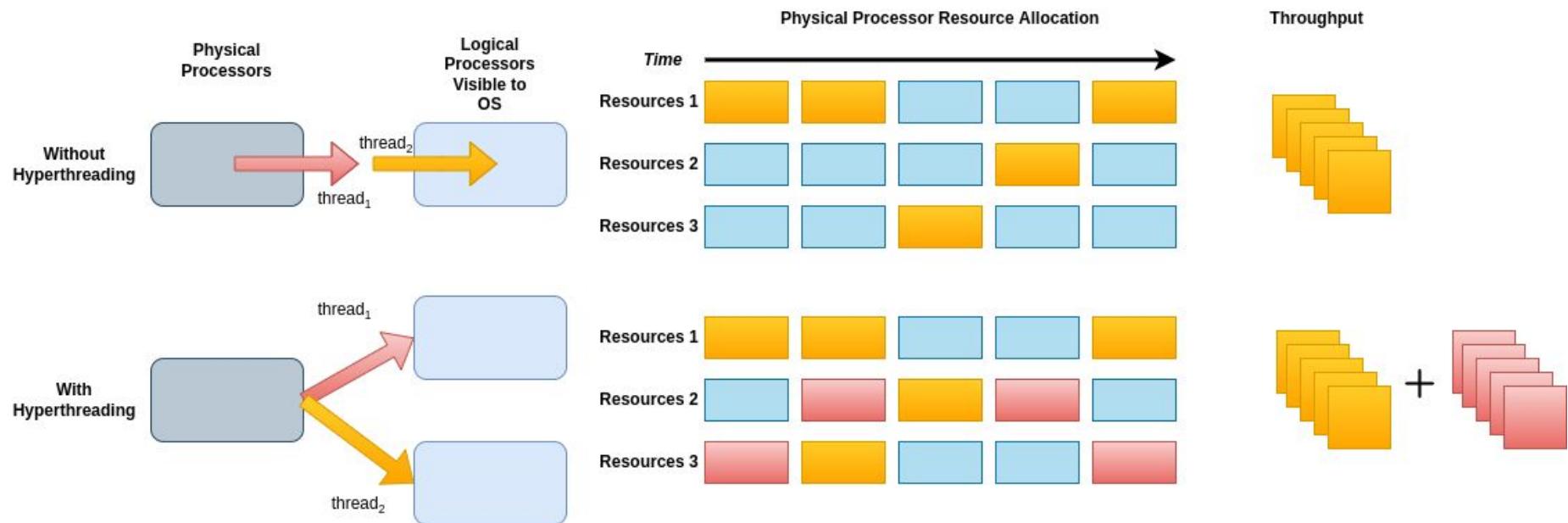
İşlemci CPU



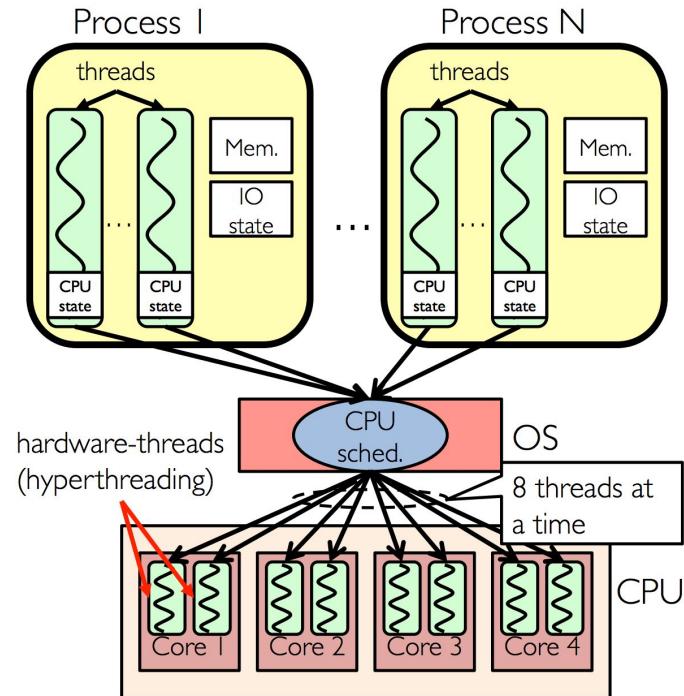
Çok İş Parçacıklı ve Çok Çekirdekli CPU



Çok İş Parçacıklı ve Çok Çekirdekli CPU

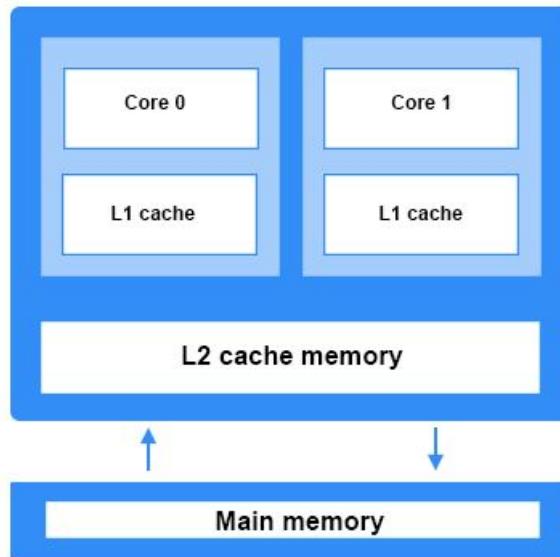


Çok İş Parçacıklı ve Çok Çekirdekli CPU

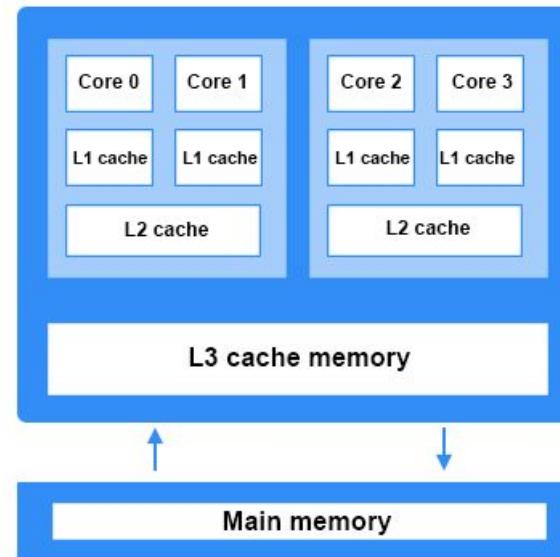


Çok İş Parçacıklı ve Çok Çekirdekli CPU

Dual-core CPU:

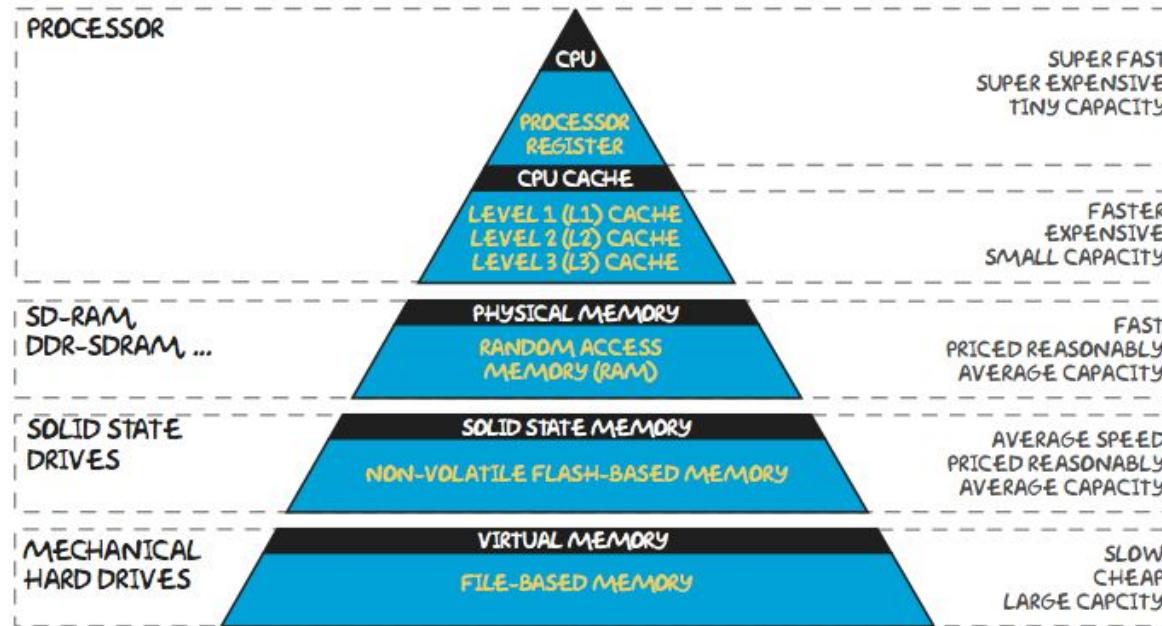


Quad-core CPU:



Bellek

THE MEMORY HIERARCHY



Bellek

32 bit CPU 32×32 bit

64 bit CPU 64×64 bit

THE MEMORY HIERARCHY

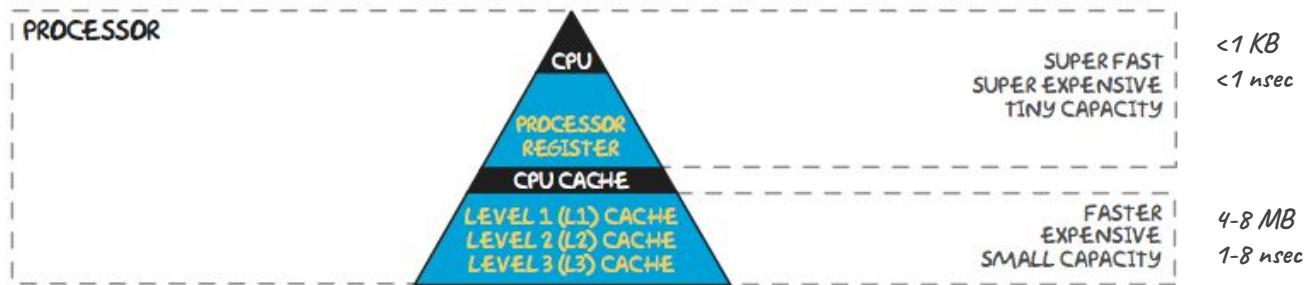


Bellek

32 bit CPU 32×32 bit
64 bit CPU 64×64 bit

0. satır 0 & 63 arasındaki adresler
1. satır 64 & 127 arasındaki adresler
....

THE MEMORY HIERARCHY



L1 önbellek:

Yoğun olarak kullanılan dosyalar,
dosya yolları (/home/projects/minix3/src/kernel/clock.c),
web adresi (URL ->IP)
vb.

Yürüttülecek kodu gözülmüş talimatları tutar
Çok yoğun kullanılan veri sözcükleri tutar
32 KB boyutundadır

L1 önbelleğe erişim: Gecikmesiz

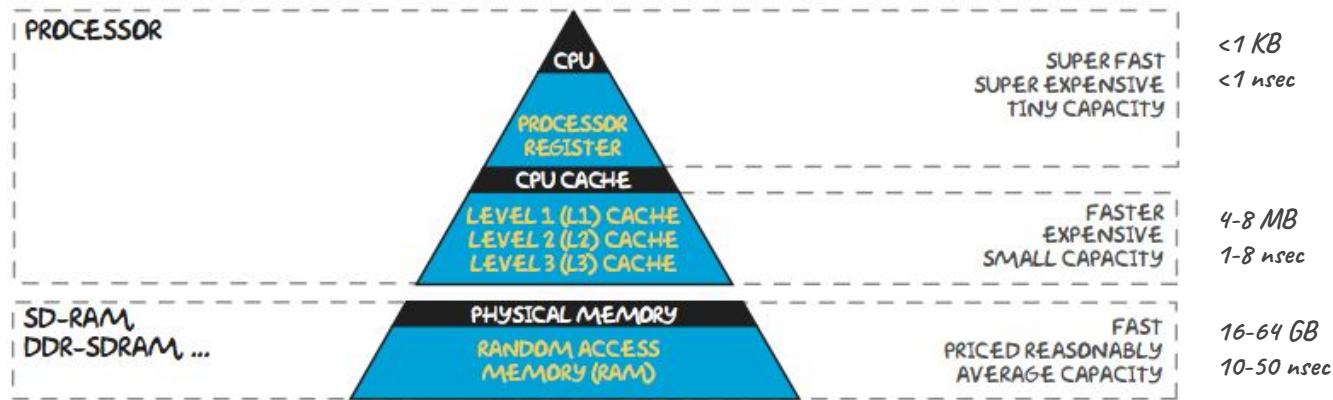
L2 önbelleğe erişim: Birkaç saat döngüsü gecikme

Bellek

THE MEMORY HIERARCHY

32 bit CPU 32×32 bit
64 bit CPU 64×64 bit

0. satır 0 & 63 arasındaki adresler
1. satır 64 & 127 arasındaki adresler
....



Önbellekten karşılanamayan tüm CPU istekleri ana belleğe gider.

ROM (Salt Okunur Bellek): Programlandıktan sonra değiştirilemez. Hızlı ve ucuzdur. bootstrap yükleyicisi ROM'da bulunur.

RAM (Random Access Memory):

EEPROM(Electronically Erasable Read-Only Memory): Uçucu değildir uzun süre de silinip yeniden yazılabilir.

Flash Bellek: Uçucu değildir ancak silinip yeniden yazılabilir. Hızlı bir alternatif olarak SSD'lerde kullanılır. Çok fazla kez silinirse yıpranır.

CMOS(Complementary Metal Oxide Semiconductor). Saati ve tarihi tutmak için ve yapılandırma parametreleri (hangi sürücüden önyükleme yapacağı) tutar.

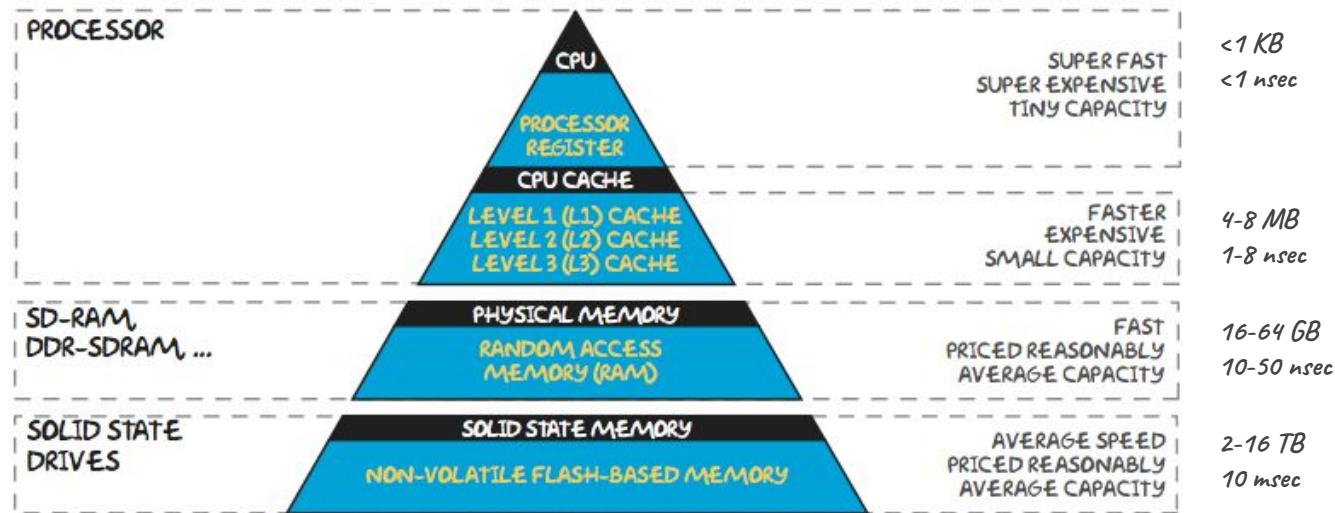
Sanal bellek sistemi: Ana belleği önbellek olarak kullanır. Sanal adresi, fiziksel adrese dönüştürmeyi CPU'daki MMU (Bellek Yönetim Birimi) yapar.

Bellek

32 bit CPU 32×32 bit
64 bit CPU 64×64 bit

0. satır 0 & 63 arasındaki adresler
1. satır 64 & 127 arasındaki adresler
....

THE MEMORY HIERARCHY



Fiziksel olarak disk olmamalarına (plakaları ya da hareketli kolları) rağmen SSD disk olarak kabul edilir.

Verileri Flash bellekte depolar. Uçucu değildirler.

Hızlı olmalarına rağmen bayt başına maliyet açısından dönen disklerden çok daha pahalıdır.

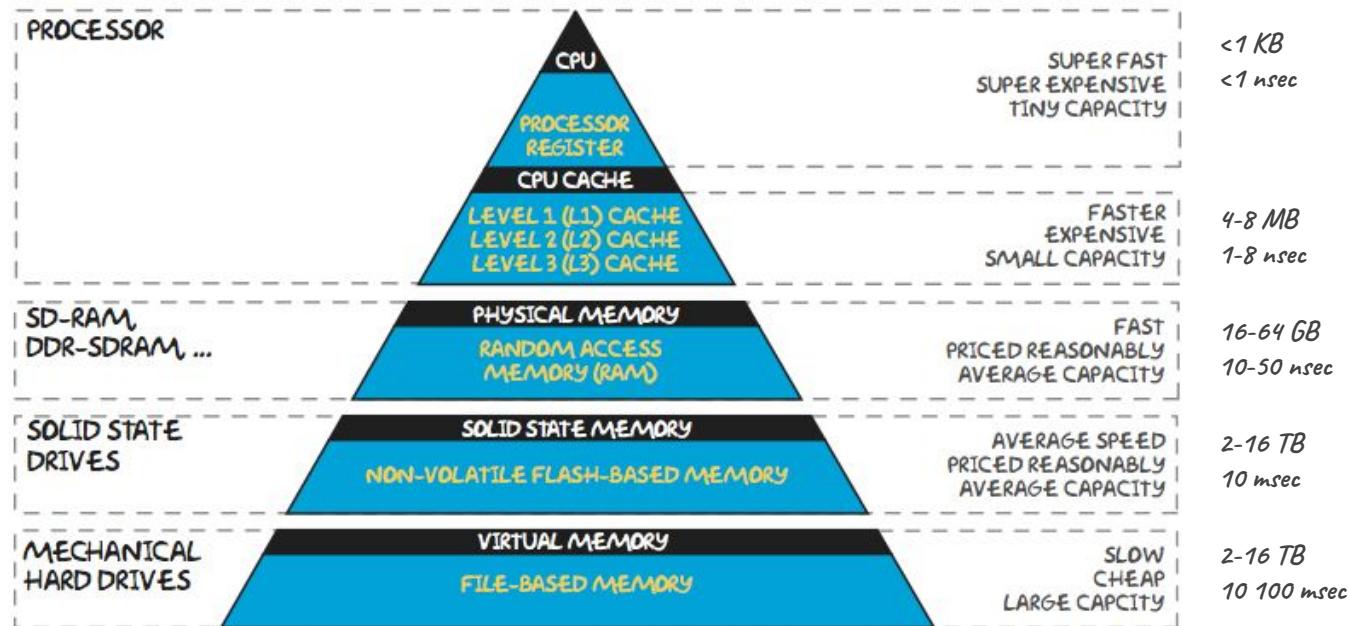
Mekanik bir kol olmadığından rastgele konumlardaki verilere erişebilirler.

Bellek

32 bit CPU 32×32 bit
64 bit CPU 64×64 bit

- 0. satır 0 & 63 arasındaki adresler
- 1. satır 64 & 127 arasındaki adresler
- ...

THE MEMORY HIERARCHY



Bellek

Disk (5400, 7200, ... RPM) hızında dönen plakalardan oluşur.

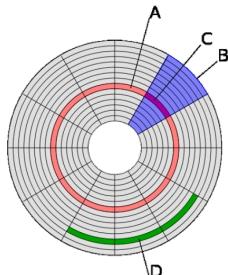
Mekanik bir kol plakaların üzerinde döner.

Bilgi disk üzerine daire şeklinde yazılır. Mekanik kol ucundaki kafa iz olarak bilinen dairesel bir bölgeyi okuyabilir.

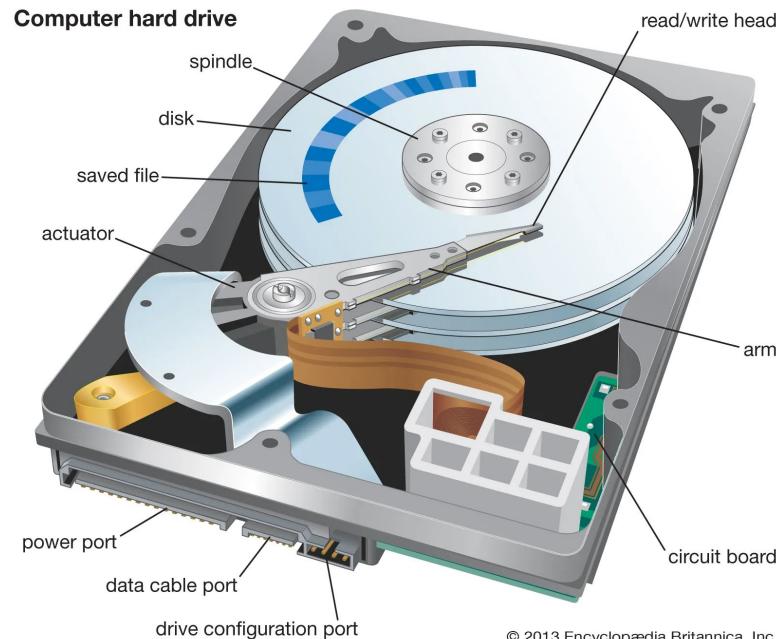
İzler, saniye başına belli sayıda sektör'e bölünmüştür.

Kolu bir silindirden diğerine taşımak yaklaşık 1 msn sürer.

Kol, sürücünün sektörün kafanın altında dönmesini bekler; gecikmeye (5-10 msn) neden olur.



Hard Drive Structure:
A = track
B = sector
C = sector of a track
D = cluster

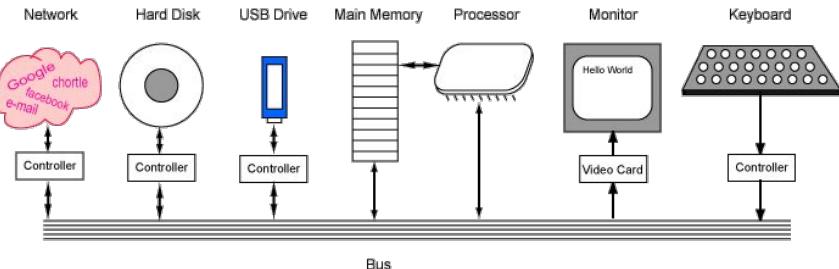


I/O Cihazları

SATA

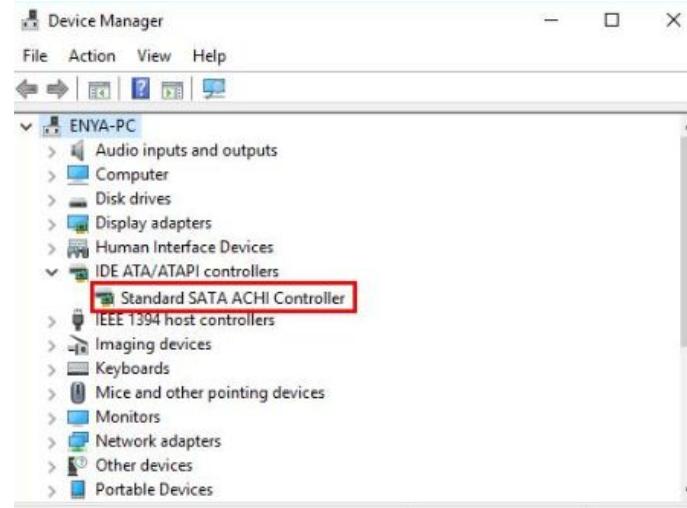


SATA Denetleyici

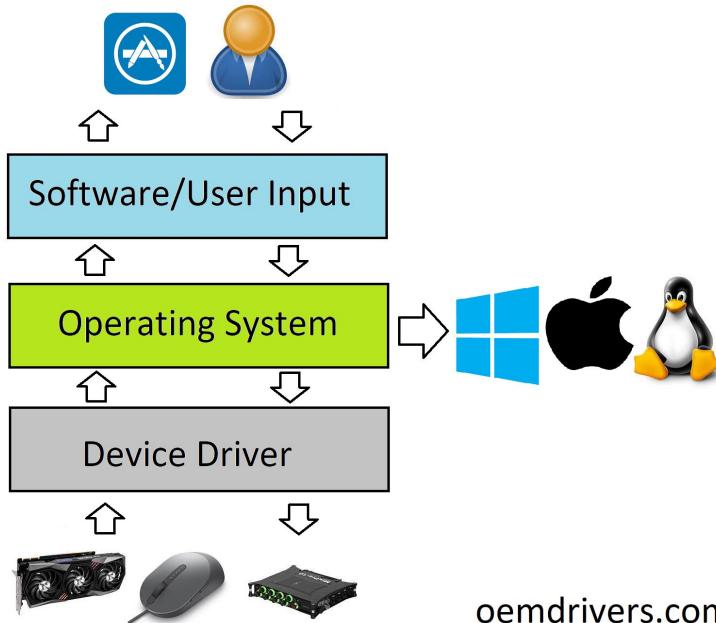


Main Components of a Computer System

SATA Windows Sürücü



I/O Cihazları

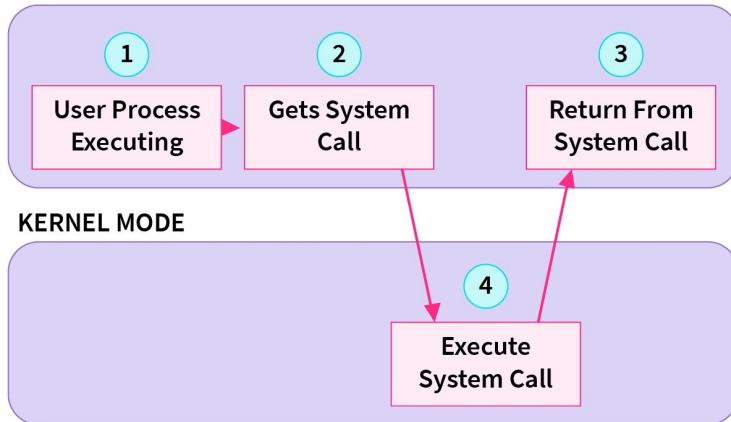


Sürücünün çekirdeğe yerleştirilmesinin üç yolu vardır.

1. Çekirdeğe yeni sürücüyü bağlamak ve sistemi yeniden başlatmaktır.
2. İşletim sistemi dosyasına sürücüye ihtiyaç duyduğunu belirten bir giriş yapmak ve sistemi yeniden başlatmaktır. Önyükleme sırasında işletim sistemi gidip ihtiyaç duyduğu sürücülerini bulur ve yükler.
3. İşletim sisteminin yeni sürücülerini galırken kabul edebilmesi ve yeniden başlatmaya gerek kalmadan anında yükleyebilmesidir.

I/O Cihazları

USER MODE

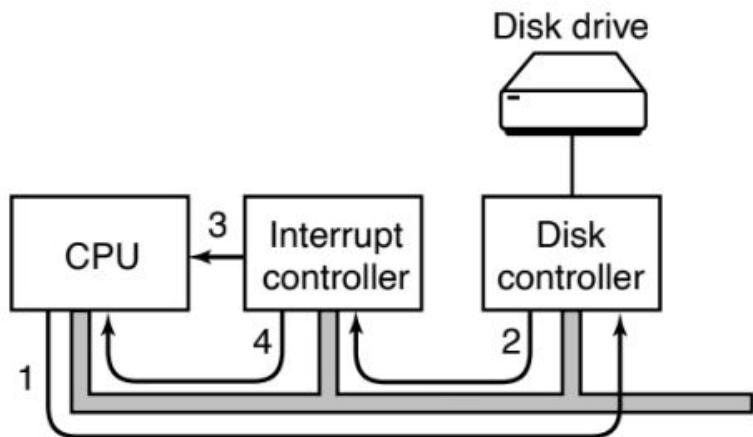


KERNEL MODE

Giriş ve çıkış yöntemi (Meşgul Bekleme)

1. Kullanıcı programı bir sistem çağrıyı yayınlar
2. Çekirdek bunu uygun sürücü için bir prosedür çağrısına çevirir.
3. Sürücü daha sonra G/Ç'yi başlatır
4. Sıkı bir denetim döngüsü içinde işlemin tamamlandığını kontrol için cihazı sürekli olarak yoklanır (genellikle cihazın hala meşgul olduğunu gösteren bir bit vardır).
5. G/Ç tamamlandığında, sürücü verileri ilgili saklayıcılara yazar
6. İşletim sistemi daha sonra kontrolü çağrıya geri verir.

I/O Cihazları



Giriş ve çıkış yöntemi (Kesme)

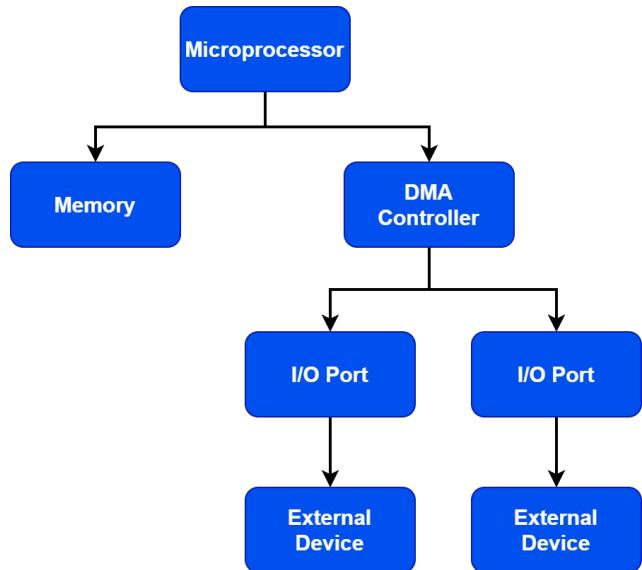
Adım 1-) Sürücü, aygit saklayıcılarını kullanarak denetleyiciye ne yapması gerektiğini söyler. Denetleyici aygıtını başlatır.

Adım 2-) Aygit denetleyicisi, işini bitirdiğinde, kesme denetleyicisine sinyal gönderir.

Adım 3-) Kesme denetleyicisi kesmeyi kabul ederse (daha yüksek öncelikli bir kesmeyi işlemekle meşgulse hazır olmayıabilir), CPU üzerindeki bir pinin uyararak bunu bildirir.

Adım 4-) Kesme denetleyicisi cihazın numarasını CPU bildirir

I/O Cihazları



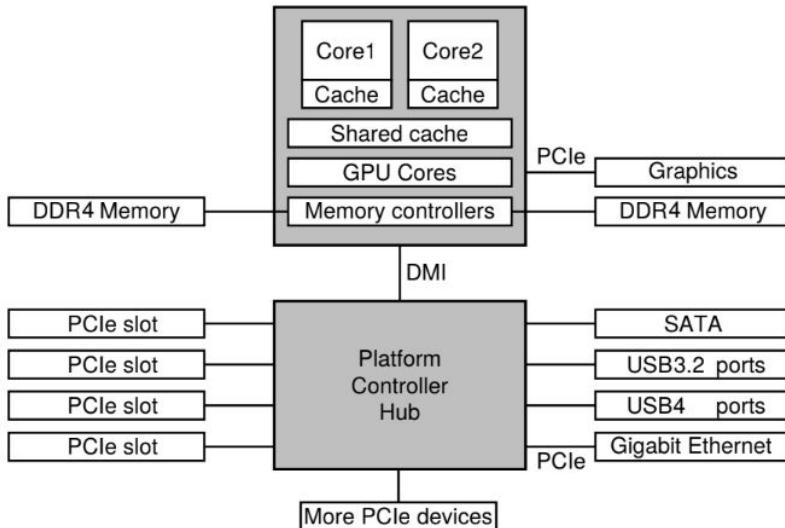
Giriş ve çıkış yöntemi (DMA Direct Memory Access)

Denetleyiciler arasındaki bit akışını kontrol eden DMA (Doğrudan Bellek Erişimi) yongası

CPU, DMA yongasını kurar (kaç bayt aktarılacağını, ilgili aygit ve bellek adreslerini bildirir)

DMA çipi işini bitirdiğinde, kesmeye tetikler.

Yollar



x86 Mimarisi

X86 Mimarisinde farklı aktarım hızına ve işleve sahip veri yolları vardır.
(PCI, USB, SATA ve DMI vb.) İşletim sistemi bu veri yollarının tümünü yönetmektedir.

Ana veri yolu PCIe (Peripheral Component Interconnect Express) veri yoludur.

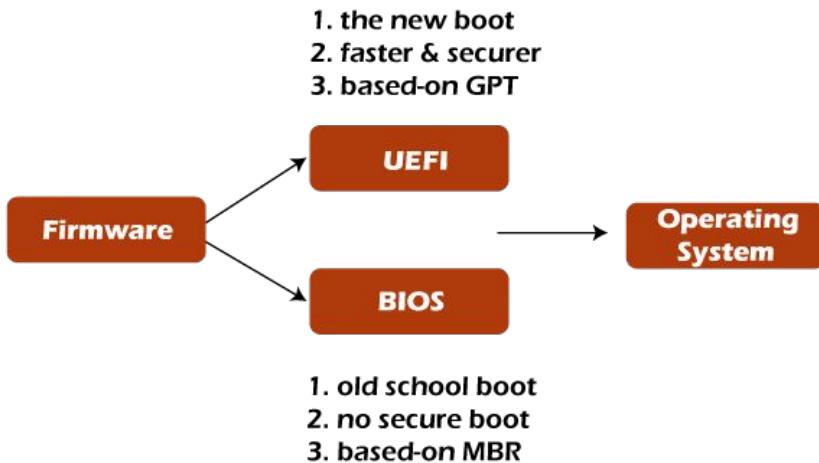
Intel tarafından eski PCI veri yolu'nun halefi olarak icat edilmiştir

Saniyede onlarca gigabit aktarma kapasitesine sahiptir

PCIe (2004) kadar, çoğu veri yolu paralel ve paylaşımlıydı.

PCIe özel, noktadan noktaya bağlantılar kullanır.

Önyükleme



Anakartdaki flaş diskte, BIOS (Temel Giriş Çıkış Sistemi) yazılımı barındırmaktadır.
BIOS kullanan önyükleme yapmak:

Yavaş önyükleme

Mimariye bağımlı

Küçük SSD'ler ve disklerle (2 TB'a kadar) sınırlı

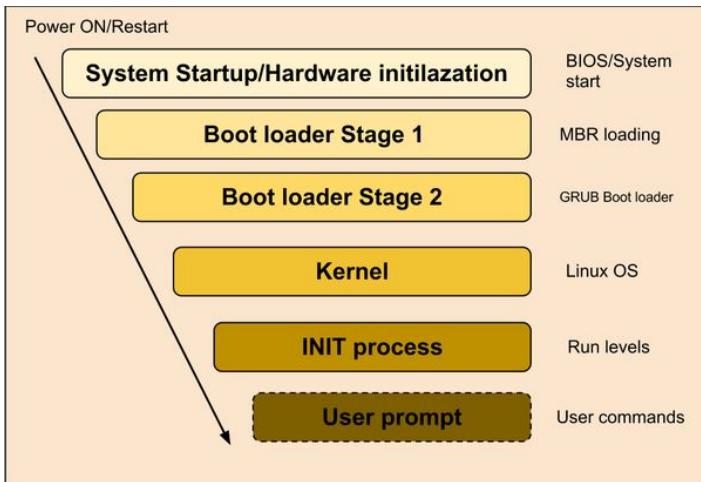
Intel, UEFI'yi (Unified Extensible Firmware Interface - Birleşik Genişletilebilir Ürün Yazılımı Arayüzü)

Hızlı önyüklemeye

Farklı mimarilere çalışma

8 ZiB kadar depolama

Önyükleme



CPU, başlatma kodunu anakarttaki flashta bulunan sabit bir adresten (sıfırlama vektörü olarak bilinir) alır.

Başlatma kodu, RAM, Platform Denetleyici Hub, PCI/ PCIe veri yollarına bağlı aygıtları tespit eder ve başlatır.

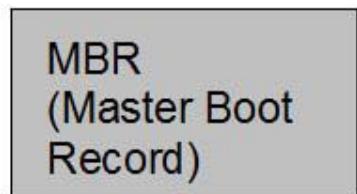
BIOS, CMOS belleğindeki listeden önyükleme aygıtını belirler.

Kullanıcı, önyüklemeden sonra BIOS programına girerek bu ön yükleme aygit listesini değiştirebilir.

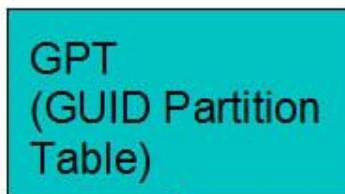
- 1-) Önyükleme aygıtındaki ilk sektör MBR, ikincil önyüklemeyi aktif eder
- 2-) ikincil önyükleme yükleyicisi işletim sistemini başlatır.
- 3-) İşletim sistemi BIOS'tan aygit sürücülerini sorgular ve gerekirse yükler.
- 4-) Arka plan işlemlerini çalıştırarak bir oturum açma programı veya GUI başlatır.

Önyükleme

BIOS



UEFI



1-) Önyükleme aygıtının ikinci sektöründeki GPT (GUID Bölümleme Tablosunun) yerini arar.

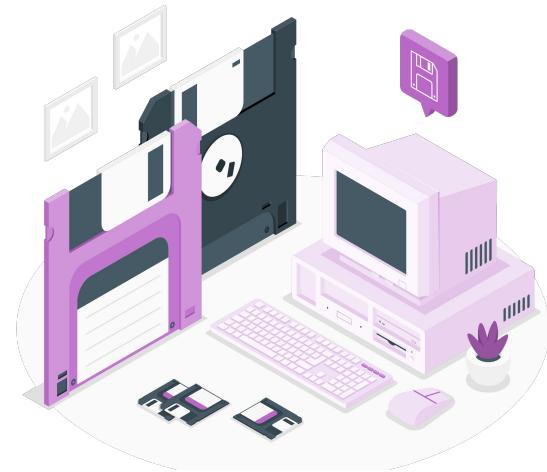
2-) UEFI tarafından desteklenen dosya sistemlerinden biri EFI sistem bölümü (ESP) olarak bilinen özel bir bölüme yerleştirilir.

3-) Önyükleme işlemi artık programları, yapılandırma dosyalarını ve önyükleme sırasında yararlı olabilecek diğer her şeyi içeren uygun bir dosya sistemi kullanabilir.

UEFI, bölümleri, dosya sistemlerini, çalıştırılabilir dosyaları vb. anlayan küçük bir işletim sistemine bezenilebilir

05

Sistem Çağrıları



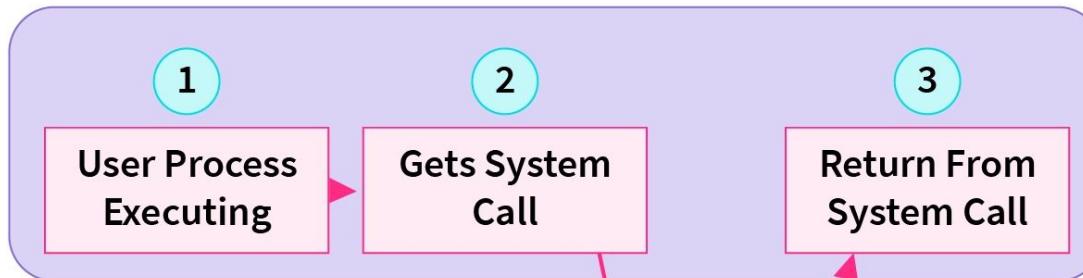
Giriş



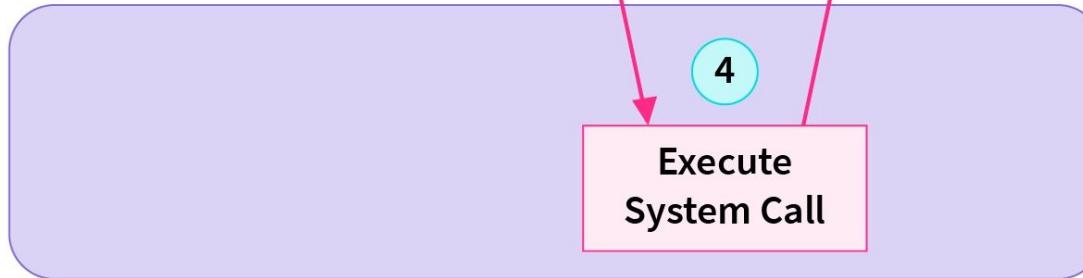
Giriş

WORKING OF A SYSTEM CALL

USER MODE



KERNEL MODE



Giriş

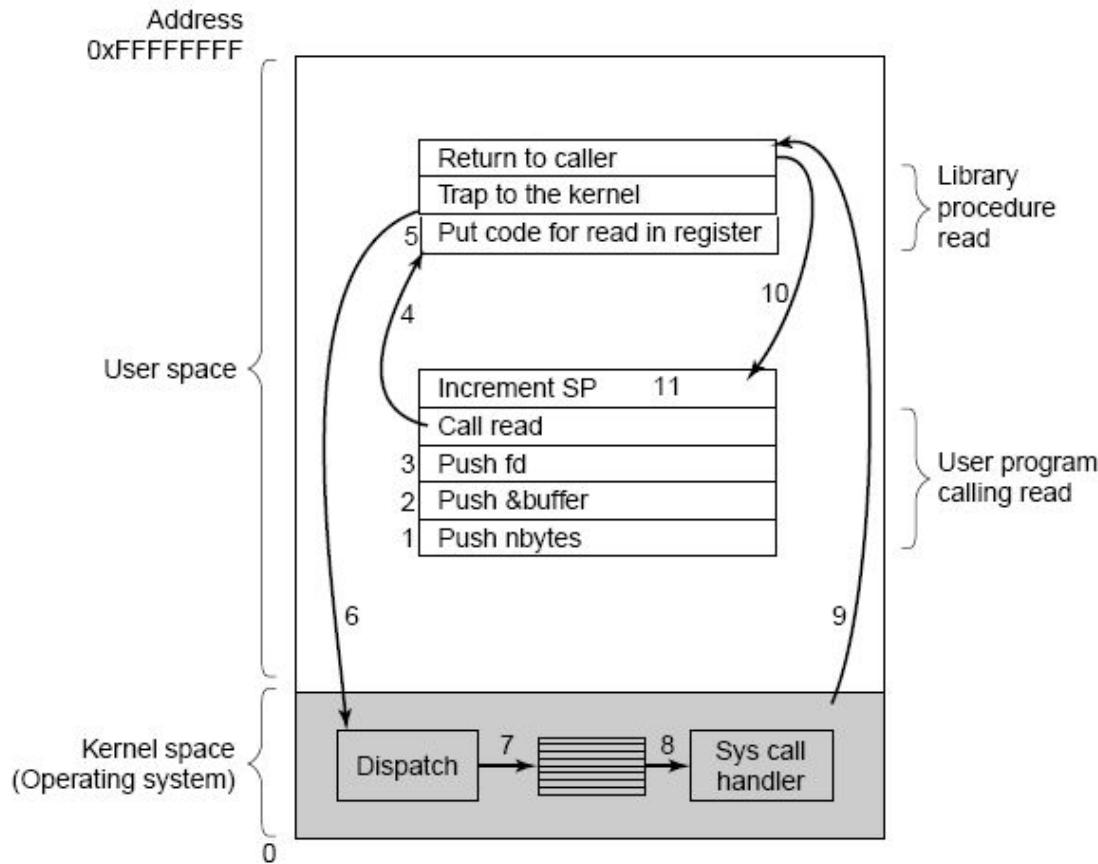
```
count = read(fd, buffer,  
nbytes);
```

count = bayt sayısını

Bu değer nbytes ile aynıdır, ancak okuma sırasında dosya sonu ile karşılaşılırsa daha küçük olabilir.

Sistem çağrısı gerçekleştirilemezse, count < 1

Hata numarası global değişkene (errno) konur.



Giriş

```
count = read(fd, buffer,  
nbytes);
```

1 ve 3-> Parametreler aktarımı

2 -> Tampon adresi aktarımı

4-> Kütüphane prosedürüne yapılan çağrı

5-> Sistem çağrı numarasını saklayıcıya (RAX) konur

6-> Sabit bir adreste yürütmemeyi başlatmak için tuzak komutu çalıştırılır

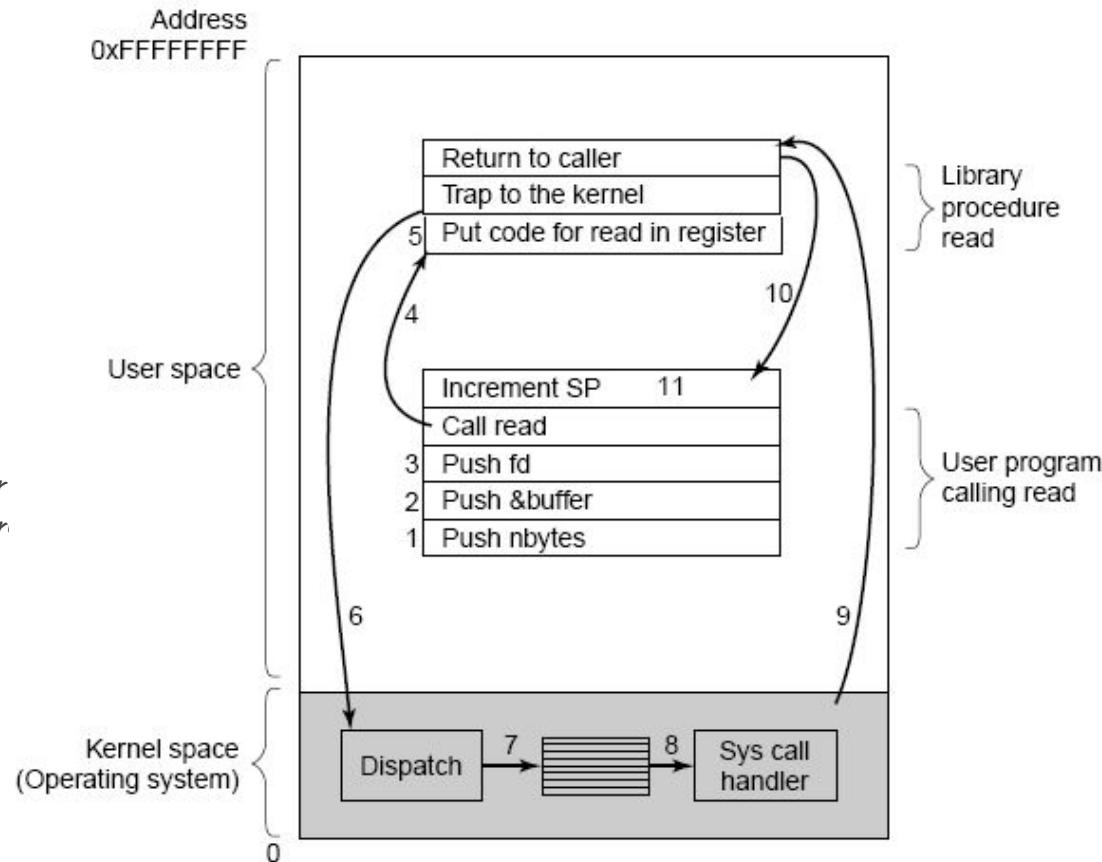
7-> İstek, sistem çağrı numarasına göre sistem çağrı işleyicisine gönderilir

8-> Sistem çağrı işleyicisi çalışır

9-> Kontrol kütüphane prosedürüne geri döndürülür

10-> Kendini çağrıran prosedüre döner

11-> Bir sonraki komutu çalıştırır



Süreç Yönetimi için Sistem Çağrıları

Process management

| Call | Description |
|---------------------------------------|--|
| pid = fork() | Create a child process identical to the parent |
| pid = waitpid(pid, &statloc, options) | Wait for a child to terminate |
| s = execve(name, argv, environp) | Replace a process' core image |
| exit(status) | Terminate process execution and return status |

Fork -> Yeni bir süreç yaratma

Orijinal sürecin tam bir kopyasını oluşturur.

Orijinal süreç ve kopya (ebeveyn ve çocuk) bağımsızdır

Çocuğun belleği ebeveyn ile copy-on-write paylaşılabilir.

Çocukta sıfır, ebeveynde çocuğun PID'sine (Process IDentifier) eşit olan bir değer döndürür.

Süreç Yönetimi için Sistem Çağrıları

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

#define TRUE 1

extern char **environ;

int main(void) {
    int status;

    while (TRUE) {
        type_prompt();
        /* sonsuza dek tekrar */
        /* ekranda istem göster */
        /* terminalden girdi oku */

        pid_t pid = fork();
        if (pid > 0) {
            waitpid(-1, &status, 0);
        } else if (pid == 0) {
            execve(command, parameters, environ);
            perror("execve");
            _exit(127);
        } else {
            perror("fork");
            exit(EXIT_FAILURE);
        }
    }
}
```

Dosya Yönetimi için Sistem Çağrıları

Dosyayı açan çağrı: O_RDONLY, O_WRONLY, O_RDWR

Dosya oluşturan çağrı: O_CREAT

Bir dosyanın herhangi bir bölümüne rastgele erişebilmesi gereklidir.

Konum işaretçisinin değerini değiştiren çağrı: lseek

Sistem çağrıları: Dosya tipini, boyutunu, son değişiklik zamanını tutan çağrı: stat

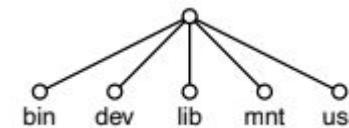


Dizin Yönetimi için Sistem Çağrıları

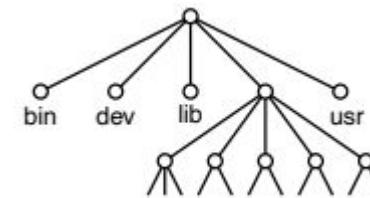
```
link("/usr/jim/memo", "/usr/ast/note");
```

| /usr/ast | | /usr/jim | |
|----------|-------|----------|-------|
| 16 | mail | 31 | bin |
| 81 | games | 70 | memo |
| 40 | test | 59 | f.c. |
| | | 38 | prog1 |

```
mount("/dev/sdb0", "/mnt", 0);
```



| /usr/ast | | /usr/jim | |
|----------|-------|----------|-------|
| 16 | mail | 31 | bin |
| 81 | games | 70 | memo |
| 40 | test | 59 | f.c. |
| 70 | note | 38 | prog1 |



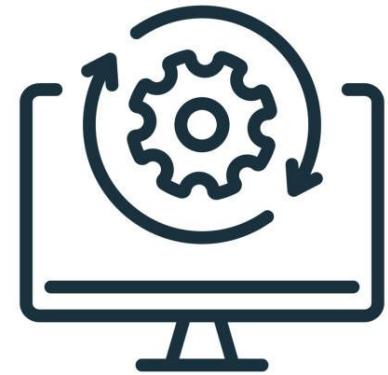
Windows API

- Windows, işletim sistemi hizmetlerini almak için kullanılan bir dizi prosedüre sahiptir (*Application Programming Interface: WinAPI, Win32 API, Win64 API*)
- Binlerce Win32 API (tüm sürümler ile uyumludur), sistem çağrılarının önemli bir kısmı kullanıcı alanında gerçekleştirilir. Dolayısıyla neyin sistem çağrıları neyin sadece kullanıcı uzayı kütüphane çağrıları olduğunu görmek imkansızdır.
- Win32 API, GUI özelliklerini yönetmek için çok sayıda çağrıya sahiptir.
- Windows'ta UNIX'teki gibi bir süreç hiyerarşisi yoktur, bir işlem oluşturulduğundan sonra, oluşturan ve oluşturulan eşittir.

| UNIX | Win32 | Description |
|---------|---------------------|--|
| fork | CreateProcess | Create a new process |
| waitpid | WaitForSingleObject | Can wait for a process to exit |
| execve | (none) | CreateProcess = fork + execve |
| exit | ExitProcess | Terminate execution |
| open | CreateFile | Create a file or open an existing file |
| close | CloseHandle | Close a file |
| read | ReadFile | Read data from a file |
| write | WriteFile | Write data to a file |
| lseek | SetFilePointer | Move the file pointer |
| stat | GetFileAttributesEx | Get various file attributes |
| mkdir | CreateDirectory | Create a new directory |
| rmdir | RemoveDirectory | Remove an empty directory |
| link | (none) | Win32 does not support links |
| unlink | DeleteFile | Destroy an existing file |
| mount | (none) | Win32 does not support mount |
| umount | (none) | Win32 does not support mount, so no umount |
| chdir | SetCurrentDirectory | Change the current working directory |
| chmod | (none) | Win32 does not support security (although NT does) |
| kill | (none) | Win32 does not support signals |
| time | GetLocalTime | Get the current time |

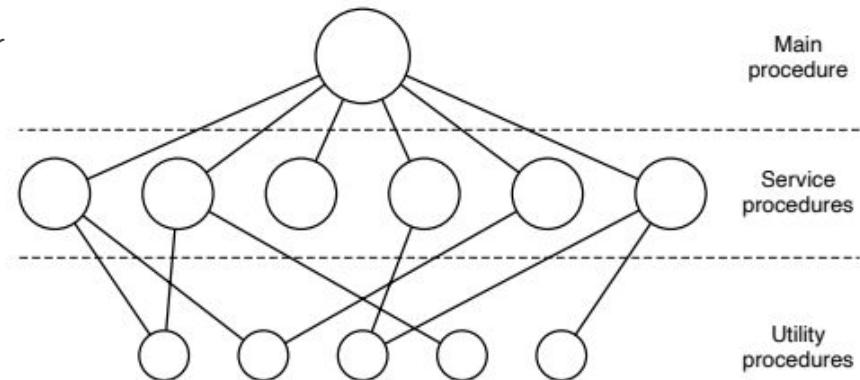
05

İşletim Sistemi Mimarisi



Monolitik Sistemler

1. Tüm işletim sistemini çekirdek modunda tek bir program olarak çalışmaktadır.
2. İşletim sisteminine ait prosedürler oluşturmak için önce tüm bireysel prosedürler derlenir ve ardından sistem bağlayıcıları kullanılarak hepsi tek bir çalıştırılabilir dosyada bir araya getirilir.
3. İşletim sistemi, birbirine bağlanmış bir prosedürler koleksiyonu olarak yazılır.
4. Sistemdeki her yordam, diğerini kolayca çağırabilmektedir.
5. Kısıtlama olmaksızın çağrılabilen binlerce yordama hantal ve anlaşılması zor bir sisteme yol açar. Prosedürlerden herhangi birinde meydana gelecek bir gökme tüm işletim sistemini çökertecektir.
6. Her prosedür diğer tüm prosedürler tarafından görülebilir.
7. Her sistem çağrısi için bir prosedür mevcuttur.
8. Yardımcı prosedürler, hizmet prosedürleri tarafından ihtiyaç duyulan görevleri yürütür.



Katmanlı Sistemler

Her biri bir altakinin üzerine inşa edilen altı katmanlardan oluşmaktadır.

Katman 0 -> İşlemci tahsisi, kesme yönetimi, eş zamanlı programa sürecinin yürütülmesi

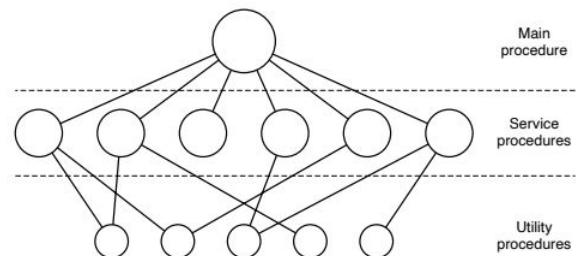
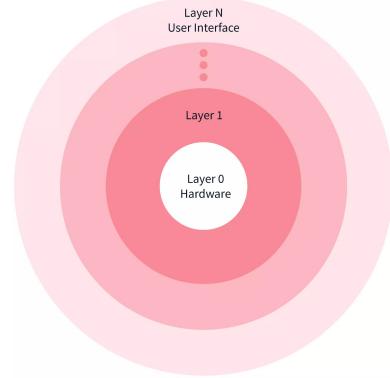
Katman 1 -> Bellek yönetimi

Katman 2 -> Her bir işlem ile kullanıcı arasındaki iletişim

Katman 3 -> G/C cihazlarının yönetilmesi

Katman 4 -> Kullanıcı programlarının yönetimi

Katman 5 -> Kullanıcıların yer aldığı katman



| Layer | Function |
|-------|---|
| 5 | The operator |
| 4 | User programs |
| 3 | Input/output management |
| 2 | Operator-process communication |
| 1 | Memory and drum management |
| 0 | Processor allocation and multiprogramming |

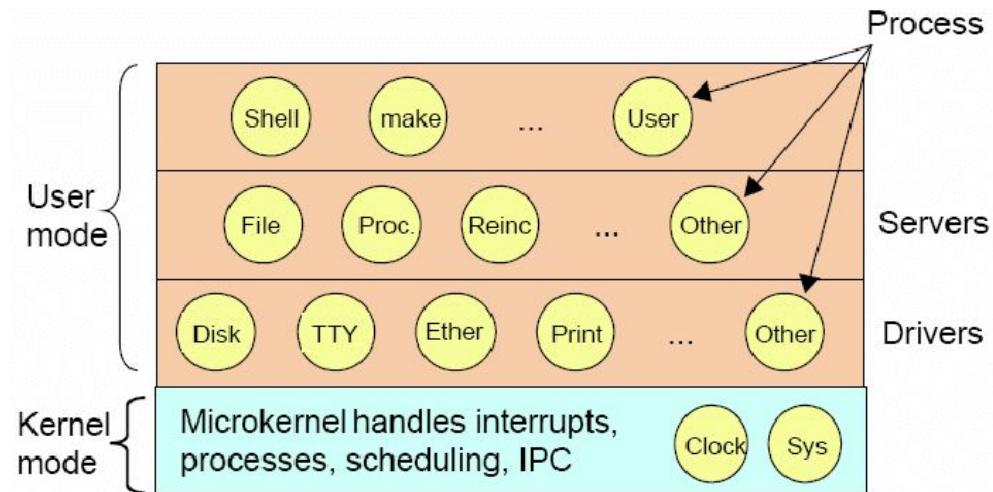
Mikroçekirdekli Sistemler

Katmanlı yaklaşımında, tüm katmanlar çekirdeğe yerleştirilir

Cekirdekteki hatalar sistemi anında gökertebilir.

1000 satır kod başına düşen hata sayısını = 2 ~ 10 hata

Monolitik işletim sistemi 5 milyon satır kod = 10.000 ~ 50.000 çekirdek hatası

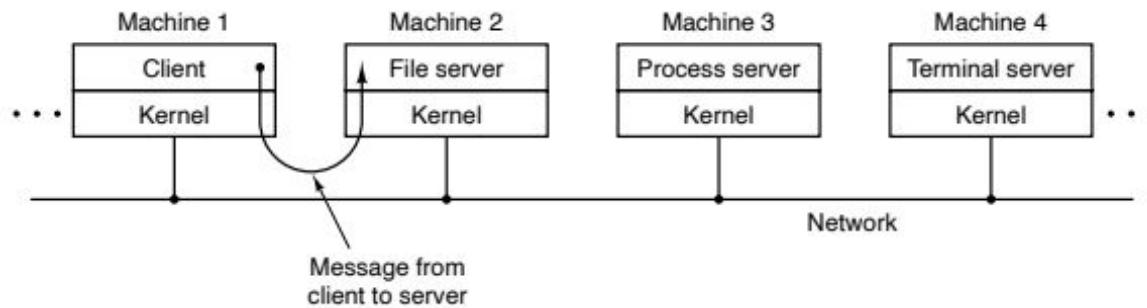


İstemci Sunucu Modeli

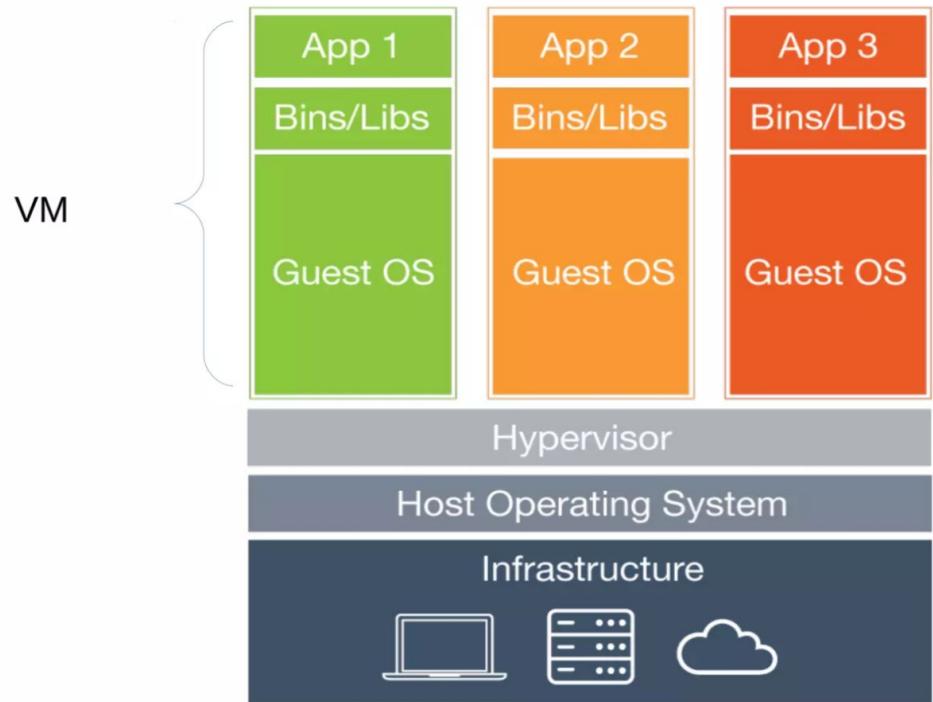
Süreçler: Sunucu ve istemci

İstemciler ve sunucular arasındaki iletişim genellikle mesaj geçişi yoluyla gerçekleşir.

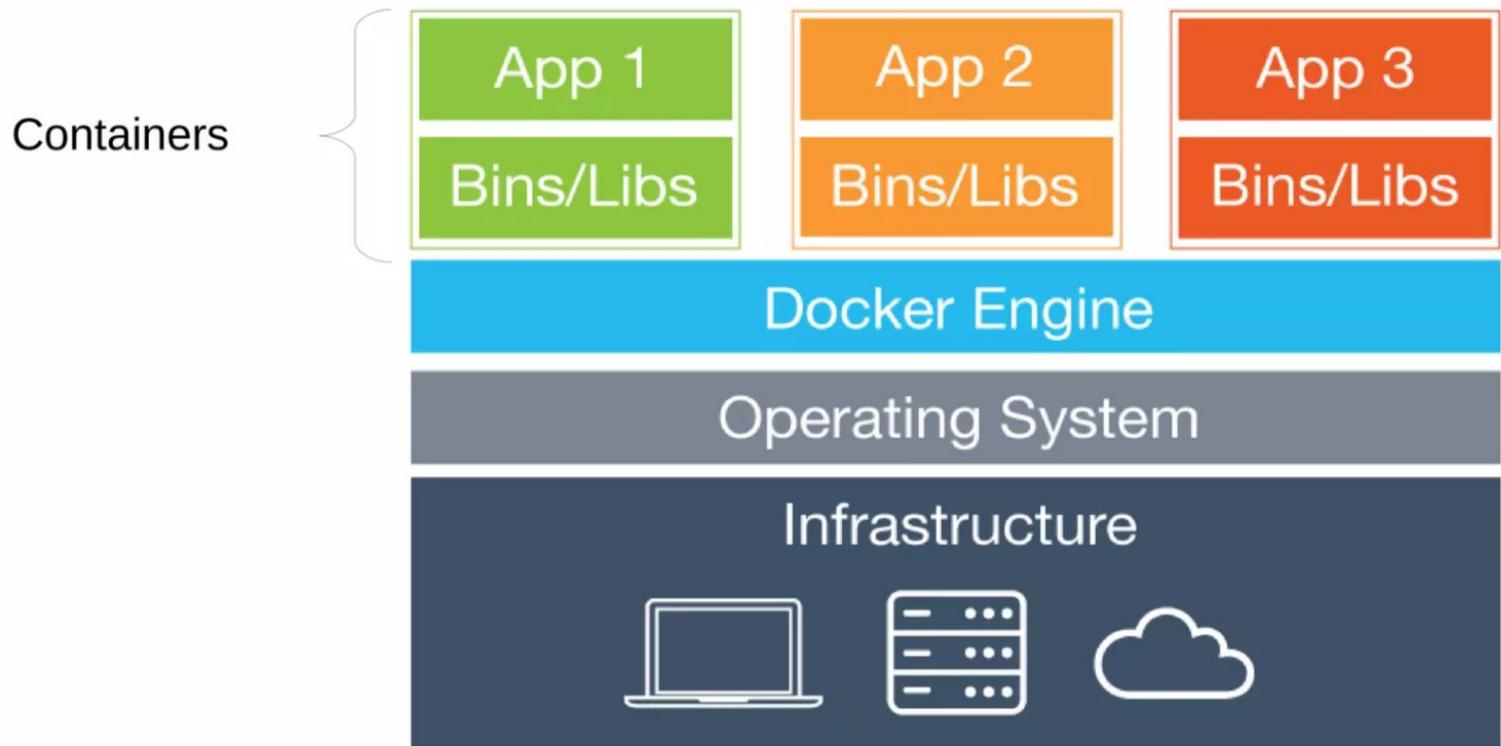
Giderek artan sayıda sisteme (kullanıcılar bilgisayarlarda) istemci olarak, başka yerlerdeki büyük makineler ise sunucu olarak çalışmaktadır



Geleneksel Sanal Makineler



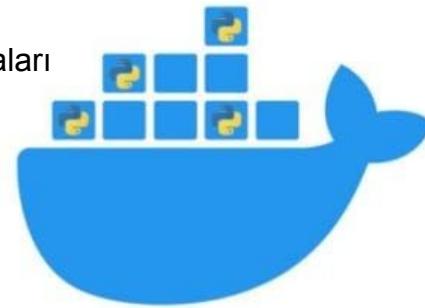
Konteyner(Docker) Sanal Makineler



Konteyner Örnek 1 (Flask + Python)

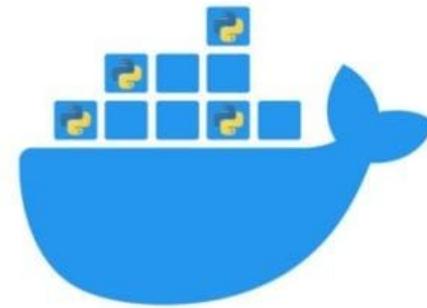
Flask, developer'ların web uygulamalarını hızlı ve basit bir şekilde geliştirmesini sağlamak için tasarlanmış bir **mikro framework**'dır.

Python, esnekliği sayesinde veri biliminden, web uygulamaları oluşturmaya kadar birçok özellik sunan bir programlama dilidir.



Konteyner Örnek 1 (Flask + Python)

```
mkdir -p ~/code/flask-hello && cd  
~/code/flask-hello  
code .
```



Adım 1: Python Uygulamasını Oluşturma



app.py

```
from flask import Flask  
  
app = Flask(__name__)  
  
@app.route("/")  
def hello_world():  
    return 'Hello, World!'  
  
if __name__ == '__main__':  
    app.run(host='0.0.0.0', port=5000)
```

Adım 2: requirements.txt Dosyasını Oluşturma



Flask==1.1.2
Werkzeug==0.16.1
Jinja2==3.0.3
itsdangerous==1.1.0

Adım 3: Dockerfile Oluşturma

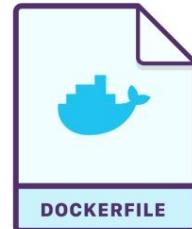
```
# Python 3.9 imajını temel al  
FROM python:3.9-slim
```

```
# Çalışma dizinini ayarla  
WORKDIR /app
```

```
# Gereksinim dosyasını kopyala ve bağımlılıkları yükle  
COPY requirements.txt requirements.txt  
RUN pip install -r requirements.txt
```

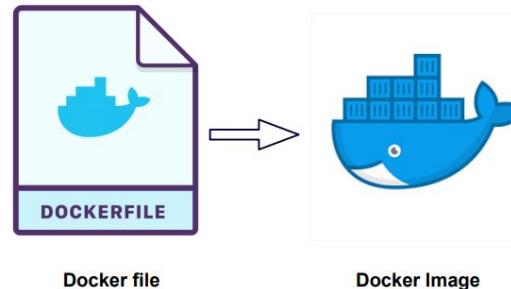
```
# Uygulama dosyalarını kopyala  
COPY ..
```

```
# Uygulamayı 5000 portunda başlat  
CMD ["python", "app.py"]
```



Docker file

Adım 4: Docker Image Oluşturma



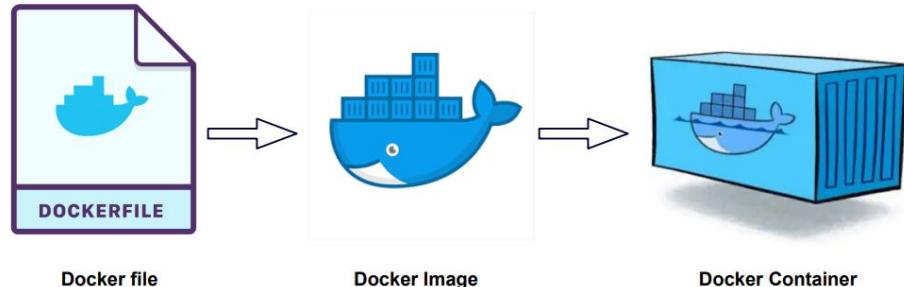
Dockerfile

```
docker build -t hello-world-app .
```

Adım 5: Docker Container'ı Çalıştırma



```
docker run -p 5000:5000  
hello-world-app
```

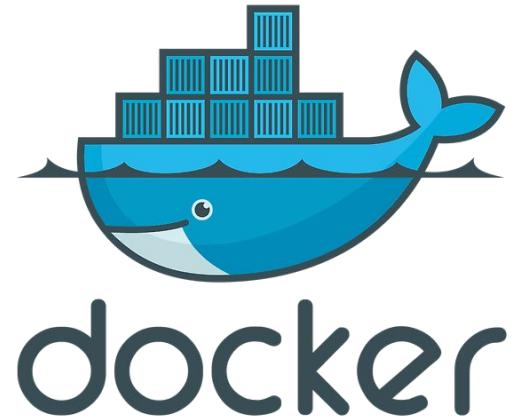
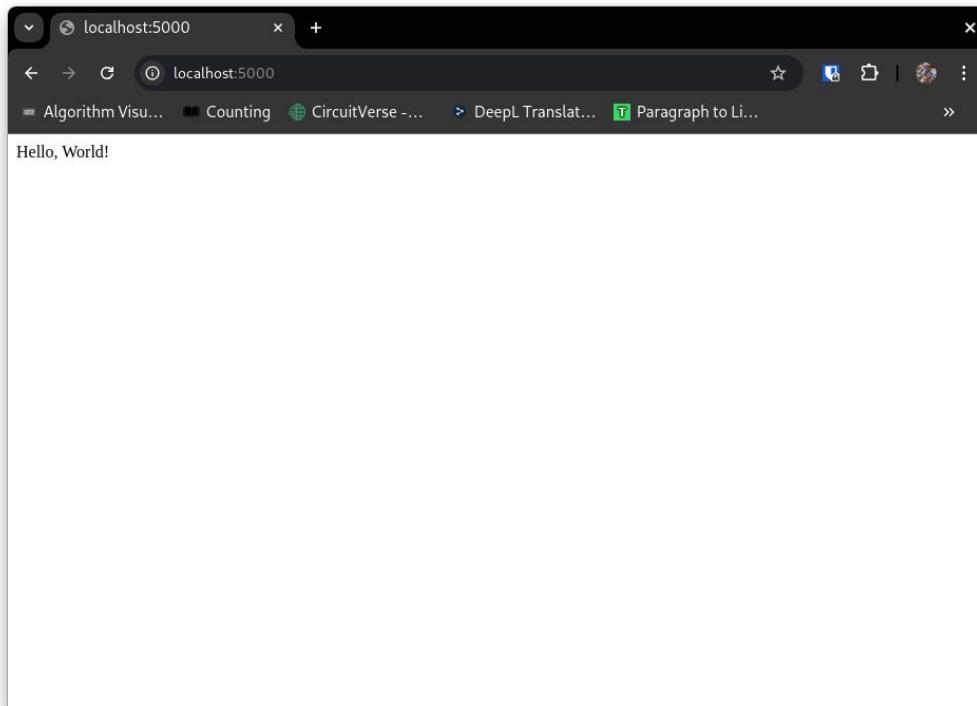


Docker file

Docker Image

Docker Container

Adım 6: Uygulamayı Test Etme



Konteyner Örnek 2 (Node.js)

[Node.js](#), V8 JavaScript motoru üzerinde çalışan, sunucu tarafında JavaScript çalışma yeteneği sağlayan açık kaynaklı bir çalışma ortamıdır.



Konteyner Örnek 2 (Node.js)

```
mkdir -p ~/code/node-hello && cd  
~/code/node-hello  
code .
```



Adım 1: Node.js Uygulamasını Oluşturma



app.js

```
const http = require('http');

// Sunucu oluşturuluyor
const hostname = '0.0.0.0';
const port = 3000;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello, World from Dockerized Node.js Application!\n');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

Adım 2: package.json Dosyasını Oluşturma



```
{  
  "name": "node-docker-app",  
  "version": "1.0.0",  
  "description": "Simple Node.js app running inside Docker",  
  "main": "app.js",  
  "scripts": {  
    "start": "node app.js"  
  },  
  "dependencies": {}  
}
```

Adım 3: Dockerfile Oluşturma

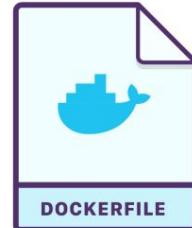
```
# Node.js'in resmi LTS Debian tabanlı imajını kullanıyoruz.  
FROM node:14
```

```
# Uygulama dosyalarını konteynerin içine kopyalıyoruz  
WORKDIR /usr/src/app  
COPY package.json .  
COPY app.js .
```

```
# Eğer bağımlılıklar varsa, burada yüklenir (npm install)  
RUN npm install
```

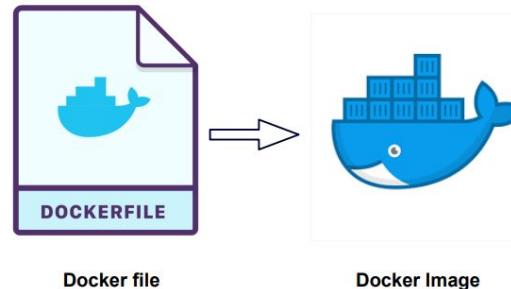
```
# Uygulamanın 3000 portunda dinlemesini sağlıyoruz  
EXPOSE 3000
```

```
# Uygulamayı başlatıyoruz  
CMD ["npm", "start"]
```



Docker file

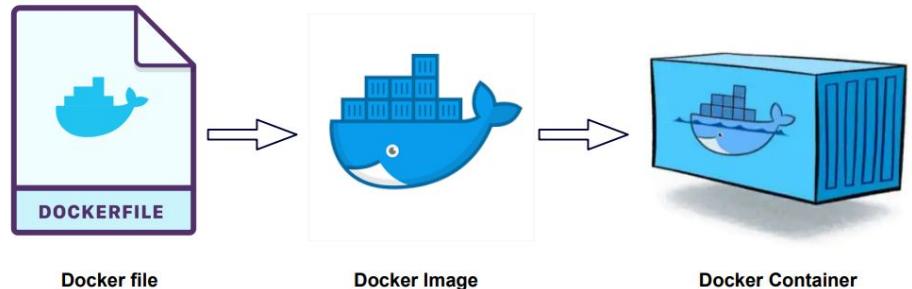
Adım 4: Docker Image Oluşturma



Dockerfile

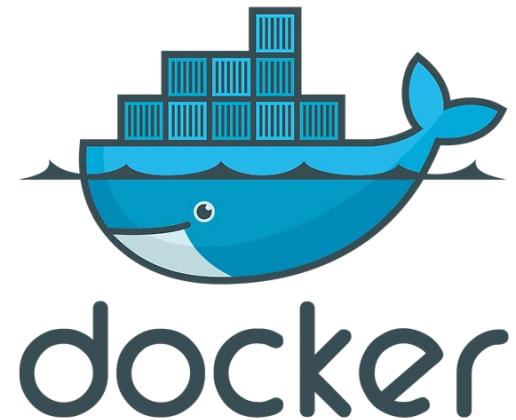
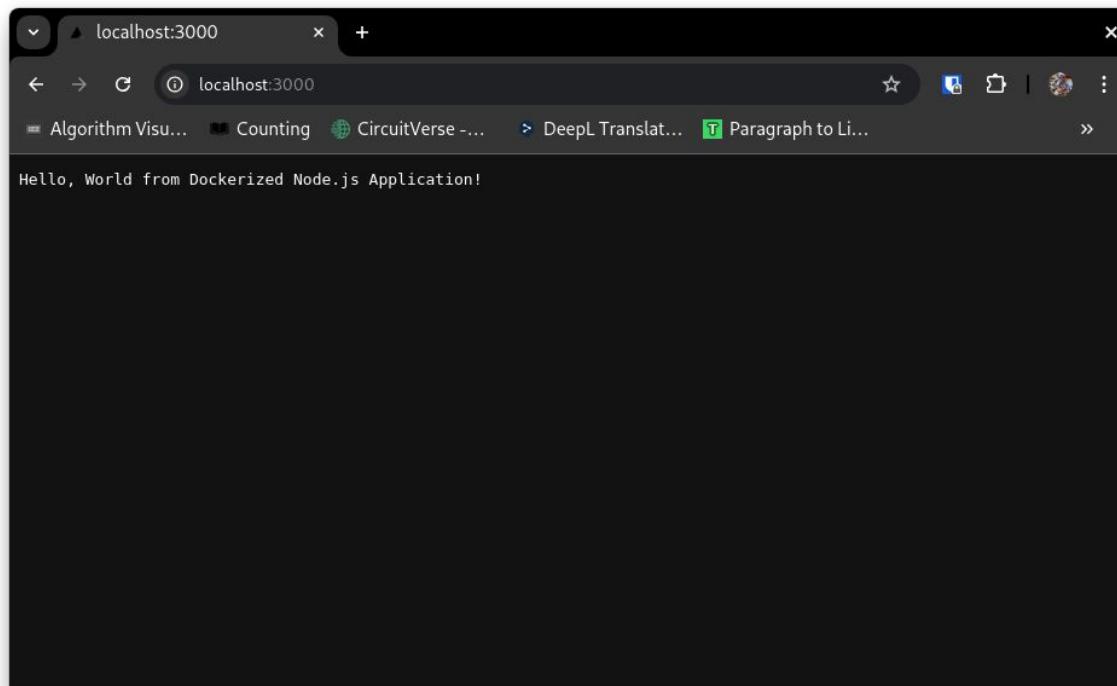
```
docker build -t my-node-app .
```

Adım 5: Docker Container'ı Çalıştırma



```
docker run -p 3000:3000  
my-node-app
```

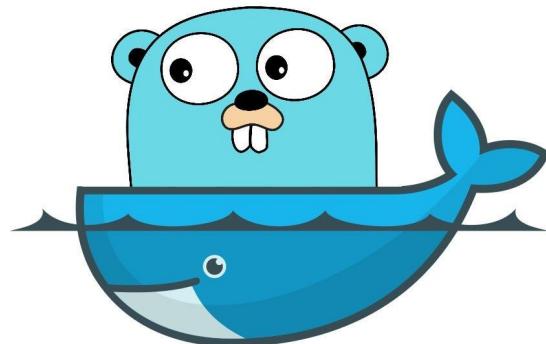
Adım 6: Uygulamayı Test Etme



Konteyner Örnek 3 (Go & scratch)

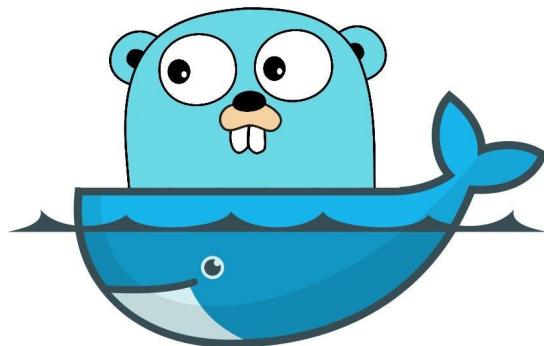
[Go](#), Google tarafından geliştirilen açık kaynaklı bir programlama dilidir

[Scratch](#), Docker sistemine sıfır imajla başlamak



Konteyner Örnek 3 (Go & scratch)

```
mkdir -p ~/code/go-hello && cd  
~/code/go-hello  
code .
```



Dizin yapısı

```
go-hello/          ← Projenin kök klasörü (istediğin isim olabilir)
  └── main.go      ← Go uygulamasının giriş noktası (main package + main())
  └── fonksiyonu
  └── go.mod       ← Go modül tanımı (modül adı + Go sürümü + bağımlılıklar)
  └── .dockerignore ← Docker build sırasında bağlama (context) DAHİL
EDİLMEYECEK dosyalar
  └── Dockerfile    ← İmajın nasıl inşa edileceğini tarif eden dosya (multi-stage
                      build)
```

Adım 1: main.go Dosyasını Oluşturma

```
package main
```

Çalıştırılabilir bir program için giriş paketi olan `main`'i kullandığını belirtir.
Go'da `package main` ve `func main()` birlikte olunca çalıştırılabilir program olur.

Adım 1: main.go Dosyasını Oluşturma

```
package main

import (
    "fmt"
    "log"
    "net/http"
    "os"
)
```

Kullanılacak standart kütüphaneler

- `fmt`: metin biçimleme/yazma
- `log`: log yazma
- `net/http`: HTTP sunucusu ve istemcisi.
- `os`: ortam değişkenleri, dosya sistemi gibi işletim sistemi işlevleri.

Adım 1: main.go Dosyasını Oluşturma

```
package main

import (
    "fmt"
    "log"
    "net/http"
    "os"
)
func main() {
```

Programın başlangıç noktası.

Adım 1: main.go Dosyasını Oluşturma

```
package main

import (
    "fmt"
    "log"
    "net/http"
    "os"
)

func main() {
    msg := os.Getenv("MESSAGE")
```

Ortam değişkenlerinden MESSAGE adlı değeri okur. Bulamazsa boş string döner.

Adım 1: main.go Dosyasını Oluşturma

```
package main

import (
    "fmt"
    "log"
    "net/http"
    "os"
)

func main() {
    msg := os.Getenv("MESSAGE")
    if msg == "" {
        msg = "Hello from minimal Go in Docker!"
    }
}
```

Eğer MESSAGE tanımlı değilse (veya boşsa) msg için bir **varsayılan** metin belirler.

Adım 1: main.go Dosyasını Oluşturma

```
package main

import (
    "fmt"
    "log"
    "net/http"
    "os"
)

func main() {
    msg := os.Getenv("MESSAGE")
    if msg == "" {
        msg = "Hello from minimal Go in Docker!"
    }

    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintln(w, msg)
    })
}
```

/ (kök) yolu için bir HTTP handler tanımlar.

Bu URL'e istek gelince `msg` içeriğini **HTTP cevabına** yazar.

`w` yazma (response), `r` okuma (request) nesneleri

Adım 1: main.go Dosyasını Oluşturma

```
package main

import (
    "fmt"
    "log"
    "net/http"
    "os"
)

func main() {
    msg := os.Getenv("MESSAGE")
    if msg == "" {
        msg = "Hello from minimal Go in Docker!"
    }

    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintln(w, msg)
    })

    http.HandleFunc("/healthz", func(w http.ResponseWriter, r *http.Request) {
        w.WriteHeader(http.StatusOK)
        fmt.Fprintln(w, "ok")
    })
}
```

/healthz yolу için ikinci bir handler.

Önce HTTP durum kodunu **200 OK** yazar, sonra gövdeye **ok** basar.

Sağlık kontrolü (liveness/readiness) gibi otomasyonlar için basit endpoint.

Adım 1: main.go Dosyasını Oluşturma

```
package main

import (
    "fmt"
    "log"
    "net/http"
    "os"
)

func main() {
    msg := os.Getenv("MESSAGE")
    if msg == "" {
        msg = "Hello from minimal Go in Docker!"
    }

    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintln(w, msg)
    })

    http.HandleFunc("/healthz", func(w http.ResponseWriter, r *http.Request) {
        w.WriteHeader(http.StatusOK)
        fmt.Fprintln(w, "ok")
    })
}

addr := ":8080"
log.Printf("listening on %s", addr)
log.Fatal(http.ListenAndServe(addr, nil))
}
```

Sunucunun dinleyeceği adres/port. " :8080 "

Adım 2: go.mod dosyasını Oluşturma

go.mod

- `module example.com/go-hello` gibi bir satırla **modül adını** tanımlar.
- `go 1.22` satırı derleme hedef sürümünü belirtir.

Adım 3: dockerignore dosyasını Oluşturma

.dockerignore

- Docker “build context”e gereksiz dosyaların gitmesini engeller; build’i **hızlandırır** ve imaj katmanlarını tertipli tutar.
- Örnek içerik:

```
.git  
.vscode  
*.log
```

Adım 4: Dockerfile dosyasını Oluşturma

```
# --- Builder stage ---
FROM golang:1.22-alpine AS builder
ENV CGO_ENABLED=0 GOOS=linux GOARCH=amd64
// Go derleyicisi
// Arch Linux x86_64

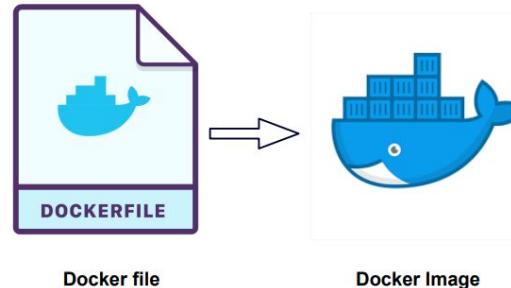
# Uygulama dosyalarını konteynerin içine kopyalıyoruz
WORKDIR /src
COPY go.mod ./

# Eğer bağımlılıklar varsa, burada yüklenir (npm install)
RUN go mod download
COPY . .
RUN go build -trimpath -ldflags="-s -w" -o /out/app

# --- Runtime stage ---
FROM scratch
EXPOSE 8080
ENV MESSAGE="Hello from minimal Go in Docker!"
// Varsayılan imaj
// 8080 portunda dinleniyor
// Varsayılan mesaj

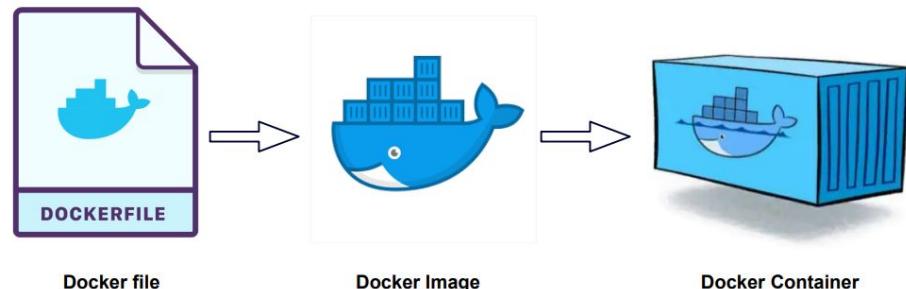
# Uygulamayı başlatıyoruz
COPY --from=builder /out/app /app
ENTRYPOINT ["/app"]
```

Adım 5: Docker Image Oluşturma



```
docker build -t go-hello .
```

Adım 6: Docker Container'ı Çalıştırma



Dockerfile

```
docker run -p 8080:8080 go-hello
```

Adım 7: Uygulamayı Test Etme

