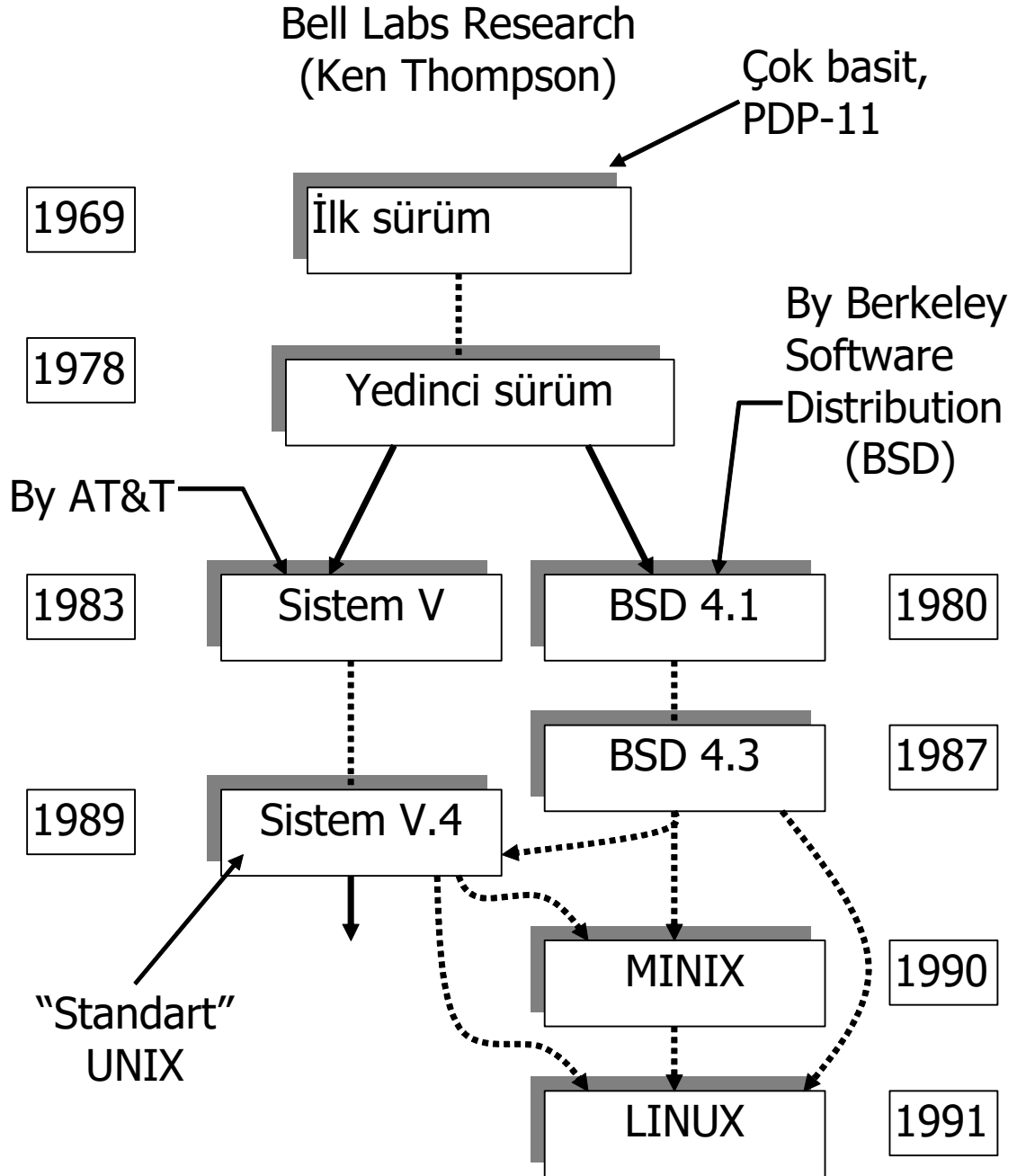
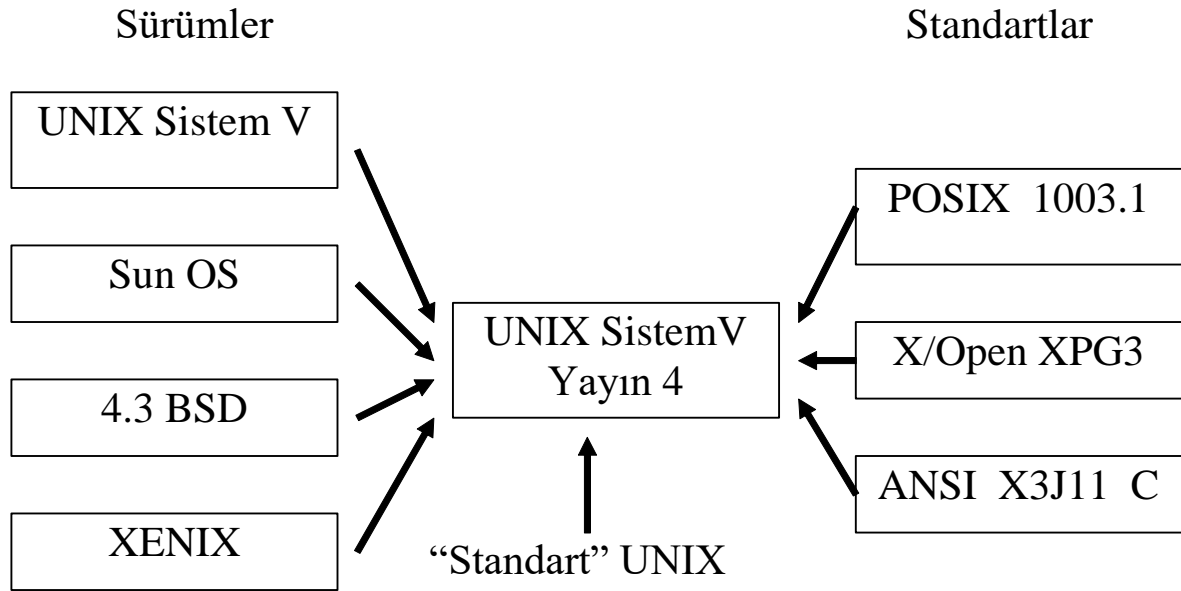


## UNIX tarihinde genel adımlar



## UNIX tarihi ve mimarisine genel bakış.

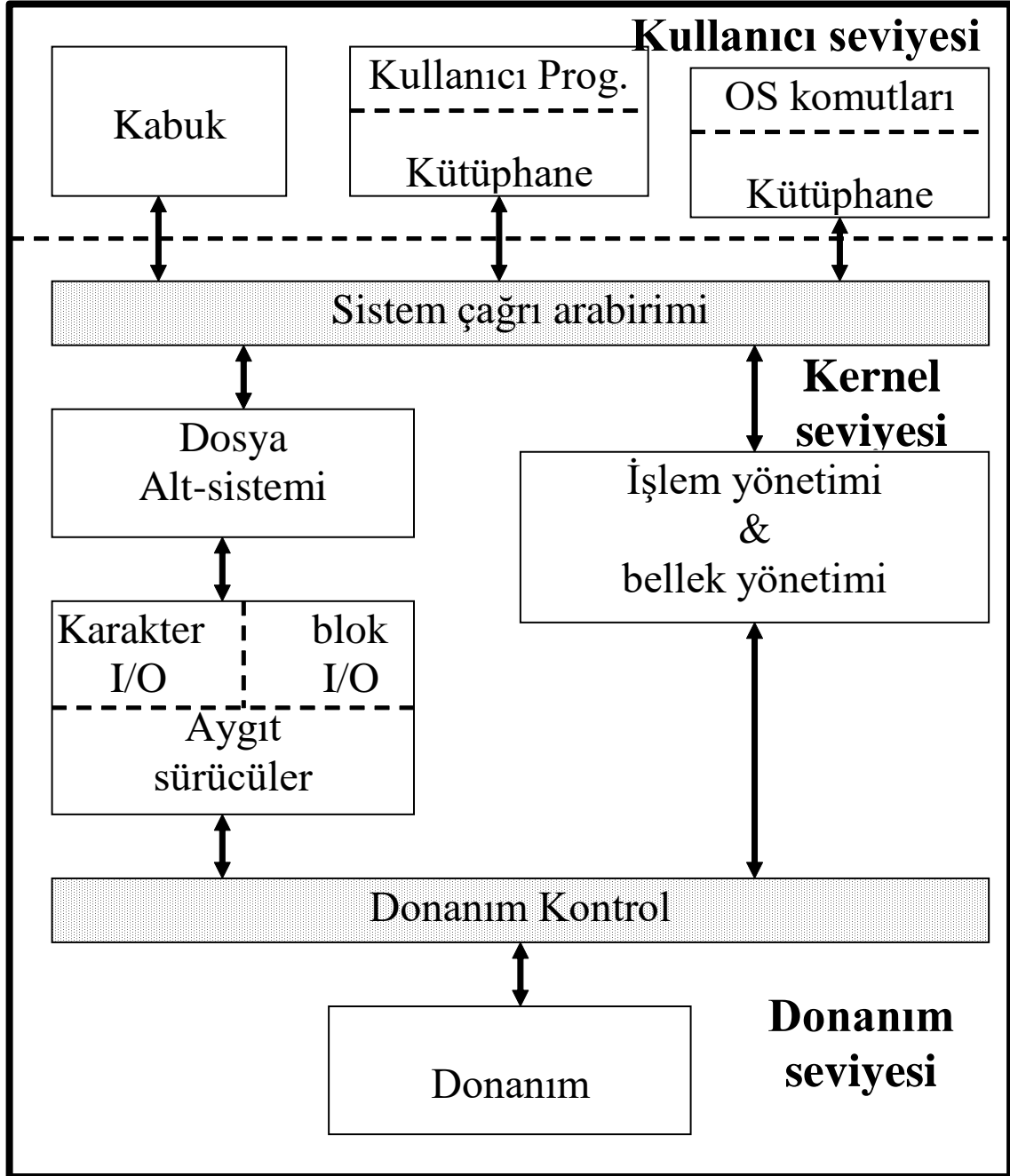
### UNIX sürümleri ve standartlarının birleştirilmesi



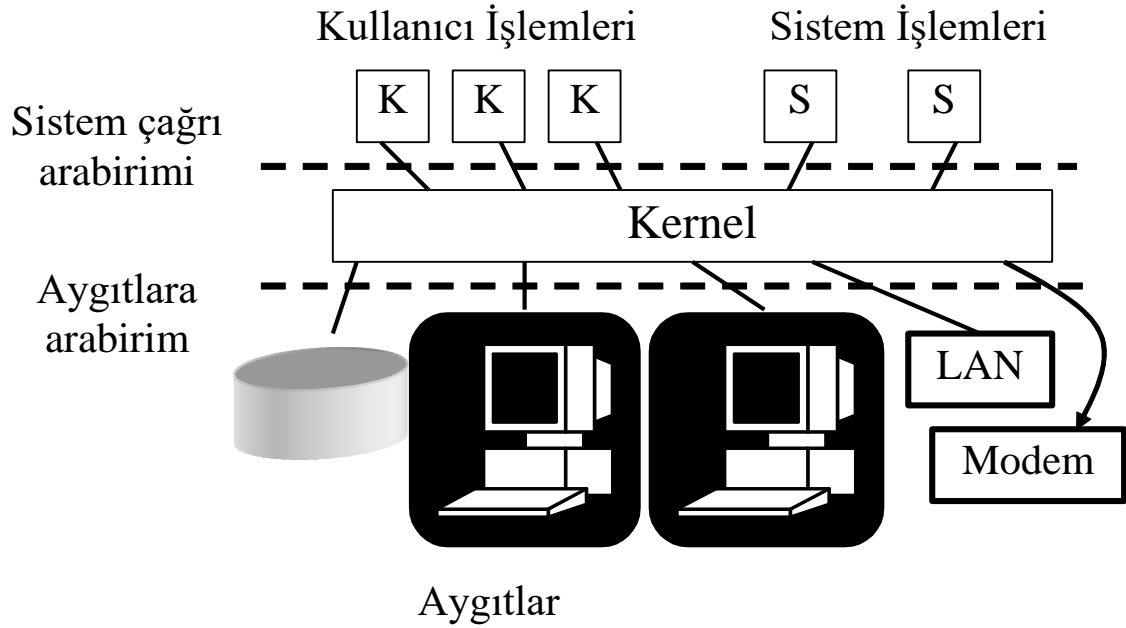
### UNIX Sistem V.4 ' ün bazı özellikleri

System V Yayın 3	<ul style="list-style-type: none"><li>• Uzak Dosya Paylaşımı (RFS)</li><li>□ Taşıma Katmanı Arabirimi (TLI)</li><li>□ STREAMS iletişim olanakları</li><li>□ İşlemler Arası İletişim (IPC)</li></ul>
4.3 BSD	<ul style="list-style-type: none"><li>• <u>TCP/IP Protokolleri</u></li><li>□ <u>Soketler</u></li><li>• Hızlı Dosya Sistemi</li></ul>
SUN OS	<ul style="list-style-type: none"><li>• Bağlı Dosya Sistemleri</li><li>□ <u>Uzaktan Yordam Çağrısı(RPC)</u></li><li>• Bellek Tabanlı Dosyalar (Ortak bellek alanları, sanal bellek, ...)</li></ul>
XENIX	<ul style="list-style-type: none"><li>• 80386 İkili uygunluk</li></ul>
Yeni Özellikler	<ul style="list-style-type: none"><li>• Sanal Dosya Sistemi</li><li>• Gerçek Zaman</li><li>• STREAMS geliştirmeleri</li></ul>

## UNIX Ana Modülleri



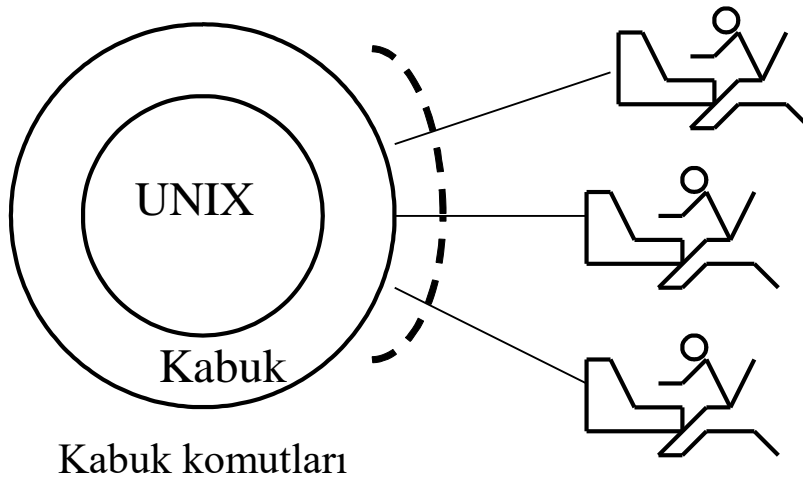
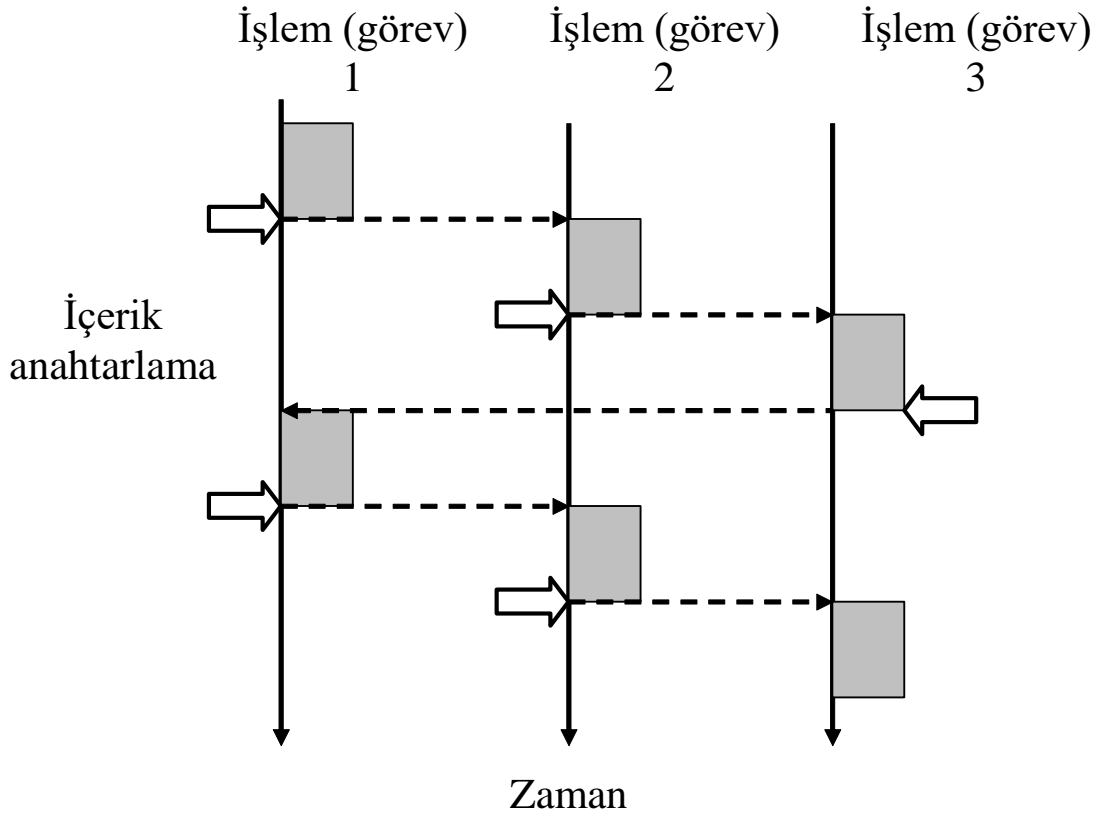
## UNIX ve ana servisleri



## Ana Servisler

- ❑ **Sistem ilklendirilmesi (sıfırlanması)**
- ❑ **İşlem yönetimi**
- ❑ **Bellek yönetimi**
- ❑ **Dosya sistemi yönetimi**
- ❑ **Giriş / Çıkış (input / output) yönetimi**
- ❑ **İletişim hizmetleri**
- ❑ **Program arabirimleri (system calls,..)**

UNIX, Sanal Belleği, Programlanabilen Kabukları ve Ağ yapılandırması ile, Çok-görevli, Çok-kullanıcılı bir işletim sistemidir(OS).



## **Bir kaç UNIX komutları**

ls or ls -l	mevcut dizin içeriğini listele
cd <yol>	bir dizin değiştir
rm <dosya>	bir dosya sil
mkdir <isim>	bir dizin oluştur
rmdir <isim>	bir dizin sil
more <dosya>	bir metin dosyası içeriğini göster

## **Tipik bir UNIX oturumu**

Login: kullanıcı\_adınız

Password: şifreniz

%komut 1

%komut 2

% ...

%logout

## UNIX yardım manueli

<b>Bölüm</b>	<b>İçerik</b>
1	Kullanıcı kom. (Kabuk komutları)
2	OS servisleri (sistem çağrıları)
3	Kütüphane fonksiyonları
4	Aygıtlar, ağlar, arabirimler
5	Sistem dosya formatları
6	Demo programları
7	Çeşitli (ASCII, v.b. )
8	Sistem bakım komutları

Her bölümün bir giriş (intro) açıklaması var

%man 2 intro

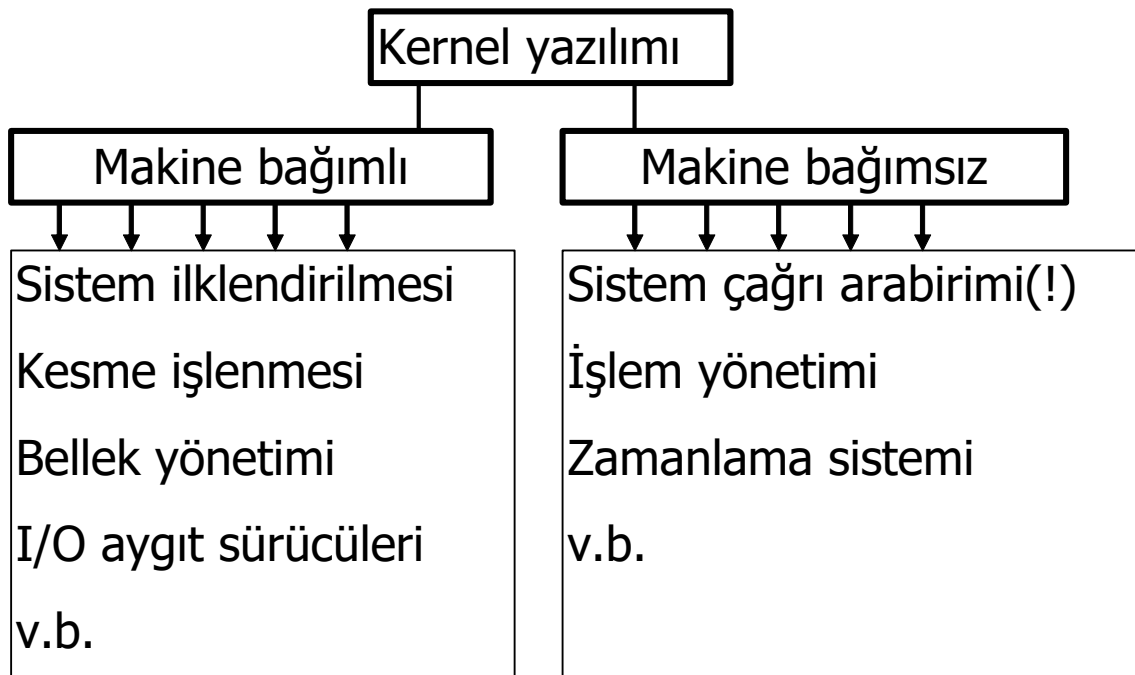
%man 2 fork

%man 3 sin

## UNIX kernel

### Ana servisler:

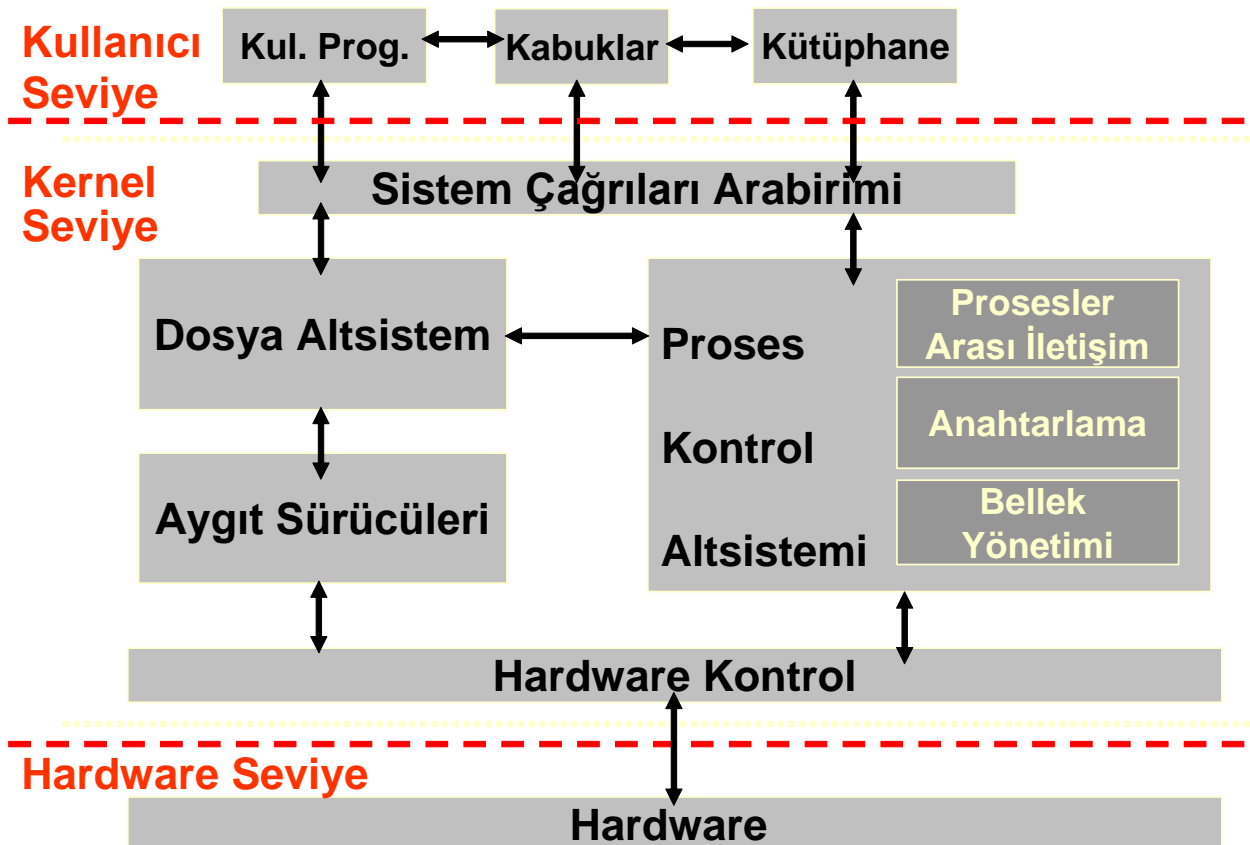
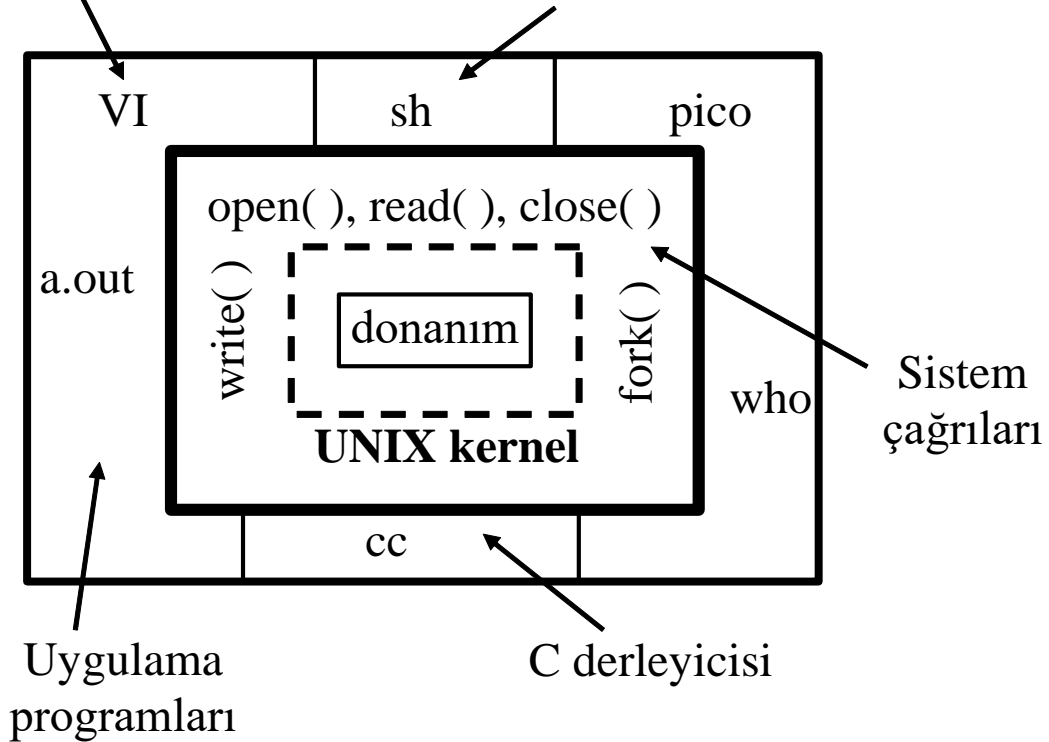
- Sistem ilklendirilmesi ("bootstrap").
- İşlem yönetimi (işlem yaratma, kontrol ve sonlandırılması).
- Bellek yönetimi (sanal bellek te dahil).
- Dosya sistemi yönetimi.
- I/O(giriş/çıkış) yönetimi.
- İletişim hizmetleri (İşlemler arası iletişim, ağlar,..).
- Program arabirimleri (sistem çağrıları).



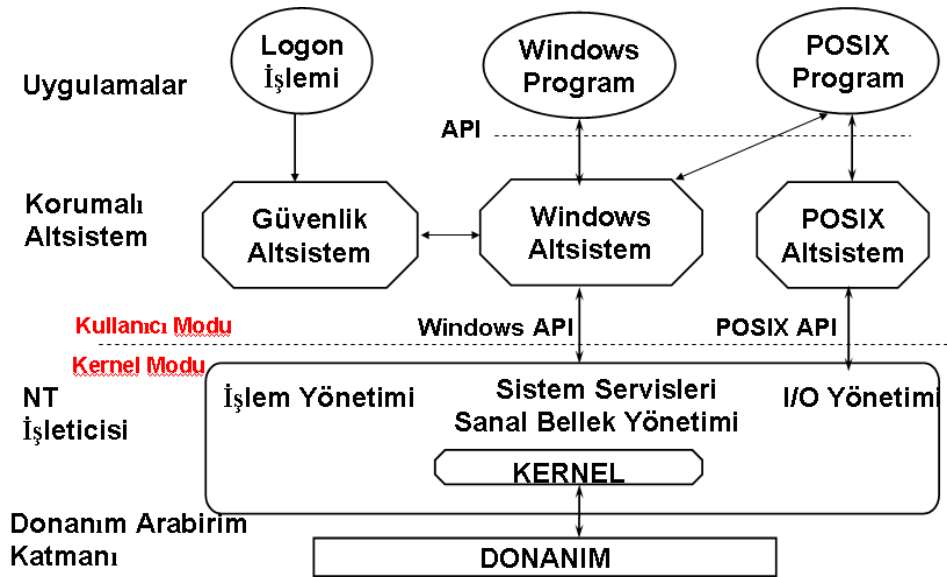


## UNIX mimarisine soyut bir bakış

Metin düzenleyici Komut-satırı yorumlayıcısı (kabuk)



## WINDOWS mimarisine bir bakış : Mikrokernel Mimarisi



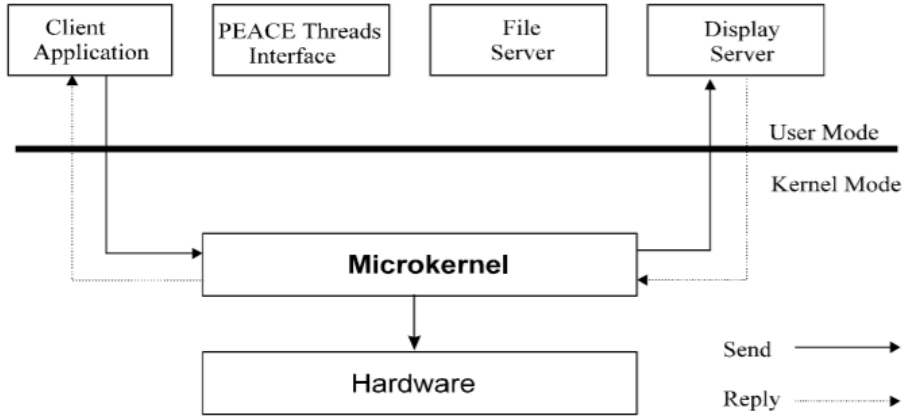
- OS birçok bağımsız proseslere (sunucu) ayrılır, her bir sunucu esasen bir çok işlevden oluşan tek bir servis sağlayıcı olarak görülebilir. Örneğin, IO sunucu, Bellek Sunucu, gibi.

- Mikrokernel mimarisinde, OS sunuculara ait temel servisler kernelde barındırılır, geri kalan kullanıcı katmanına (Altsistem katmanına) taşınır. İhtiyaç halinde, kernel (istemci olarak), altsistemden (sunucu) gerekli hizmeti temin eder.
- Bir sunucuya ait tüm işlevler, kullanıcı katmanına devrededilebilir. Bu durumda, kernel, sadece ilgili servisi kullanmak için gerekli temel işlevleri içerir.
- Hatta kernelin derinliklerinde yer alan, Sanal Bellek, IPC, sayfalama, gibi servislerde kernel' den üst katmanına taşınabilir. Örnekleri - Mach, Amoeba, Plan 9, Windows NT, Chorus

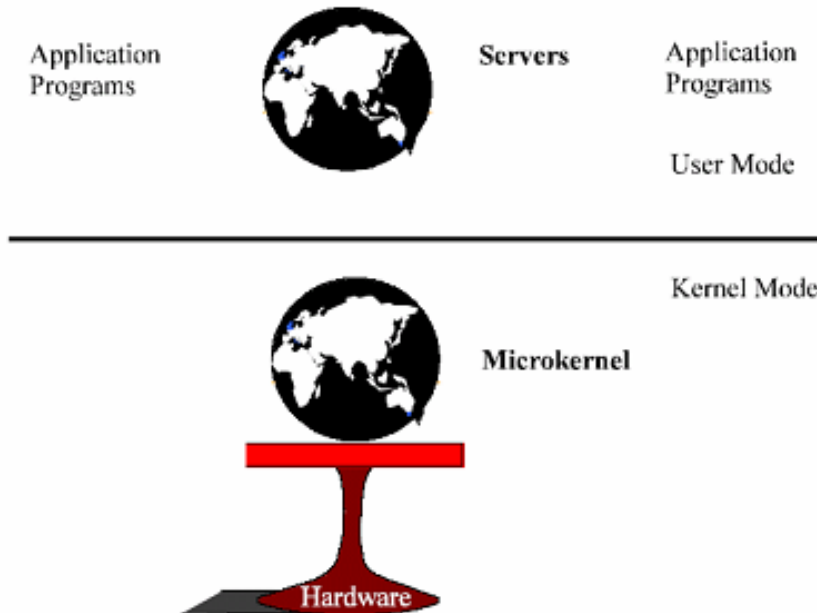
- Sunucular birbiriyle Prosesler Arası İletişim (IPC) üzerinden mesaj gönderme yoluyla haberleşirler. Bu mimari esasen İstemci-Sunucu mimarisidir; prosesler işletim sistemi servislerini IPC üzerinden sunucu proseslere istek göndererek çağırabilirler.

- Genel olarak, mikrokernel' de tutulan bazı servisler: Short-term scheduling (Kısa-sürelili Proses Anahtarlama), Low-level memory management (Alt-seviye bellek yönetimi), Inter-process communication (Mesajlaşma yoluyla IPC), Low level Input/Output (Alt-seviye Giriş/Çıkış), Low-level network support (Alt-seviye ağ desteği)

- Herbir sunucu(altsistem) kullanıcı modunda çalışır ve istemcilere (diğer proses yada sunucular) hizmet verir. İstemciler, mesaj yoluyla hizmet talep eden başka bir sunucu proses, kernel proses yada kullanıcı pogramı olabilir. Orneğın, şekle referansla , kullanıcı prog. kernelden görüntü işlevleriyle ilgili hizmet talep eder fakat kernel bu işi kullanıcı katmanında Display sunucuya yaptırır ve sonucu kullanıcı prog. döndürür.

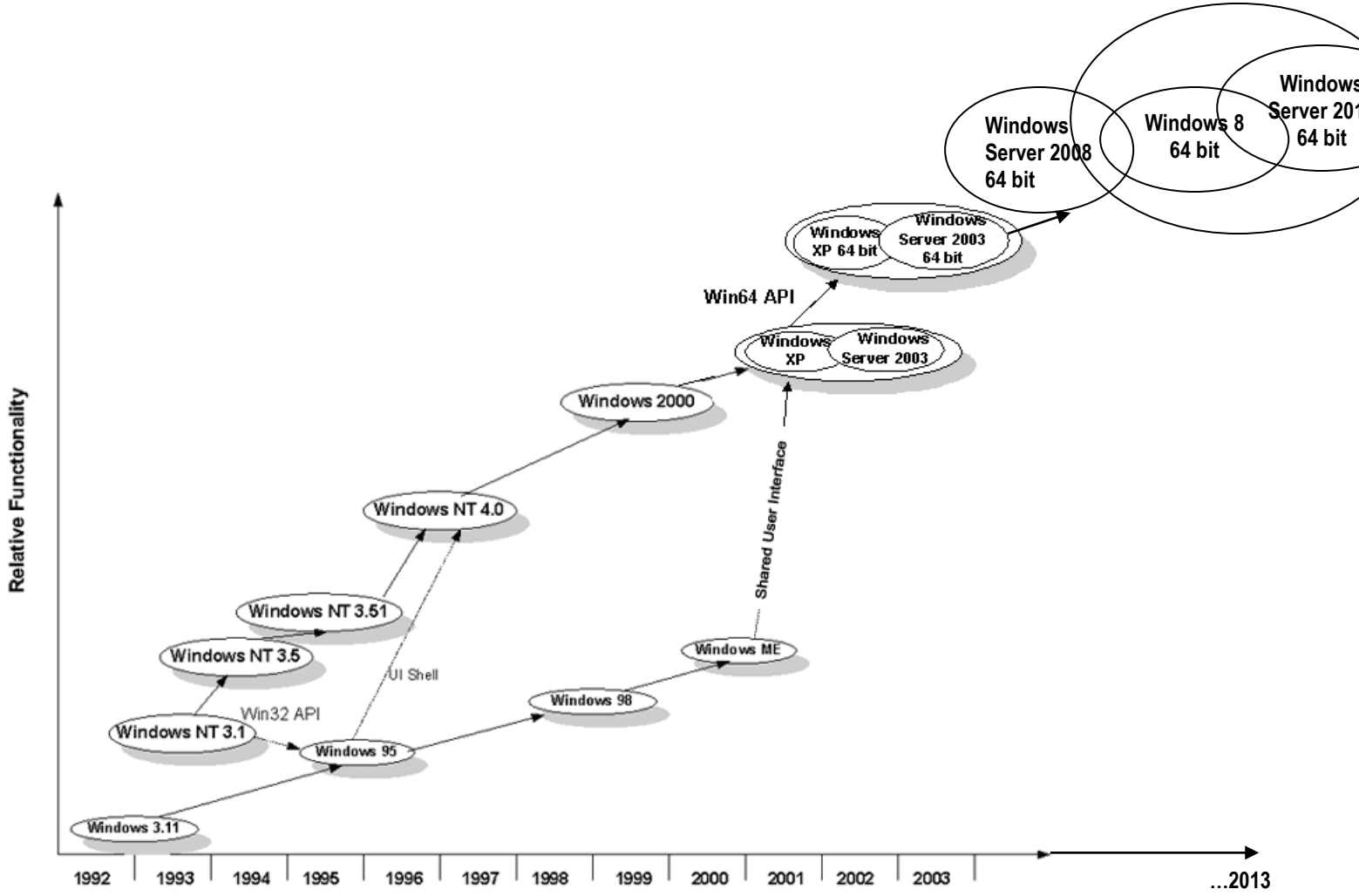


## OS tasarımında popüler yaklaşım: Mikrokernel



OS = Mikrokernel + Kullanıcı Altsistemi

Mikrokernel internet üzerinden Kullanıcı Altsisteminin sunduğu hizmete erişebilir.



## UNIX VE LINUX

UNIX **telif hakkı bulunan** bir üründür. Sadece bazı büyük firmalar UNIX ticari markasını ve ismini kullanabilir. IBM AIX, Sun Solaris ve HP-UX UNIX işletim sistemleridir. **The Open Group** adlı kuruluş UNIX ilişkili tüm **ticari lisanslama** programlarını yönetir. Pek çok UNIX sistemi doğal olarak **ticari amaçlıdır**. Popüler ve yaygın birkaç UNIX işletim sistemleri HP-UX, IBM AIX, Sun Solairs, Mac OS X, IRIX

**POSIX** (UNIX için Taşınabilir İşletim Sistemi Arabirimi) standartlarına göre **Linux**; UNIX olarak görülebilir. POSIX standardını kullanan programlar bir sistemden diğerine kolayca taşınabilir. POSIX, İşletim Sistemi ve Programlar arasında, IEEE tarafından sağlanan standart bir arayüzdür (Bir grup sistem çağrıları veya API'ler gibi düşünülebilir). Linux'un resmi kernel README sayfasında şöyle yazar:

*"Linux; **Linus Torvalds** tarafından internet üzerindeki bir grup hacker yardımıyla sıfırdan yazılan bir **UNIX klonudur**. POSIX standartlarına uymayı hedefler."*

Fakat Open Group **Unix-benzeri yapıları onaylamaz** ve bunu UNIX ticari markasının kötüye kullanımı olarak görür.

Linux sadece bir çekirdektir ve **Linux dağıtımları** onu kullanılabilir işletim sistemi haline getirir. Popüler ve yaygın birkaç Linux dağıtımı aşağıdaki gibidir: Redhat Enterprise Linux, Fedora Linux, Debian Linux, Suse Enterprise Linux, Ubuntu Linux.

## **WINDOW- LINUX(Unix) Temel Farklılıklar**

- WINDOWS POSIX standardına uygun farklı OS ların Windows Kernel üzerinde çalışmasına izin verir→ Sanal Makine
- UNIX te API fonk. Sistem Çağrıları olarak geçer

UNIX te kernel nesneleri SİSTEM ÇAĞRILARI ile yaratılır. WIN de WIN API ile.

- Unix çekirdek yapısı büyüktür (ilk versiyonları)
  - OS işlemlerinin büyük bir kısmını yüklenir
  - Sisteme ekleme yapmak zordur (çekirdek derlenmesi gerekir)
- WINDOWS çekirdek yapısı küçüktür(mikro kernel)
  - Yeni OS işlemleri sonradan yüklenebilen modüller olarak tasarlanır
  - OS genişletilmesi daha kolaydır
- WINDOWS'da GUI yönetimi, çekirdeğin bir parçasıdır, Linux da ise kullanıcı modülleridir. Windows arayüzü uzun zamandır çok büyük bir değişim göstermedi. Linux'da ise arayüz, çekirdek sistemden tamamen ayrı ve arayüz ortamınızı baştan yüklemelerle uğraşmadan değiştirebiliyorsunuz. KDE, Gnome, Cinnamon gibi çok kullanılan bazı arayüzlerle Linux'un çeşitli değiştirilebilir arayüzleri bulunmaktadır.
- Unix Kabuk programları (komut yorumlayıcıları) çoktur→ Bourne shell, C shell, Korn shell, RC shell, Almquist shell, ...,
  - Kabuk programları üzerinde scriptler daha çok programlanabilir (daha güçlüdür)
  - Linux'ün grafik arayüzü ne kadar gelişirse gelişsin, komut satırı üzerinden sağladığı yönetim fonksiyonları ileri seviye bilgisayar kullanıcıları için vazgeçilmez.

- WIN kabuk programı sadece cmd.exe ve explorer.exe dir. Windows'un komut satırı çoğu kullanıcının farkına bile varmayacağı kadar gözden uzak ve işlevsiz.
- UNIX te içsel komut kavramı yoktur. Bütün komutlar çalışabilen bir dosyadır.
- UNIX daha çok çoklu işlem (process) güdümlü iken WINDOWS çoklu işlem-parçacıkları (thread) güdümlüdür. (thread ler processlerden daha az kaynak kullanırlar)
- WINDOWS genel olarak 2 tip olarak karşımıza çıkar : (aynı anda) tek-kullanıcılı (WIN XP gibi) yada (aynı anda) çok-kullanıcılı (WINDOWS SERVER). UNIX direk çok kullanıcılı olarak doğmuştur.
- LINUX te işlemler arasında bir hiyerarşi vardır. WINDOWS ta işlemler aynı nesile aittir. Hiyerarşi gerekirse uygulama bunu sağlamalıdır. Linux'te bulunan çok katmanlı çalışma mantığı sayesinde işletim sisteminin grafik arayüzünde problem yaşansa bile komut istemcisiyle çalışmaya devam edebilmek mümkün. Windows ise arayüzde bir problem yaşanması durumunda ulaşılabilen bir güvenli kip bulunmasına karşın bu modda her istenilen yazılım çalışmayabiliyor ve sadece bu yol kullanılarak sorun giderilemeyebiliyor.
- LINUX te yapılandırma ayrı dosyalarda tutulur, Bir yerdeki arıza başka bir yeri etkilemez. Windows'da uygulama ayarlarını, donanım bilgilerini ve çok daha fazlasını tutan **kayıt defteri** (registry), Linux'da bulunmyor. Linux'da uygulamalar kendi ayarlarını

kullanıcılar altında ayrı ayrı tutuyorlar. Dolayısıyla Linux'da temizlik gerektiren bir veritabanı bulunmuyor.

- LINUX GNU/Özgür lisanslama modelidir, Açık kaynak kodludur. İstedığınız gibi kod üzerinde değişiklik yapabilirsiniz. WINDOWS Microsoft'a aittir. Hiçbir değişiklik yapamazsınız. Bilgisayarınızın göreceği zarardan ya da yazılımdaki sorunlardan dolayı Microsoft'u suçlayamazsınız. Olası sorunların düzeltilmesi için ücret ödemelisiniz.
- UNIX Ext2, Ext3, Ext4, Jfs, ReiserFS, Xfs, Btrfs, FAT, FAT32, NTFS dahil olmak üzere 250 den fazla dosya sistemi destekler, WINDOWS FAT, FAT32, NTFS, exFAT dosya sistemlerini destekler
- Linux her türlü donanım için optimize edilerek çalıştırılabilir. Minimum donanım gerekleri ise oldukça düşüktür. Windows yüksek performanslı donanımlar üzerinde çalışmak üzere tasarlanmıştır. Bu nedenle her türlü donanım üzerinde çalışmayabilir.
- Linux işletim sisteminde 60-100 arası virüs yazılmış olmasına karşın. Gelişen teknolojisi ile bugün bilinen bir virüs Linux'a zarar verememektedir. Windows işletim sistemi için yazılmış 100.000'den fazla virüs bulunmaktadır. Bunlardan korunmak için alınacak yazılımlar \$80-\$400 arası maliyetlerle piyasada satılmaktadır.



- Windows'da bir programı kurabilmek için genellikle onun yükleme paketini internetten bulup indirmeniz gerekiyor. Birçok Linux sisteminde bununla uğraşmanız gerekmez Paket Yöneticisi (Synaptic PY, Yazılım Merkezi), size uygulamalar arasında dolaşabileceğiniz, onları yükleyebileceğiniz ve kaldırabileceğiniz bir merkezi denetim alanı sunuyor.
- Linux işletim sistemi Linus Torvalds tarafından geliştirilmiştir. Çekirdek geliştirme ekibinin başında hala o vardır. Dünyanın hemen her yerinden binlerce geliştirici katkı vermektedir. Ücretli ya da ücretsiz destek alınabilmektedir. WINDOWS Microsoft tarafından üretilmiştir. Birkaç yüz geliştirici tarafından geliştirilmeye devam etmektedir. Ücretli destek alınabilir.

## **Sistem Programlama Nedir ?**

Sistem programlama, bilgisayar donanımı ve işletim sisteminin işleyişine doğrudan etki eden ve sistem kaynaklarını yönetme veya kontrol etme amacı taşıyan yazılımların geliştirilmesidir. Bu tür programlama, kullanıcı uygulamalarının altında çalışan yazılım katmanlarını içerir ve genellikle düşük seviyeli dillerle (örneğin, C veya Assembly) gerçekleştirilir. Sistem programlama, donanım ve işletim sistemi kaynaklarına doğrudan erişim gerektirir ve genellikle performans, güvenlik ve kaynak yönetimi konularında kritik bir rol oynar

Sistem programlama bir sisteme (donanım veya yazılımdan oluşan bilgisayar sistemi) servis sunan yazılımlar geliştirmeyi amaç edinen yazılım geliştirme modelidir. Sıradan programlama daha çok kullanıcılara servis sağlayan yazılımlar geliştirmeyi amaçlar. Uygulama yazılımları, kullanıcı yada uygulamanın gereksinimlerini karşılarken, sistem yazılımları

bir bilgisayar sisteminin gereksinimlerini karşılar. Bu sebeple, sistem programlama, altta yatan platformun (işletim sistemi veya makine donanımı) işlevselliklerini belli amaca yönelik olarak kullanarak, sistem çapında yazılım geliştirmeyi amaçlar. Sistem programlama modeli geliştirmelerin daha çok işletim sistemi ve altta yatan donanım arasında yada işletim sistemi ve uygulama katmanı arasında olduğu bir programlama modeli olarak görülebilir. Bu sebeple sistem programları geliştirilirken kullanılan diller aşağı seviyeli olma eğilimindedir. Bunları yazmak için belli miktar teori ve mühendislik bilgisi gereklidir.

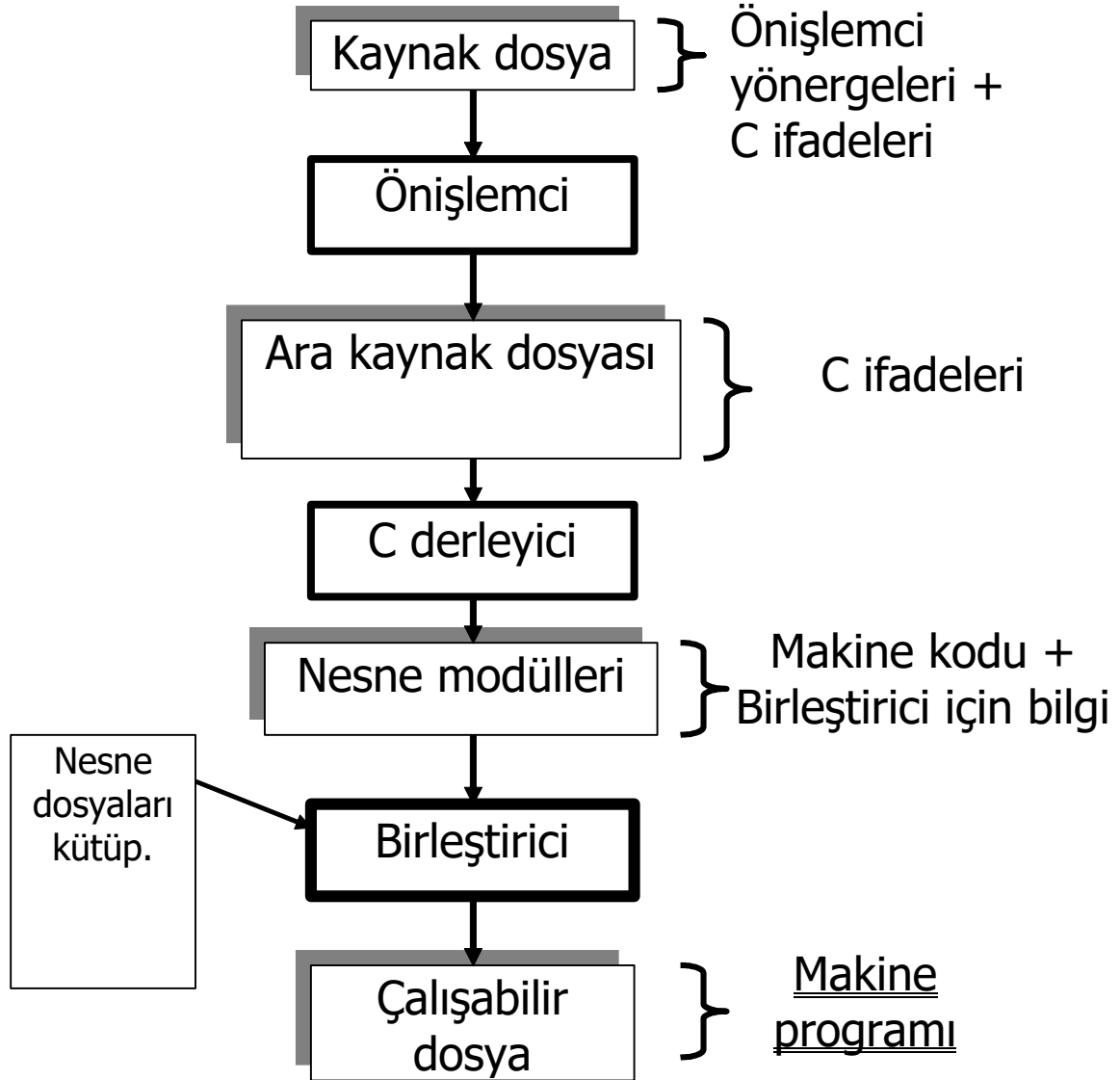
Sistem programlamadan günümüzde pratik olarak, yüksek seviyeli dillerde uygulama geliştirmek yerine (C#,Java, Ruby vb.) daha alt seviye diller yoluyla gerçekleştirilir. Sistem programlama faaliyetleri için en çok kullanılan diller C, C++ ve Sembolik Makine (spesifik ve performans gerektiren konular için makine dili gerekebilmektedir) dilleridir. İşletim sistemi çekirdeği ile standart C kütüphanesindeki fonksiyonlar ve sistem çağrılarını yoluyla iletişim kurulmasını gerektirir.

Tipik sistem programları şunlardır:

- İşletim Sistemleri
- Derleyiciler ve yorumlayıcılar
- Editörler
- Debug Programları
- Virüs ve Antivirüs yazılımları
- Haberleşme programları
- Gömülü sistem programları
- Aygıtların programlanması, aygıt sürücüler
- Veritabanı motorları

- Sanallaştırma yazılımları
- Oyun motorları
- ...

## Bir C programının uygulanmasındaki adımlar



**Önişlemci:** önişlemci direktiflerini (# ile başlayan komutlar) ve makroları (fonksiyon direktifleri) işletir, header dosyalarını dahil eder ve sonuç olarak derlenecek **kaynak dosyayı** bir bütün olarak **oluşturur**. Önişlemci direktifleri ve makrolar **Ders2b.pdf(lab dersi olarak)** dosyasında anlatılmıştır.

**Örnek** (modul1.c içeriği aşağıdaki gibi olsun. ilk.h header dosyası tüm moduller, modul1, modul2, ...moduln, tarafından kullanılmış olsun. Derleme OS ve kullanıcı bağımlı olsun. Kullanıcı Ali ise ali\_lib.h kütüphanesi kullanılsın) :

-----**ilk.h**-----

/\* kaynak dosyada tekrarlar hata üretir. Bu sebeple header içeriği, kaynak dosyaya eklenmediyse bileşenleri eklemek gerekir \*/

```
#ifndef _ilk_tanimlar_  
#define _ilk_tanimlar_
```

```
#define WIN32  
#define USER ALI
```

```
#endif
```

-----**modul1.c**-----

```
#include "ilk.h"
```

```
#if USER==ALI  
    #include "ali_lib.h"  
#endif
```

**/\* diyelim ki derleme OS bağımlı \*/**

```
#ifdef WIN32  
    ULONGLONG myvar;  
#else  
    unsigned long long myvar;  
#endif
```

**/\* diyelim ki derleme bit order 'a bağımlı \*/**

```
#if defined(LINUX) || defined(WIN32)  
    #define BIT_ORDER_LTOH  
    #undef BIT_ORDER_HTOL  
#elif defined(SOLARIS)  
    #undef BIT_ORDER_LTOH  
    #define BIT_ORDER_HTOL  
#endif
```

**Örnek**(Bir çok C derleyicisi -D seçeneği ile derleme zamanında makro atamalarına izin verir).

```
cc -D USER=ALI -o modul1 modul1.c
```

Diyelim ki modul1.c programı aynı zamanda ali\_lib.c kütüphanesine bağımlı. Bu durumda derleme aşağıdaki gibi olur.

```
cc -D USER=ALI -o modul1 modul1.c ali_lib.c
```

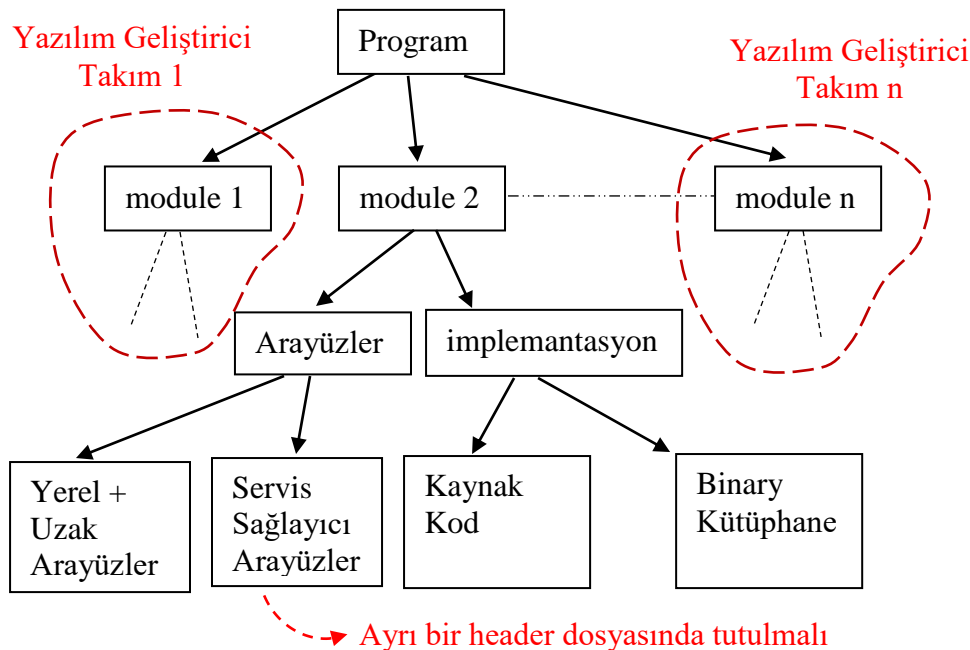
## Ara kaynak kodunu elde etmek için:

```
cc -Q produce .i <kaynak C dosyası>
```

**Derleyici:** Ara kaynak dosyasını alır, önce hedef makinanın assembly diline ve sonra makina koduna çevirir. Sonuç makine kodu ve çözümlenemeyen referanslar içeren bir nesne modülüdür. Bu modül aynı zamanda birleştirici için bilgide taşır.

**Birleştirici:** Bir grup nesne dosyalarını birlikte alır ve çalışabilen tek bir modüle dönüştürür. Bu modül artık içerisinde çözümlenemeyen referanslar içermez. Referansların çözümlenmesi nesnede kullanılan kütüphanelerin birleştiriciye tanımlanması ile olur.

## Tipik bir programın tasarımı



Program modüllere ayrılır, her bir modül bir çok işlevlerden oluşur. Bu işlevlerin bir kısmı sadece modül içinde kullanılır bazıları ise diğer modüllere servis sunan, servis sağlayıcılarına ait olabilir. Modüller, diğer modüllerden hizmet almak için sadece servis sağlayıcıların sunduğu arayüzleri (örneğin API'ler) bilmesi yeterlidir, bu arayüzlerin implementasyonu bilinmesi gerekmez.

İçeriğe göre API(Application Programming Interface) ifadesinin anlamı değişmekle birlikte, genel olarak API'ler servis sağlayıcı katmanına ait servisleri, bu servislerin implementasyonuna erişmeksizin, kullanıcı katmanında sadece arayüzleri ile kullanmayı sağlar.

Servis sağlayıcıların sunduğu işlevlerin, **arayüz ve implementasyonu ayrıştırılması** gereklidir. Bunun sebebi bir yazılımcı ekibin, diğer ekibin ürettiği servislerin implementasyonu olmadan yazılım geliştirme sürecini paralel bir şekilde devam ettirmektir.

## **UNIX te bir C programının hazırlanması**

### **Durum 1:** Program sadece bir kaynak dosyadan oluşur

1. Bir metin düzenleyicisini açtıktan sonra(örneğin PICO), kaynak C kodunu yazıp kaydedin(örneğin <b>myprog.c</b> ).	%pico . . . . Pico ile çalışma . . . .
2. Programı derleyip birleştirin. Sonuç çalıştırılabilir bir dosya (Örneğin, <b>myprog</b> ).	%cc -o myprog myprog.c
3. Programı başlatın	%myprog veya %myprog param1, param2.

**Durum 2:** Program bir yada daha fazla kaynak dosyadan oluşmuşsa.  
(örneğin, module1.c, module2.c, module3.c)

1. Metin düzenleyicisini açıp, her bir kaynak dosyayı ayrı ayrı kaydedin.	
2. Her bir kaynak dosyasını ayrı ayrı derleyin. Sonuç bir grup nesne dosyalarıdır (module1.o, module2.o, module3.o)	<pre>%cc -c module1.c %cc -c module2.c %cc -c module3.c</pre>
3. Nesneleri birleştirin. Sonuç bir çalışabilen dosyadır.	<pre>%cc -o myprog module1.o module2.o module3.o</pre>

## **make** dosyası kullanımı



**make** uygulaması çalıştırıldığında, bulunulan dizinde **makefile** dosyası aranır ve bu dosyaya göre derleme yapılır.

### **Örnek** (uygun makefile örneği)

CC=cc

CFLAGS = -g

LIBS = /lib/libm.a

all: prog1 prog2

prog1 : prog1.o

tab \${CC} \${CFLAGS} -o prog1 prog1.o

prog1.o: prog1.c

tab \${CC} -c prog1.c

prog2 : prog2.o

tab \${CC} \${CFLAGS} -o prog2 prog2.o \${LIBS}



```
prog2.o: prog2.c  
    tab    ${CC} -c prog2.c
```

```
clean:  
    tab    rm -f *.o *#core prog1 prog2
```

## **makefile kullanımı:**

%make

Gerekli yeniden derlemeler otomatik yapılır.

## **Make Dosyalarının İçeriği**

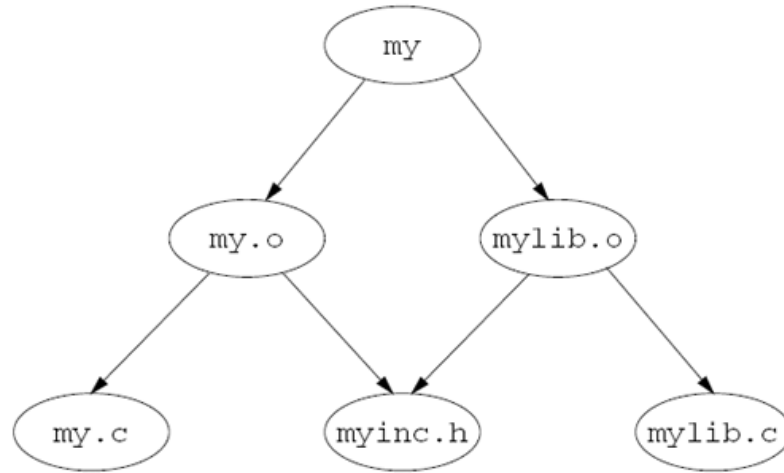
1-Macro tanımlamaları # zorunlu değil
2-all: hedef1 hedef2 #derlenecek hedefler(zorunlu değil)
3-kurallar #gerekli

Bir kuralın genel biçimi şöyledir:

```
hedef: bağımlılıklar  
<TAB> hedefi üretecek komut  
<TAB> ...  
Diğer kurala geçmeden bir boş satır  
...
```

: satırına bağımlılık (dependency) denilmektedir. Bağımlılık şu anlama gelir: :’nin sağındaki dosyalardan herhangi birinin zamanında değişiklik olursa sol taraftaki hedefler(birden fazla olabilir) yeniden üretilir. Kuralların sırası önemlidir. Bağımlılığı en yüksekten, en düşüğe doğru bir yazım biçimi uygulanmalıdır.

**Örnek** (bağımlılık grafiği aşağıdaki gibi olan my programına karşılık gelen makefile içeriği)



```
my:    my.o mylib.o
      cc -o my my.o mylib.o
my.o:  my.c myinc.h
      cc -c my.c
mylib.o: mylib.c myinc.h
      cc -c mylib.c
```

**Örnek:** makefile dosyaları içerisinde shell'in çevre değişkenleri kullanılabilir. Ayrıca bir makro komut satırından da girilebilmektedir. Örneğin;

make "OBJS = a.o b.o"

make içerisinde hiç tanımlamadan kullanılabilecek önceden tanımlanmış çeşitli makrolar da vardır. Bunların bazıları şunlardır:

CC = cc

SHELL = /bin/sh

Örneğin;

#deneme

x : \${OBJS}

{CC} -o x \${OBJS}

a.o : a.c

```
CC -c a.c
b.o : b.c
CC -c b.c
${OBJS} : x.h
```

## **Birden fazla kaynak dosyadan oluşmuş program örneği : hadi derleyelim**

hellomake.c	hellofunc.c	hellomake.h
<pre>#include "hellomake.h"  int main() { //dış fonk. çağırımı     myPrintHelloMake();      return(0); }</pre>	<pre>#include &lt;stdio.h&gt; #include "hellomake.h"  void myPrintHelloMake(void) {     printf("Hello makefiles!\n");      return; }</pre>	<pre>/* example include file */  void myPrintHelloMake(void);</pre>

Normalde tek satırda şöyle derleme yapılabilir

**%gcc -o hellomake hellomake.c hellofunc.c -I.**

Burada **-I**dir: header dosyalarını **dir** dizininde ara demektir.

### Makefile 1

```
hellomake: hellomake.o hellofunc.o
    gcc -o hellomake hellomake.c hellofunc.c -I.
```

### Makefile 2

```
CC=gcc
CFLAGS=-I.
```

```
hellomake: hellomake.o hellofunc.o
    $(CC) -o hellomake hellomake.o hellofunc.o $(CFLAGS)
```

### Makefile 3

```
CC=gcc
CFLAGS=-I.
```

```
hellomake: hellomake.o hellofunc.o
$(CC) -o hellomake hellomake.o hellofunc.o $(CFLAGS)
```

```
hellomake.o: hellomake.c hellomake.h
```

```
$(CC) -c hellomake.c $(CFLAGS)
```

```
hellofunc.o: hellofunc.c hellomake.h
```

```
$(CC) -c hellofunc.c $(CFLAGS)
```

#### Makefile 4

```
CC=gcc
```

```
CFLAGS=-I.
```

```
DEPS = hellomake.h
```

```
OBJ = hellomake.o hellofunc.o
```

```
hellomake: $(OBJ)
```

```
$(CC) -o $@ $< $(CFLAGS)
```

```
%.o: %.c $(DEPS)
```

```
$(CC) -c $< $(CFLAGS)
```

“.” uzantılı tüm dosyaları işle ve dosya adını % ile temsil et

Sol Taraf(hellomake)

Sag Taraf (OBJ değeri)

## C programında 2 tip main fonksiyonu

```
main( ){  
    Main ana fonk.  
}
```

Komut satırı parametreleri kullanmadan  
Başlatmak için:  
%program\_ismi

```
main(argc, argv)  
int argc ;  
char *argv[ ] ;  
{  
    Main ana fonk.  
}
```

Bir yada daha fazla komut satırı parametreleri kullanarak  
Başlatmak için:  
%program\_ismi par1 par2 ..

### Örnek:

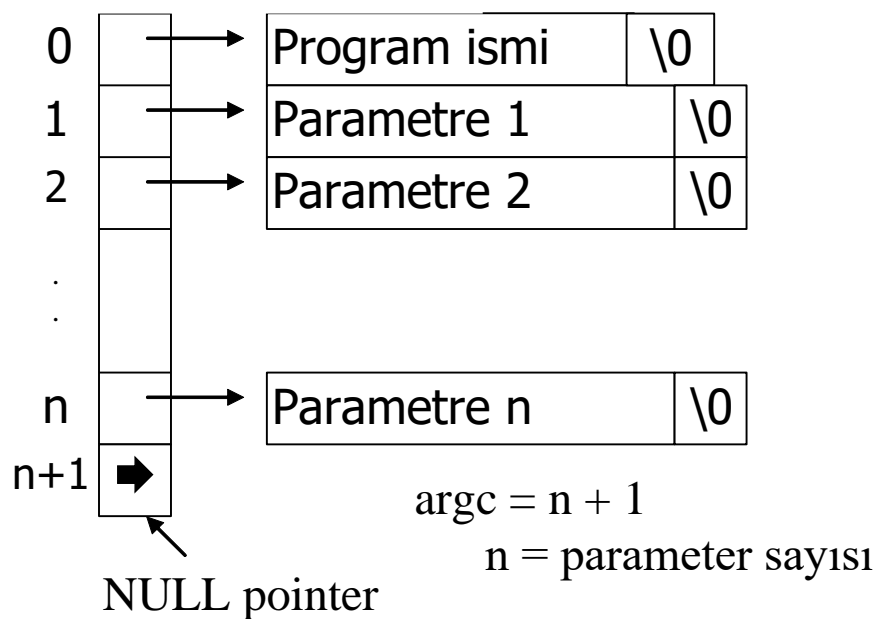
%time  
%mkdir **dizin\_adı**

← parametresiz  
← **parametrelili**

## **main()** fonksiyonun parametrelerini anlamak

```
main (argc, argv )
    int  argc ;           /* komut satırı kelime sayısı */
    char *argv[ ] ;      /* parametrelere pointer */
{
    . . . . .
}
```

**\*argv [ ]** yapısı



Using argv in the program:

argv[0] – program adını gösterir

argv[1] – parametre 1 adını gösterir

\*argv – program adını gösterir

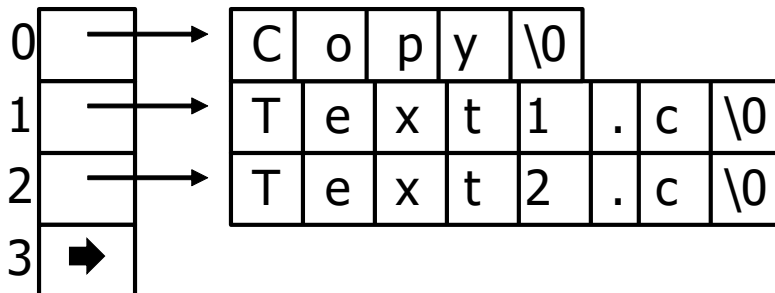
\*argv[2] – program adındaki 2. karakteri gösterir

## **Örnek:**

copy text1.c text2.c

argc = 3

\*argv [ ] yapısı



## **Örneklerle Komut Satırı Parametreleri:**

### **Örnek-1**

myprog.c dosyasının içeriği aşağıdaki gibi olsun :

```
#include <stdio.h>
main(argc, argv)
int argc; char *argv[] ;
{ for ( ; *argv ; ++argv )
    printf("%s\n", *argv ) ;
}
```

Derleme işlemini yapalım :

```
% cc -c myprog.c
%cc -o myprog myprog.o
```

## Derleme işlemi tek modül için kısa yol:

% cc -o myprog myprog.c

## Komut Satırı:

%myprog this is a test

## Output:

myprog  
this  
is  
a  
test

## **Örnek-2:**

```
/* kaynak myprog.c , calisan dosya myprog */  
#include <stdio.h>  
main ( argc, argv )  
    int argc ;  
    char *argv [ ] ;  
{ if ( argc < 2 )  
    { printf( "Usage : %s parameter\n", argv[0] ) ;  
      exit ( 1 ) ;  
    }  
    printf("Starting program %s \n", argv[0] ) ;  
    printf("with %d parameter(s)\n", argc-1 ) ;  
    printf("First parameter is %s\n", argv[1] ) ;  
    exit ( 0 ) ;  
}
```

---



### Komut satırı1:

%myprog /\* hatalı (yada parametresiz) giriş \*/

### Output:

Usage: myprog parameter

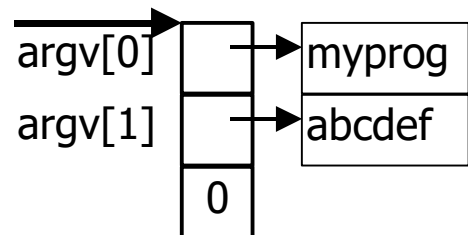
### Komut satiri2:

%myprog abcdef /\* ← doğru giriş \*/

### Output:

Starting program myprog  
with 1 parameter(s)

First parameter is abcdef



### Programin bazı esdeger ifadeleri:

arv[0] → \*argv

arv[1] → \*++argv

## Ornek 3 :

```
/* Kaynak myprog.c , integer parameter */
```

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

```
main ( argc, argv )
```

```
    int  argc ;
```

```
    char *argv [ ] ;
```

```
{ int  p ;
```

```
    if ( argc < 2 )
```

```
    { printf( "Usage : %s parameter\n", argv[0] ) ;
```

```
      exit ( 1 ) ;
```

```
    }
```

```
    printf("Starting program %s \n", argv[0] ) ;
```

```
printf("with %d parameter(s)\n", argc-1 );  !!!
```

```
    p = atoi(argv[1] );  
    printf("First parameter is %d\n", p ) ;  
    exit ( 0 ) ;  
}
```

---

Komut Satiri:

%myprog 12

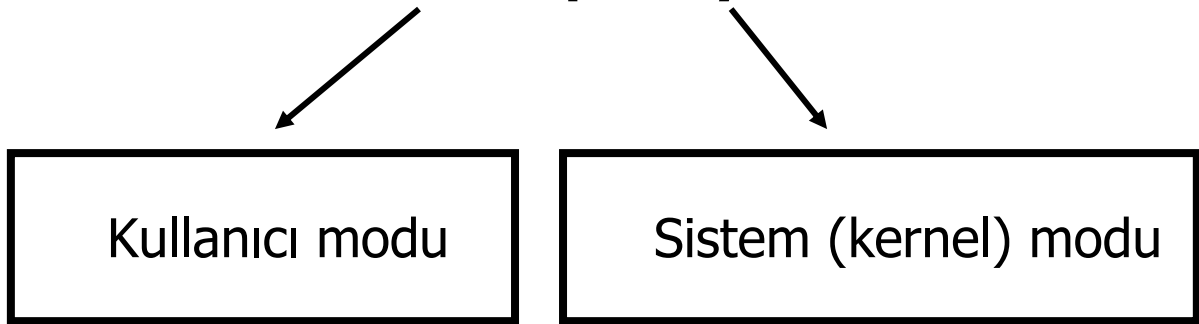
Output:

. . . . .

. . . . .

First parameter is 12

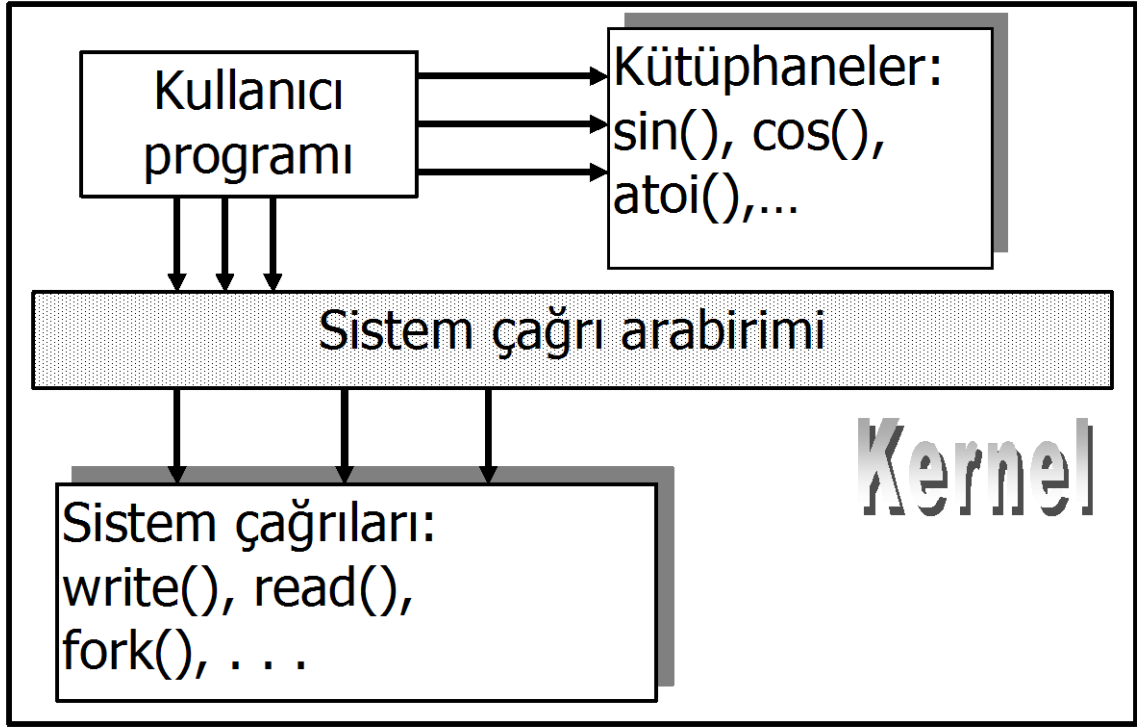
## **UNIX te 2 operasyon modu**



Bir çok operasyon yasak Tüm operasyonlar izinli

- Kesme aktif etme
- Kesme pasif etme
- I/O aygıtlara direk erişim
- vesaire.

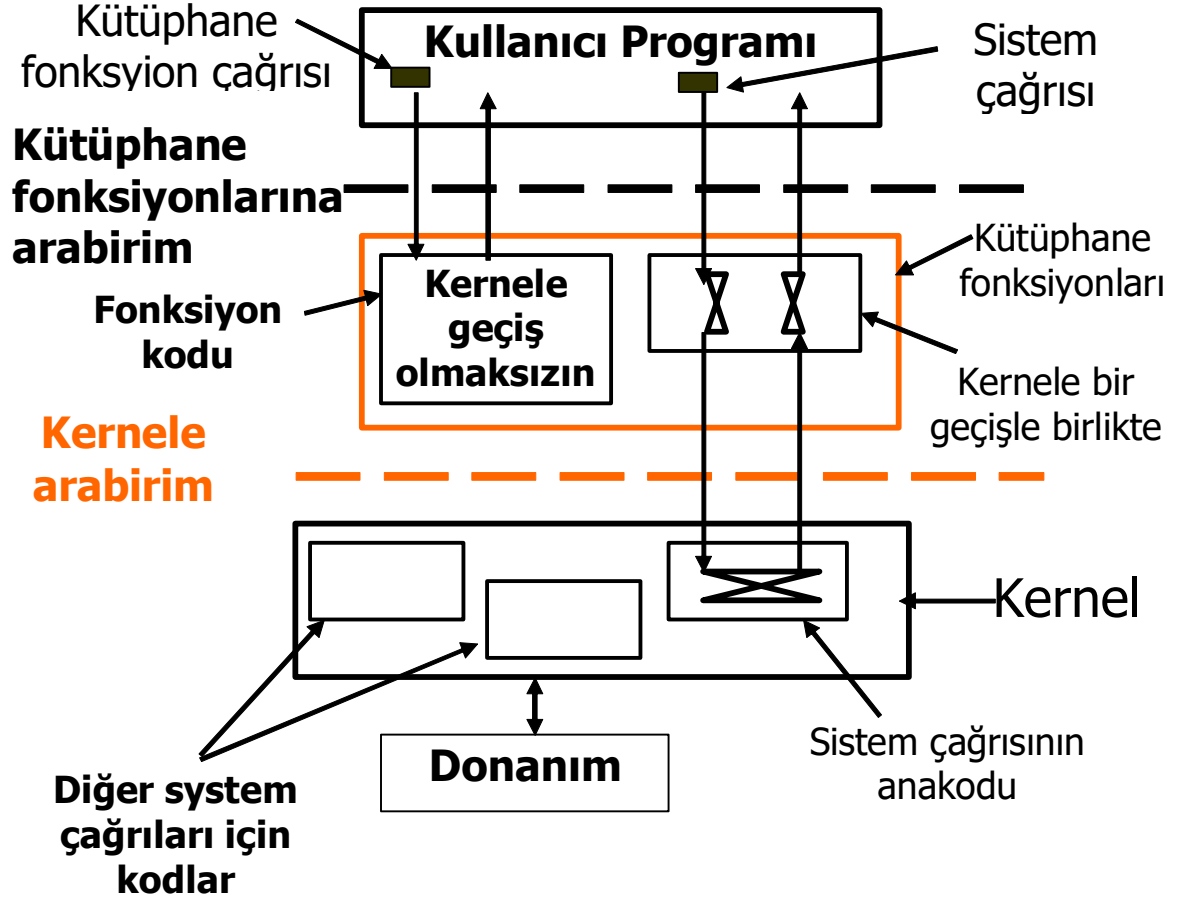
## Kütüphane çağrıları ve sistem çağrıları



### **Dikkat:**

1. Standart kütüphane fonksiyonları system çağrı arabirimini ilgilendirmez
2. Sistem çağrıları her zaman system çağrı arabirimi ve OS kernelini kullanır.

## Kullanıcı Programı, kütüphane fonksiyonları ve system çağrıları

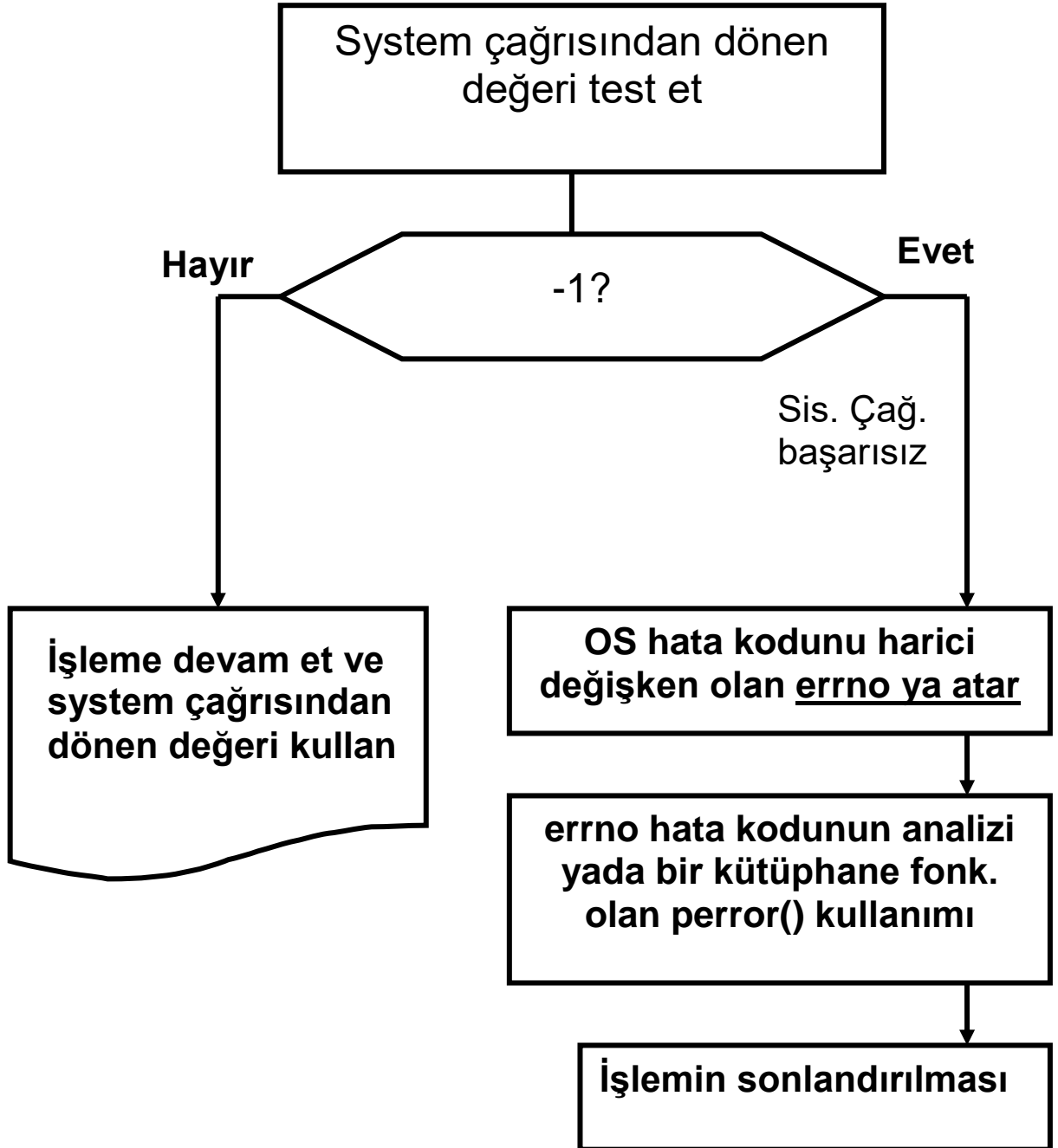


### Bir system çağrısının genel formu :

[ dönen\_değer = ] system\_çagri\_adi ( parametreler ) ;  
(eğer varsa)

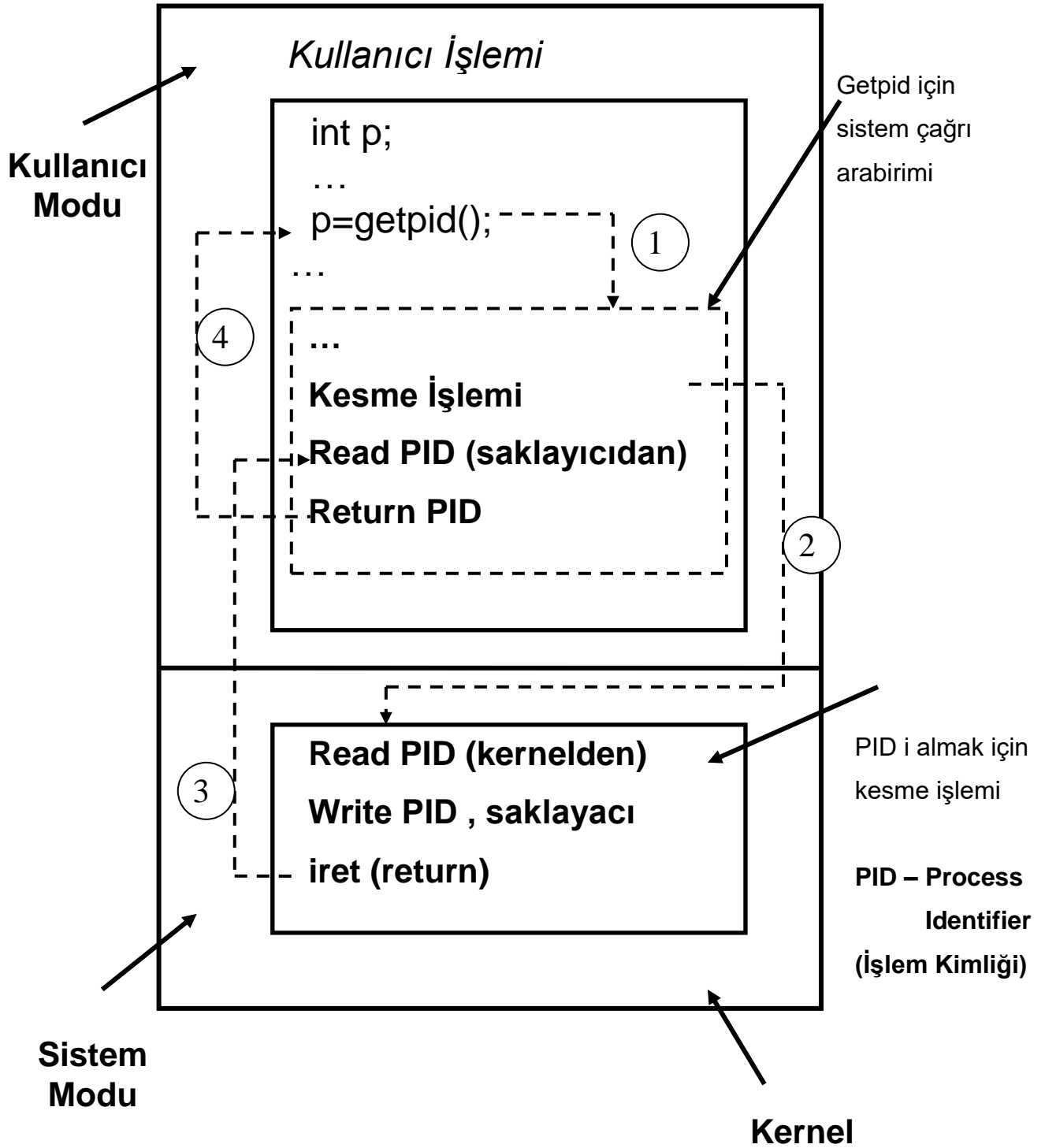
dönen\_değer =  $\left\{ \begin{array}{l} \text{pozitif tamsayı , eğer tamamsa} \\ -1, \text{ eğer hatalıysa} \end{array} \right.$   
(bu durumda hata kodu harici değişken olan *errno* ya *yerleşir*)

## **Bir system çağrısından sonra programdaki hareketler**



## LINUX ta bir Sistem Çağrısının Uygulanması

### Örnek : getpid system çağrısı için



## Kullanıcı Programında system çağrısının tipik kullanımı

```
#include <fcntl.h>
#include . . .
```

```
main( argc, argv )
```

```
int argc ;
char *argv[] ;
{
```

```
    int fd ;
```

```
    . . . . .
```

```
    fd = open( argv[1], O_RDONLY ) ; /* sis. çağrısı */
```

```
    if ( fd == -1 )
```

```
    {
```

**Hata mesajı yaz ve çık**

```
    }
```

```
    . . . . .
```

```
    perror( argv[1] ) ; exit( 1 ) ;
```

**Program Kullanımı :**


%programe filename

argv[1]



## perror() kütüphane çağrısının kullanımı

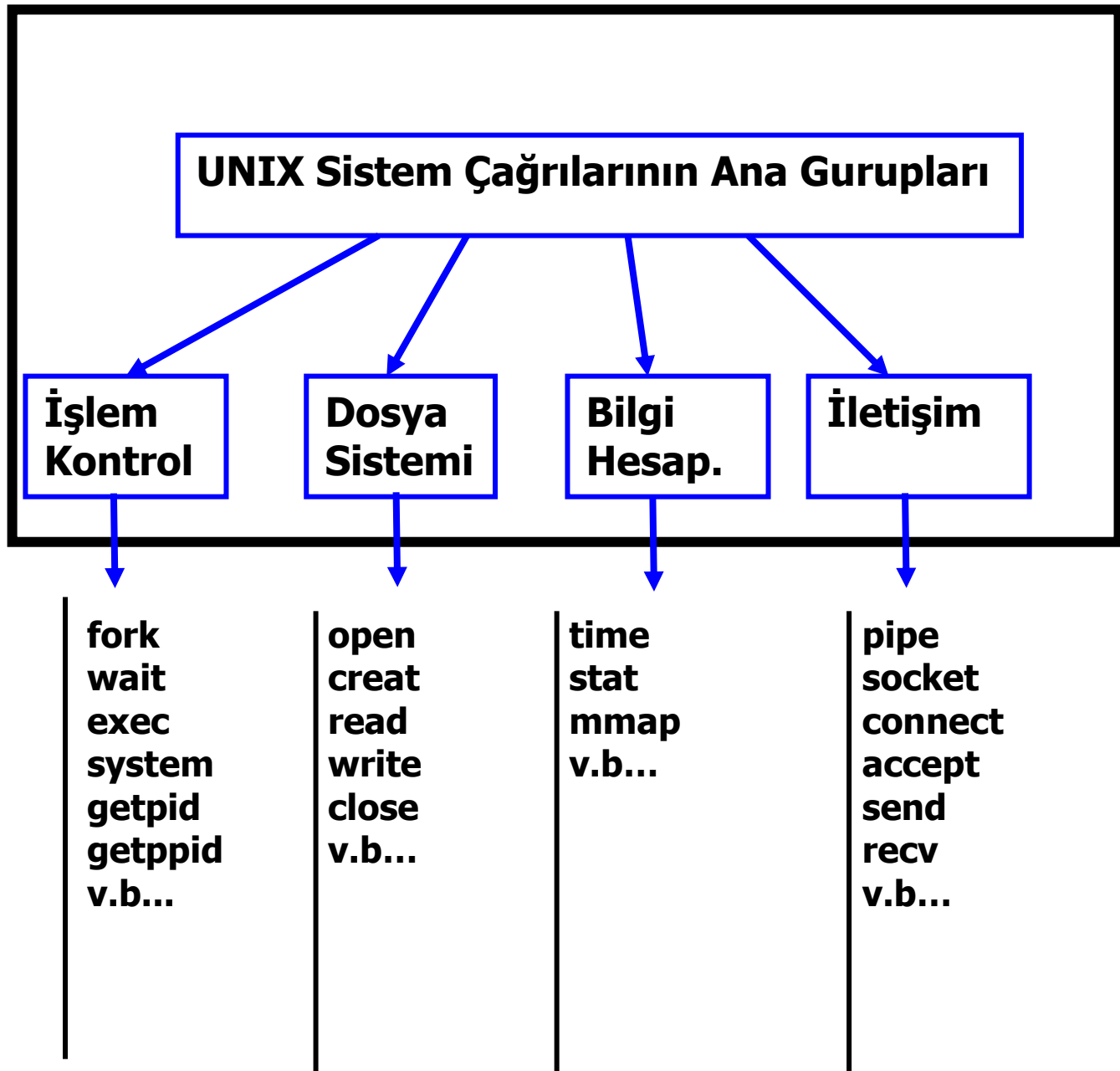
```
#include <stdio.h>
. . . . .
main( )
{
    int fd ;
    fd = open("file_name", . . . ) ;
    if ( fd == -1 )          /* Başarısız      */
    { perror("file_name") ; exit(1) ; }
    . . . . .
}
```

 **String yada String değişkeni**

**Başarısızsa Output Hata Mesajı :**

file\_name : **Bazı sistem mesajları**

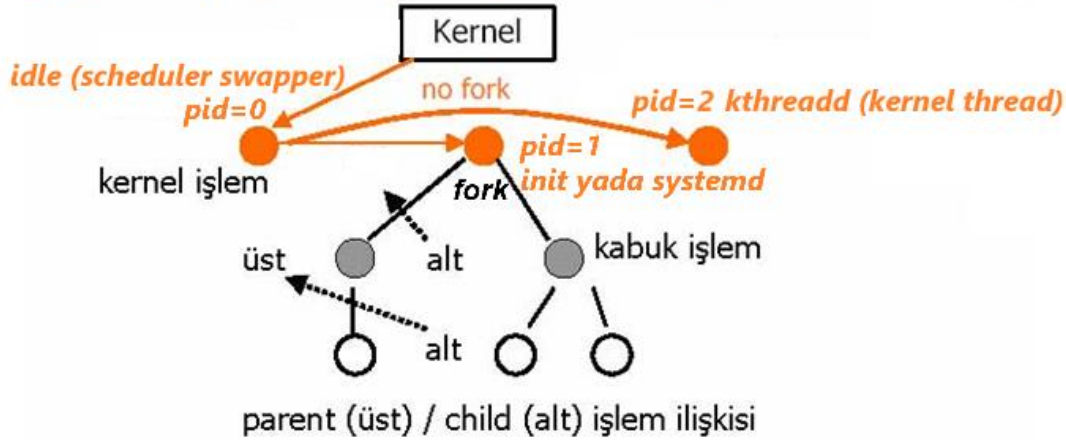
 **Örneğin**      No such file or directory



**Hatırlatma : Windows OS de system çağrıları = API**

## Unixte İşlemler (prosesler)

*pid(proses id)=0, pid=1 ve pid=2 prosesleri direk çekirdek tarafından başlatılır (no fork)*



**pid=1 (init yada systemd) , pid=0 ve pid=2 dışında tüm işlemleri yaratan işlem**

*pid 0 idle (boşta) zamanda arka alanda swapper gibi temel işlemler yapar*

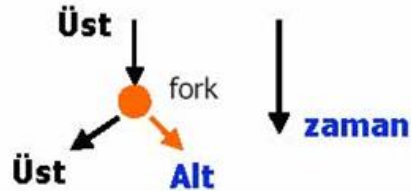
*pid 2 Kernel thread'leri yönetir (bellek, disk v.b. yönetimler için thread yaratır ve yönetir)*

*pid\_t fork ( void ) ;*

- fork alt (child) işlem yaratan bir sistem çağrısıdır.
- Dönen Değer
  - pid\_t=0 ise işlem alt (child) işlem
  - pid\_t>0 ise alt işlemin PID değeri
  - pid\_t<0 ise fork işlemi başarısız

Mümkün hata mesajları :

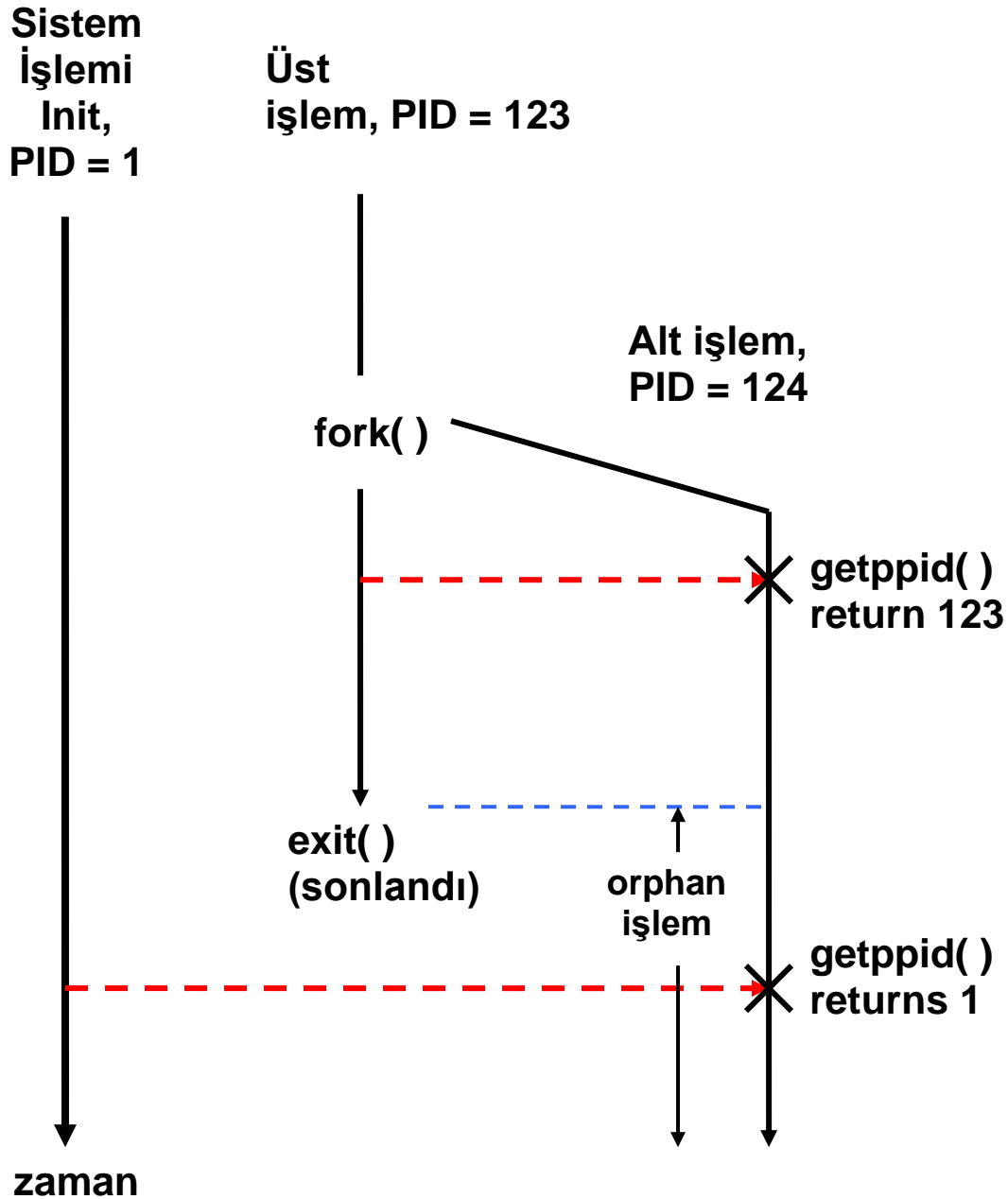
- Resource temporary unavailable (CPU kaynağı bitti)
- Not enough space (Bellek kaynağı bitti)



UNIX/LINUX sistemlerinin proses yapısı tamamen hiyerarşik bir yapı göstermektedir. Diğer işletim sistemlerinde olduğu gibi POSIX sistemlerinde de bir proses başka bir programı çalıştırabilir yani yeni bir proses yaratabilir. Yeni çalıştırılan proses (child proses) ile onu çalıştıran proses (parent proses) arasındaki ilişki Windows sistemlerine göre sıkıdır. Klasik olarak POSIX sistemlerinde her proses'in bir ID değeri (pid) vardır. Bu ID değeri sistemde tektir.

POSIX sistemleri yüklendiğinde temel yükleme işlemleri için üç adet proses çekirdek tarafından direk olarak yaratılır (pid=0, 1 ve 2). Pid=0 prosesi (idle proses), CPU'nun boşa kaldığı anlarda işlem yapar. Genellikle, hiçbir iş yapılmadığında çalışacak bir işlem gibi düşünülebilir. Bu süreç son kullanıcı tarafından görünmez ve sistemin iç işleyişinde kullanılır. Pid=1 (init proses) artık daha çok systemd proses olarak bilinmektedir. Fork yaparak diğer tüm prosesleri yaratan sistem prosesidir. Sisteme girmek için kullanılan login de bir proses olup, init proses tarafından çalıştırılır. Login tipik olarak init proses'inin bir alt proses'dir (child proses). Kullanıcı username ve password bilgilerini başarılı bir biçimde girdiyse /etc/passwd dosyasında belirtilen shell programı çalıştırılır. Yani shell proses'i tipik olarak login proses'inin alt proses'i olarak çalışır. Bu durumda tipik bir POSIX sisteminde sisteme login olduğunda proses hiyerarşisi şöyle olacaktır. Bu noktada artık shell üzerinden bir program çalıştırsak çalışan proses shell proses'i olarak çalıştırılacaktır. UNIX/Linux sistemlerinde bir proses'in yeni bir proses'i çalıştırması fork ve exec fonksiyonları ile yapılmaktadır. Bu sistemlerde yaratılan alt proses'ler üst proses'in pek çok bilgisini doğrudan almaktadır.

# Üst işlem alt işlemden önce ölürse ne olur ?



## İşlem ID (PID) nasıl öğrenilir ve İşlemin orphan durumu

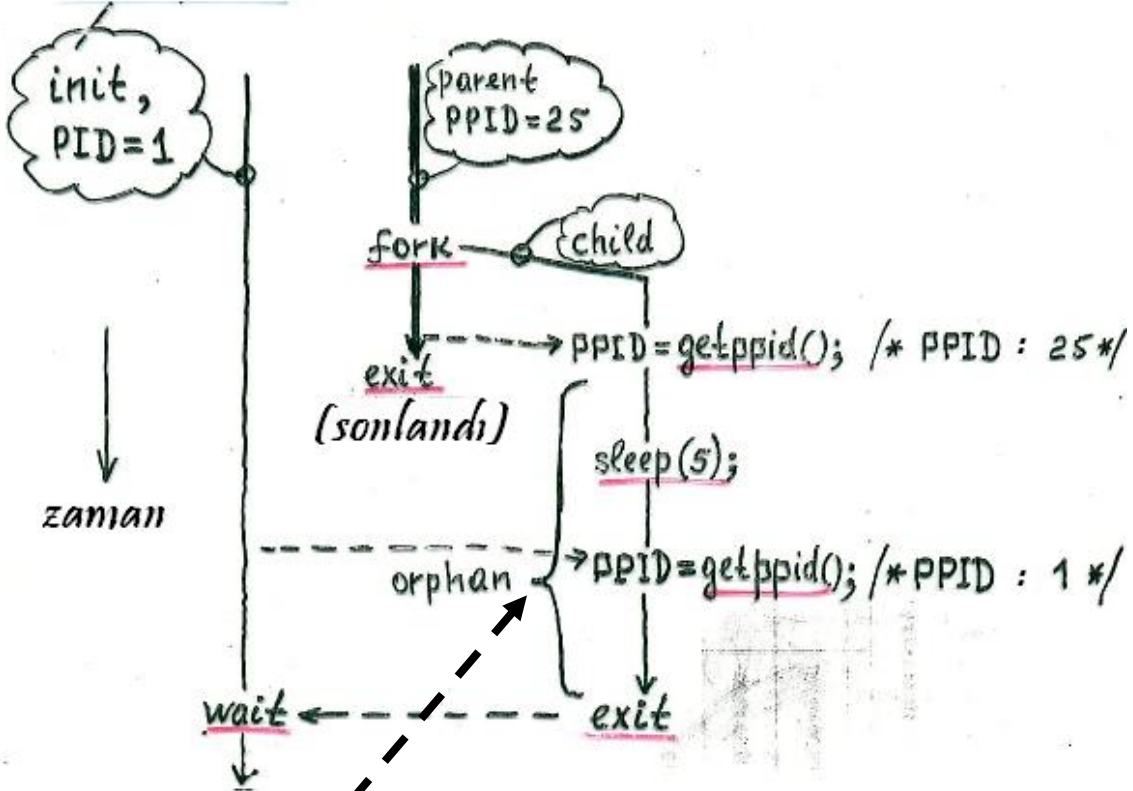
`pid_t getpid(void);`  
`int`  $\swarrow$   
`pid_t getppid(void);`

include dosyaları :  
<sys/types.h>  
<unistd.h>

örnek:

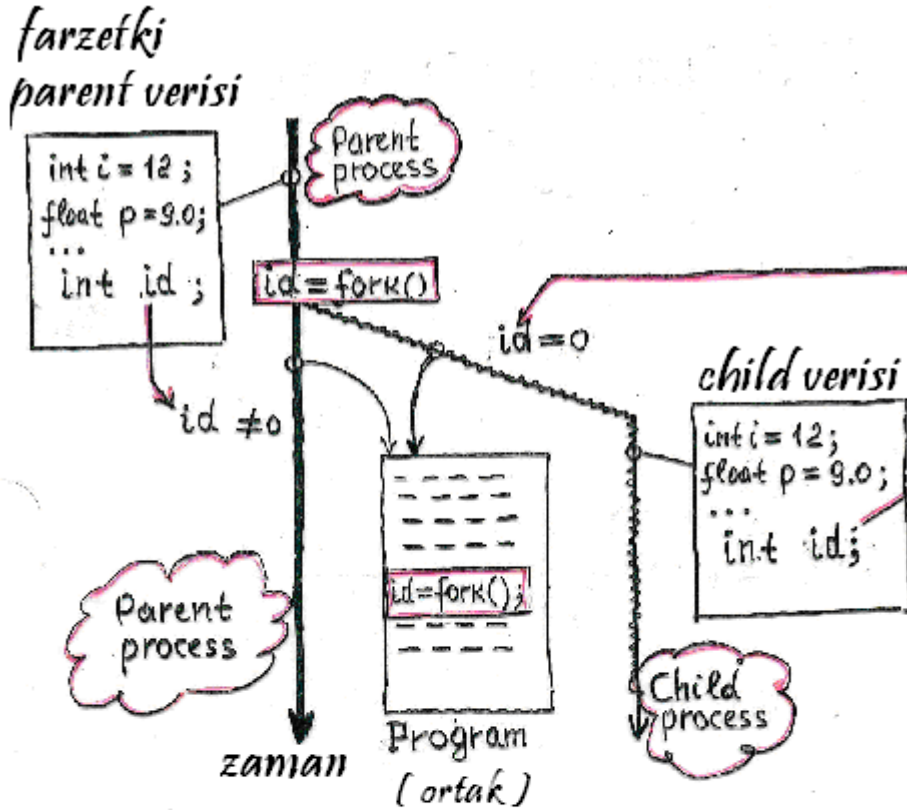
```
printf("My PID : %d\n", getpid());  
printf("My parent PID : %d\n", getppid());
```

Sistem islemi

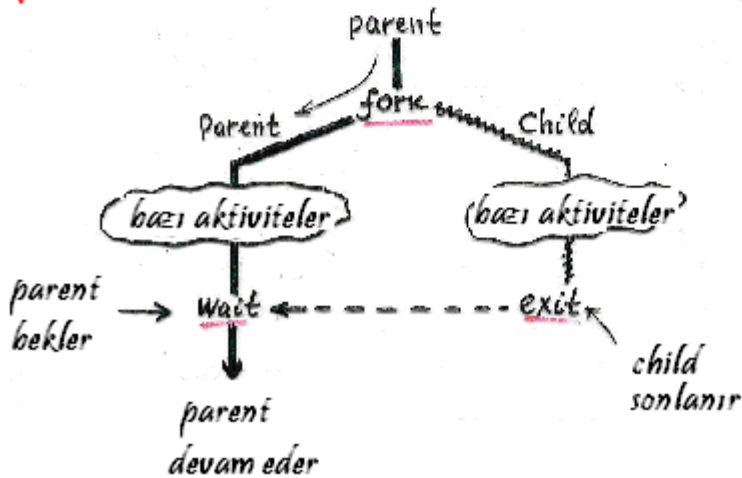


Bir işlemin orphan durumu

## Üst ve alt işlemlerin adres uzayları



parent ve child arasındaki senkronizasyon :



## Alt islemin zombie durumu

$P$  : Parent ,  $C$  :  $P$ 'nin Child (alt) işlemi olsun

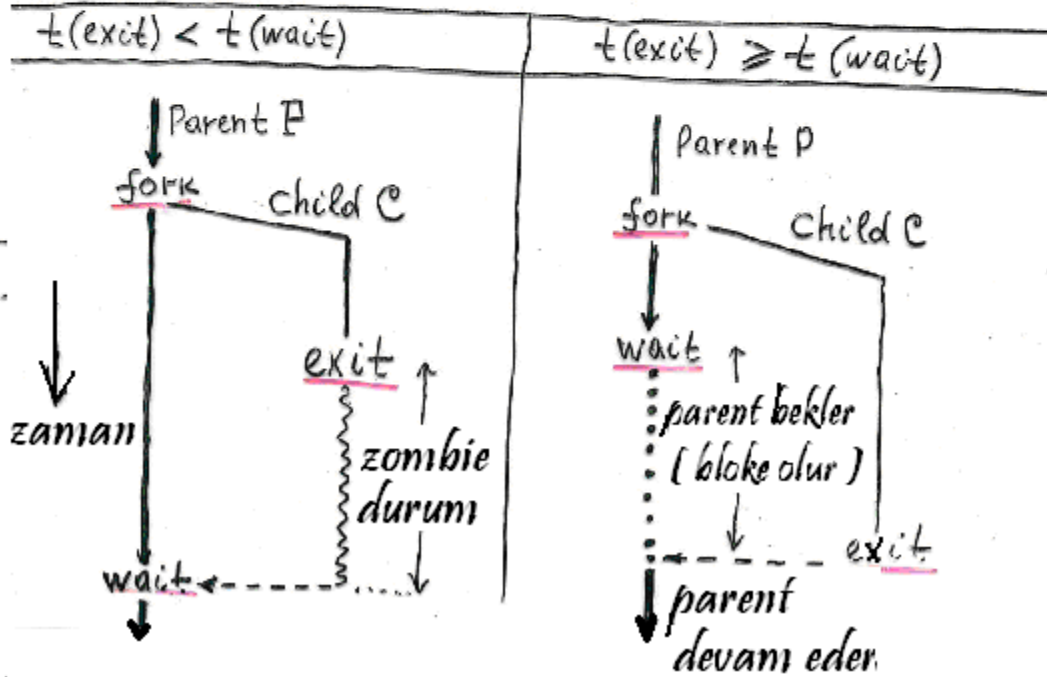
Aşağıdaki gösterimleri kabul edelim :

$t(\text{fork})$  , parent  $P$  nin sistemi çağrısı fork u  
islettiği zamanı,

$t(\text{exit})$  , child  $C$  nin sonlandığı zamanı,

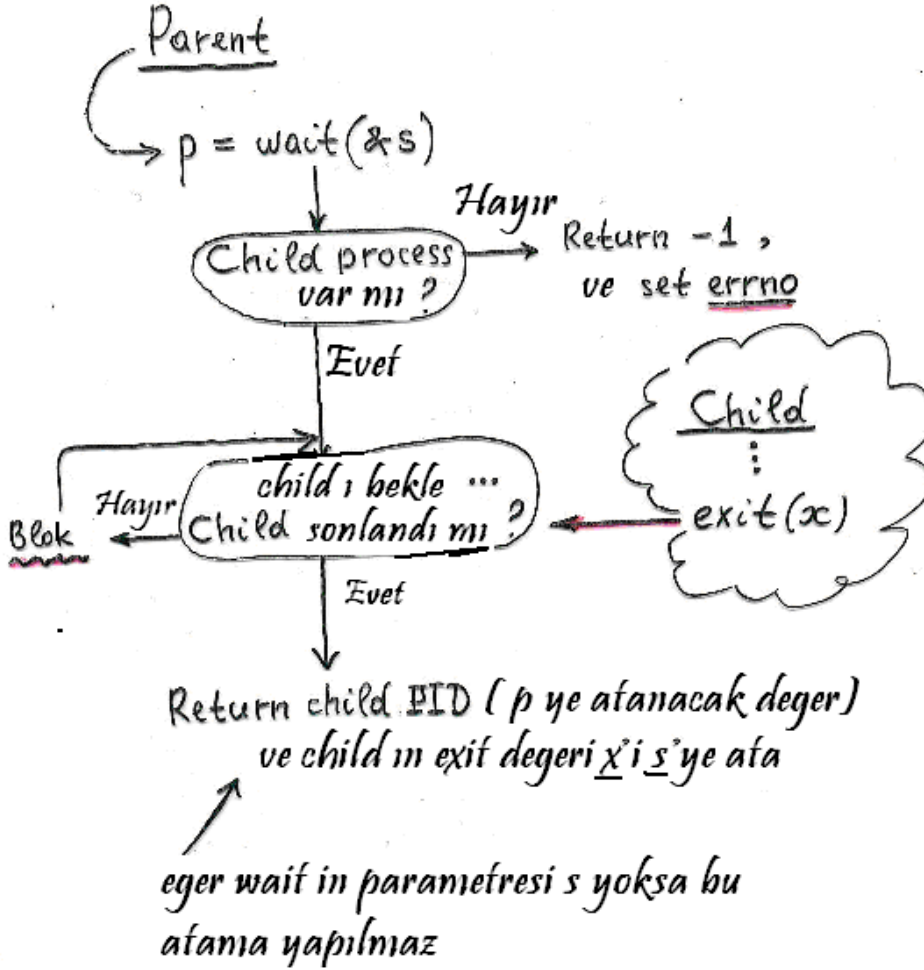
$t(\text{wait})$  , parent  $P$  nin sistemi çağrısı wait i  
islettiği zamanı,

açıktır ki,  $t(\text{fork}) < t(\text{exit})$ ,  $t(\text{fork}) < t(\text{wait})$





## UNIX te sistem çağrısı wait ve exit arasındaki ilişki

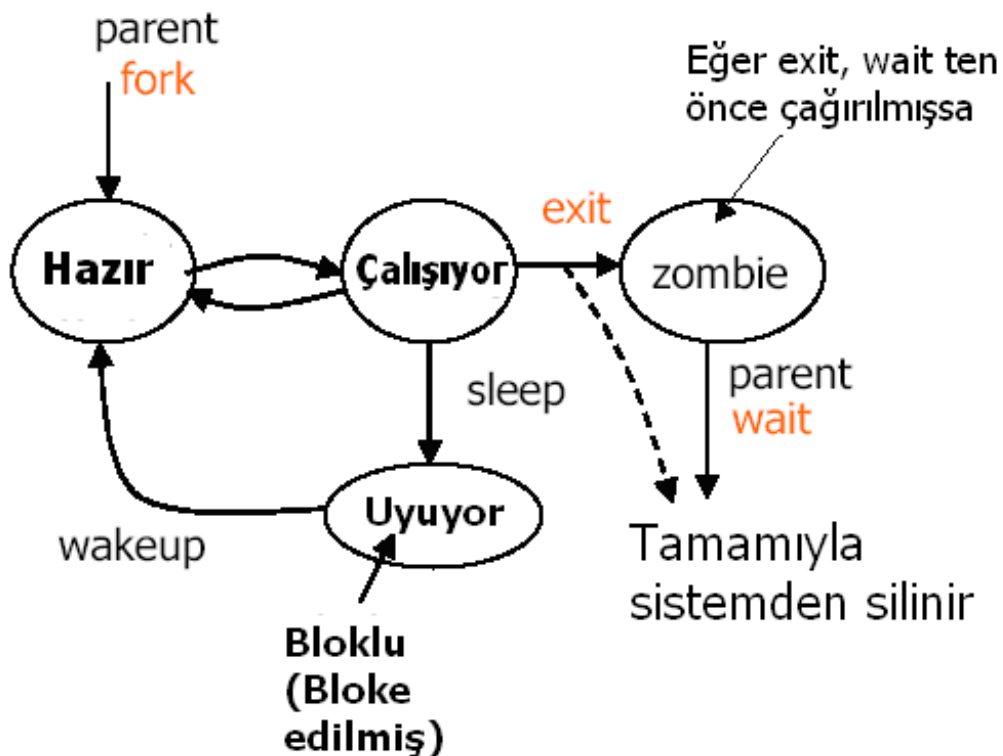


Bir işlem sonlandığında (`exit`), kullandığı tüm kaynaklar iade edilir sadece `exit` statü değeri process tablosunda bırakılır bu şekilde üst işlem alt işlemin `exit` kodunu öğrenir ve sistemden tamamını iletilir. Üst işlem ölmüşse bu işi `init` process yapar

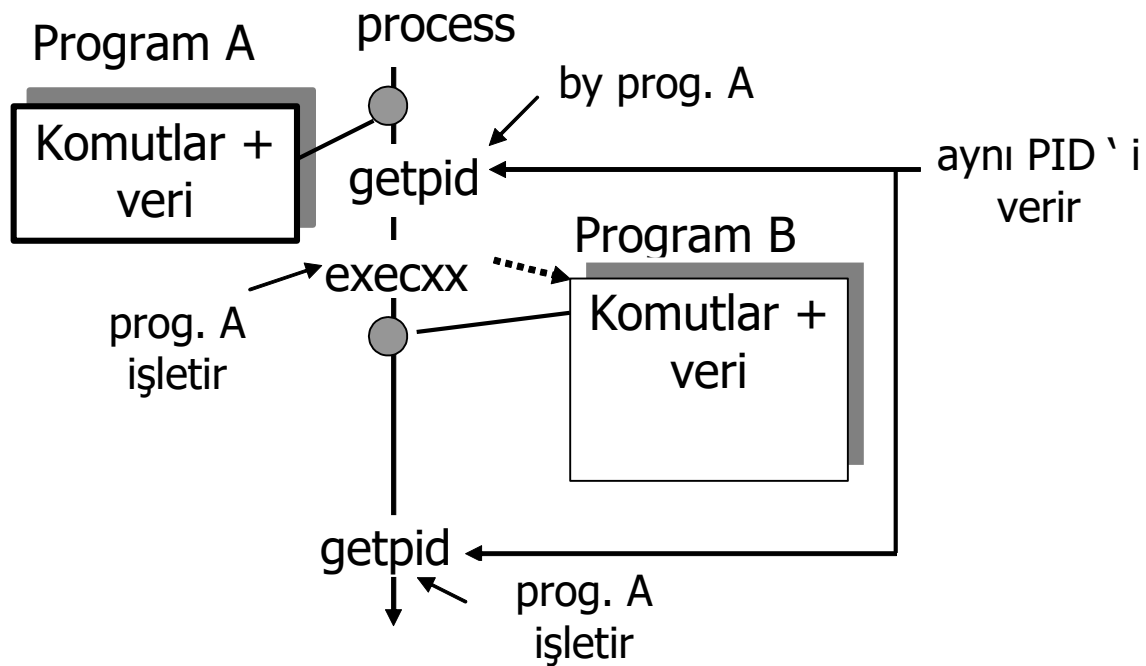
## Örnek : üst proses tüm çocukların sonlanmasını bekliyor

```
int i, pid, status, w;  
:  
for (i=0; i<3; ++i) /* starting 3 children */  
{  
    pid = fork();  
    if (pid==0) { child works and terminates }  
}  
  
while ((w = wait(&status)) && w != -1)  
    printf("Child %d returns status %d\n", w, status,
```

## UNIX te proses durumları (Basitleştirilmiş)



## Bir process içinde başka programa geçiş



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

```
int execl ( const char *path, const char *arg0, ...,  
            const char *argn, char * /*NULL*/ );
```

**Başarılı ise Dönen Değeri:** Hayır (Dönmez)

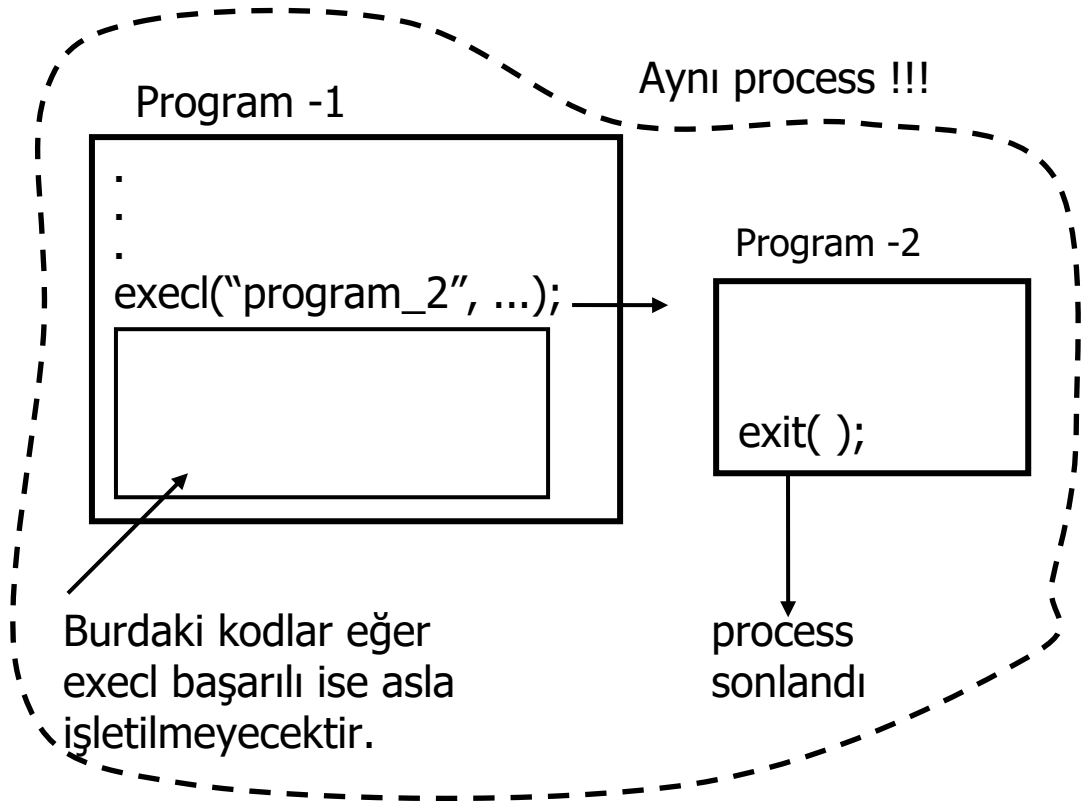
Örnek:

```
execl("new_program", "new_program", 0 );
```

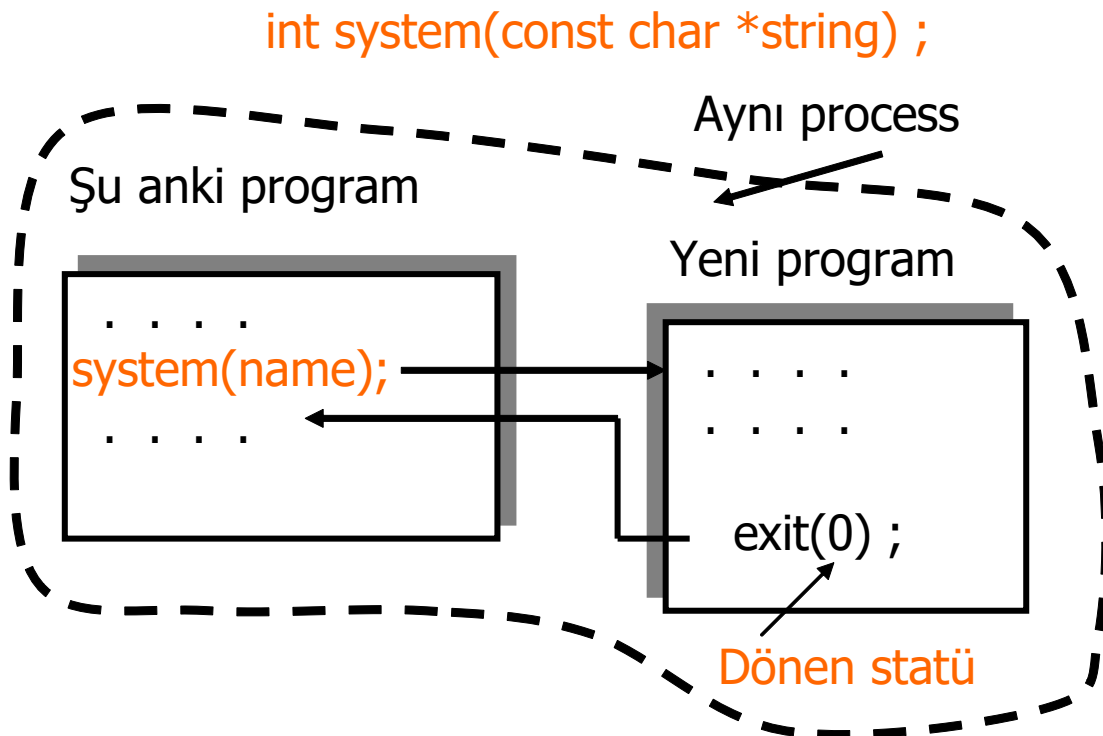
Annotations for the example code:

- yol** (path) points to "new\_program".
- arg0** points to "new\_program".
- NULL** points to 0.
- paramtre listesinin sonu** (end of parameter list) points to 0.

## Başka bir programa eski programa dönmeden geçiş



## Başka bir programa eski programa dönecek bir geçiş



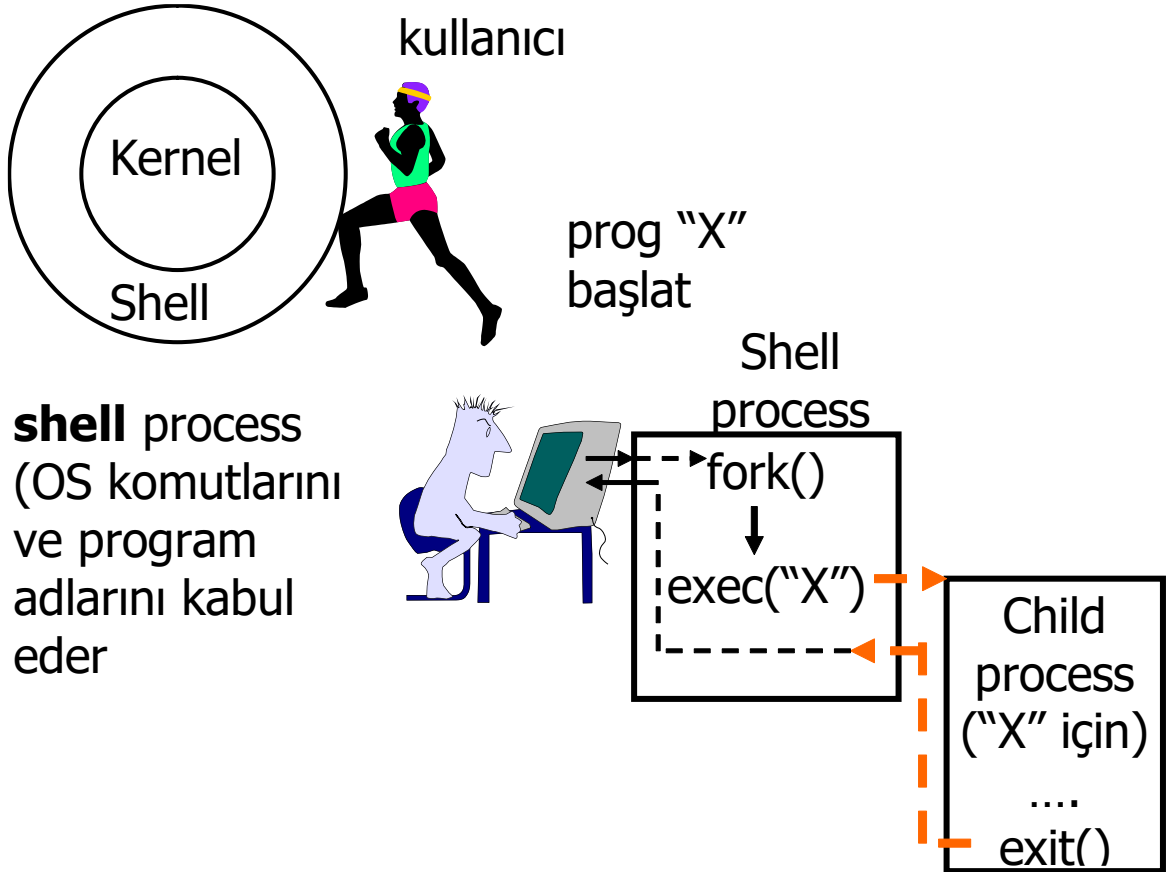
### Örnek:

```
main()
{
    int status ;
    . . . .

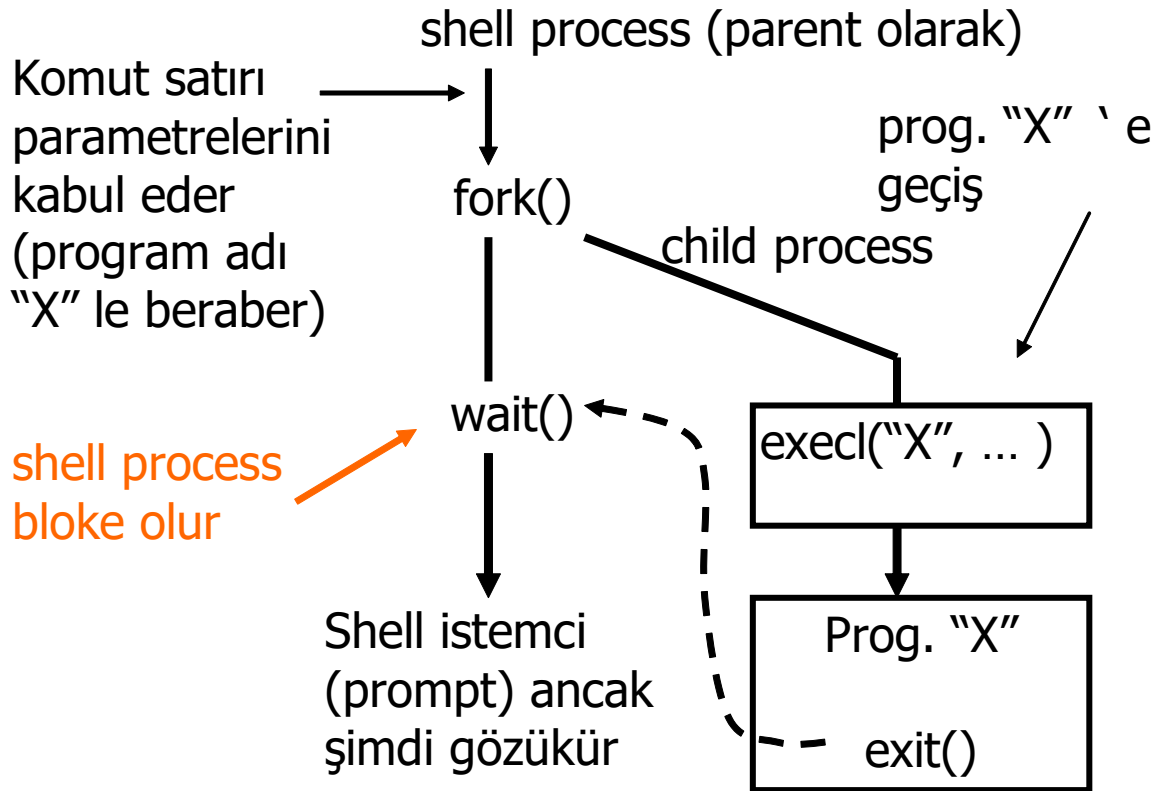
    status = system( "new_prog par1 par2" ) ;

    //status dönen değeri analizi...
    . . . .
}
```

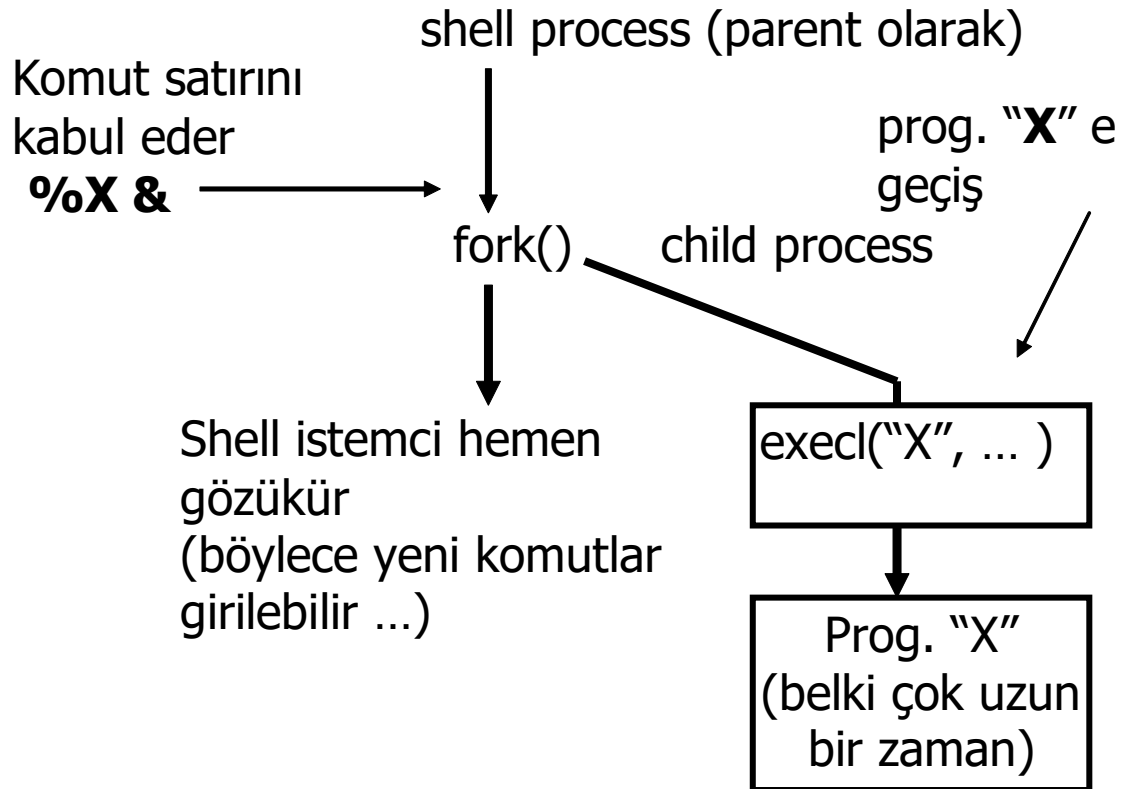
## Kullanıcı programları kabuk (shell) process ' e ait child işlemlerce işletilir



## Shell process (önelan işlemlerinin başlatılması)

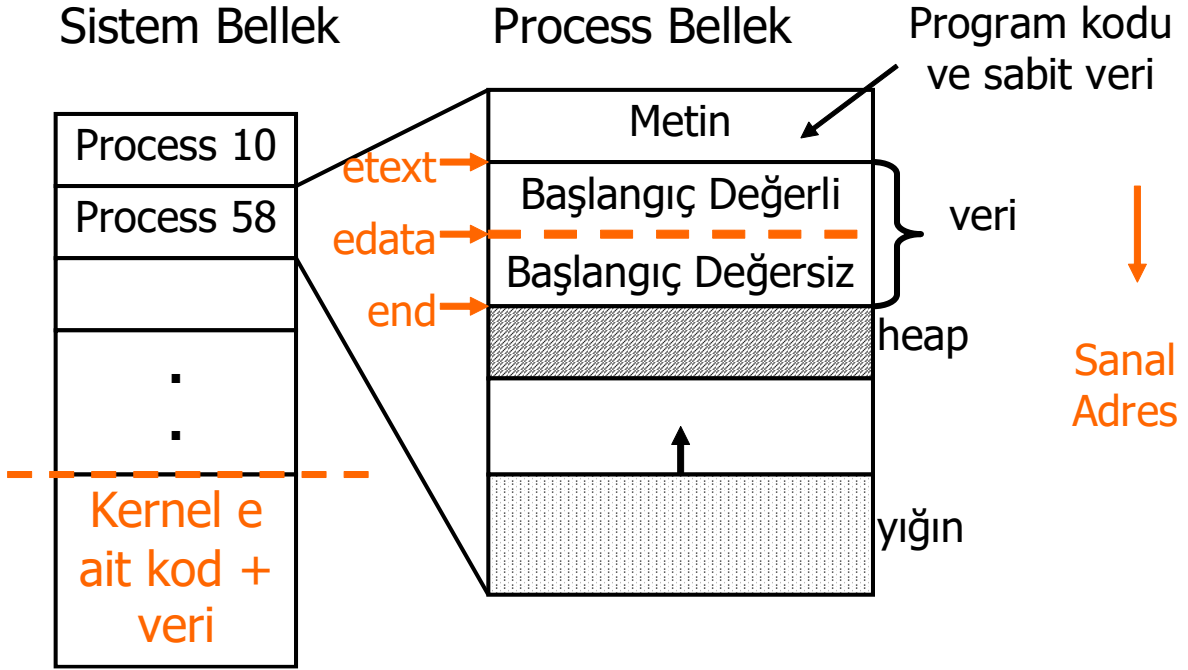


## Shell process (arkaalan işlemlerinin başlatılması)





## Sistem ve Process Belleği (sanal!)



Sanal Bellek : prosese özgü adres uzayının (veri+ heap+yığın segmentleri) tüm fiziksel belleği sanal olarak adreslemesi + düzgün bir sırayla sanal adres alması + (opsiyonel) belleğin disk ile eşlenmesi

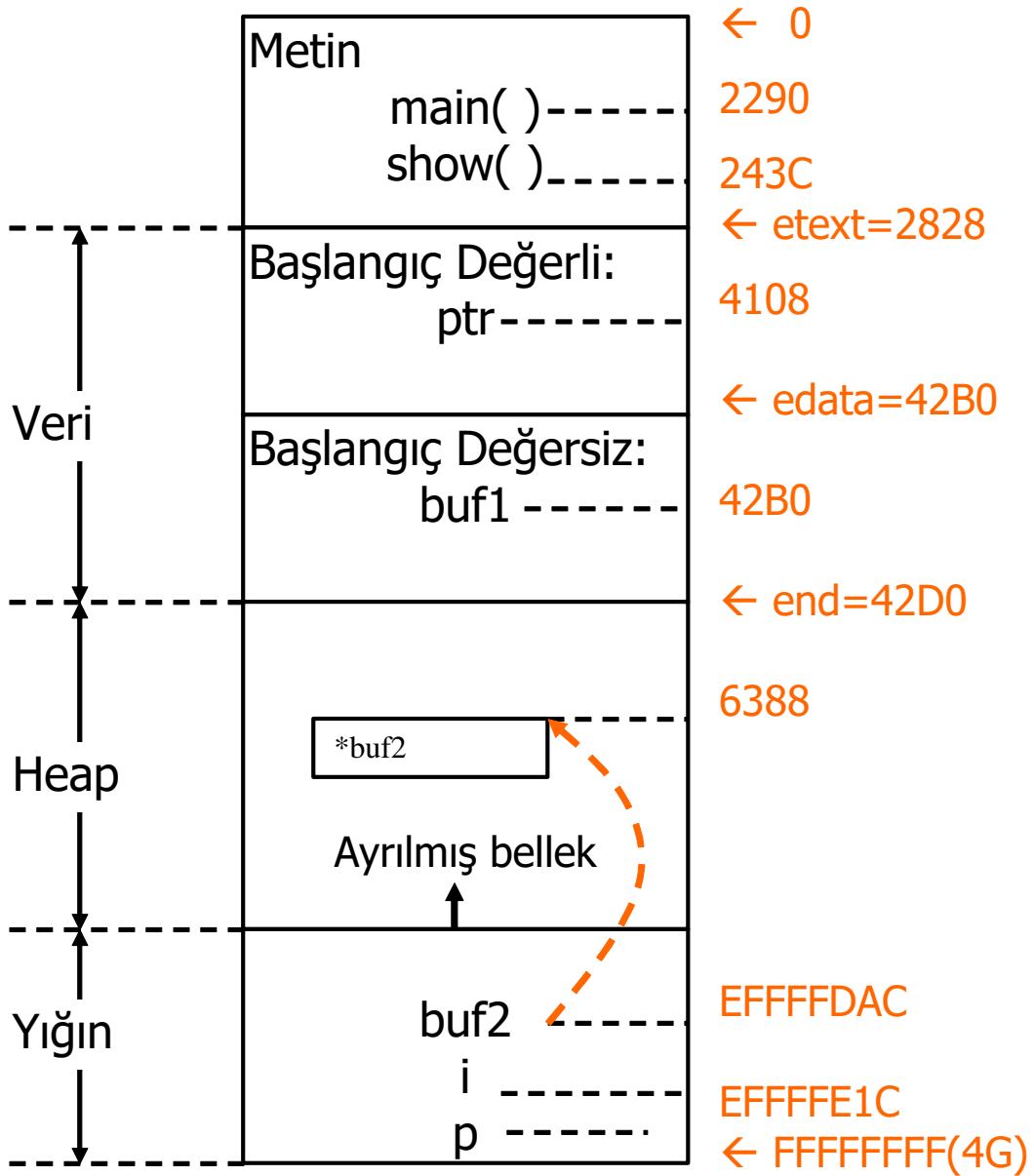
fork() çağrısının hemen sonrasında çocuk prosesin sanal belleği, parent prosesin sanal belleği olarak kopyalanır (forktan dönen değer dışında belleklerin imajı aynıdır)

exec() yada system ile başka bir programa geçiş yapan prosesin sanal bellek imajı tamamen değişir (prosesin kimlik bilgisi değişmez)

## Örnek:

```
#include . . .
char *ptr = "Hello\n"; /*static*/ ← başlangıç değerli alan
char buf1[25];          ← başlangıç değersiz alan
main()
{
    int show(char *) ;
    int i = 0 ;          /* otomatik */ ← yığın
    . . .
    for ( ; i < 3 ; ++i ) show( ptr ) ;
}
↓
int show( char *p){
    char *buf2 ; ← yığından kullanacak
    buf2 = malloc( (unsigned)(strlen(p)+1)) ; ← heap
    strcpy(buf2, p) ;
    printf("%s", buf2) ;
    free( buf2 ) ; ← heap alanına iade edilecek
    exit(0) ;
}
```

## Program için Sanal Bellek



## Dosya işlemleri için ana system çağrıları

**Not: #include <fcntl.h> ekleyin**

Sis. Çağrısı	Amacı	Dönme değeri
open	Dosyaları okumak yada yazmak için açar	Dosya tanımlayıcısı (int) yada -1
creat	Yeni boş bir dosya yaratır	Dosya tanımlayıcısı (int) yada -1
read	Açık dosyadan buffer .	Gerçekte okunan karakter sayısı veya -1
write	Buffer dan dosyaya yazar	Gerçekte yazılan karakter sayısı veya -1
close	Açık dosyayı kapatır.	0 veya -1
unlink	Dosyayı siler	0 veya -1

## **open()** kullanarak dosya yaratma

```
fd = open("myfile", O_WRONLY|O_CREAT|O_APPEND, 0644);
```

### İşlem:

1. Eger "myfile" dosyasi varsa, append modunda acilir
2. Yoksa 0644 haklariyla yaratilir.

## **Bir Dosyayı Blok Olarak Kopyalama**

```
#include <fcntl.h>
```

```
char *source = "myfile";      /* çalışma dizininde */  
char *dest = "/tmp/copy" ;
```

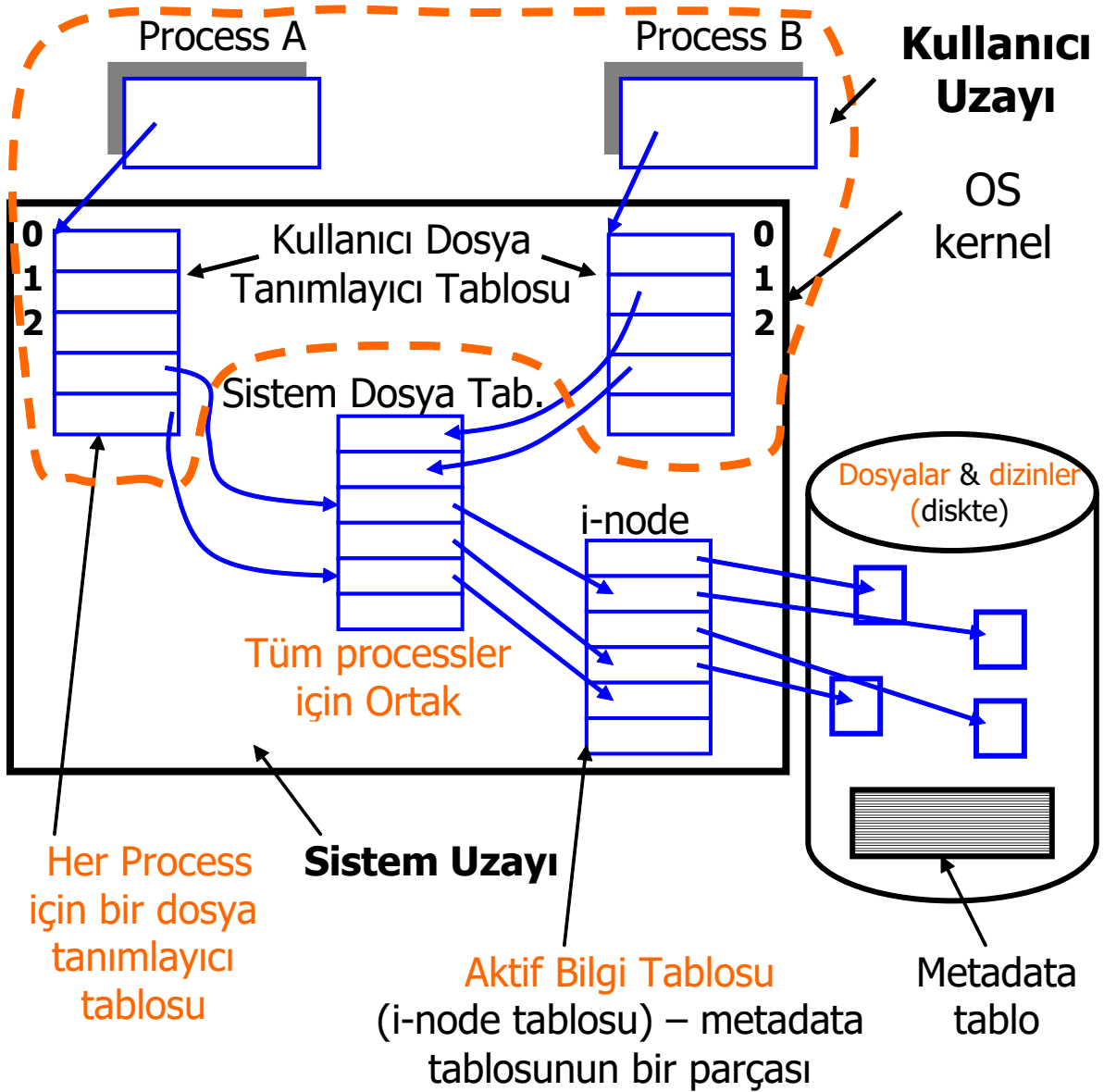
```
main()  
{  
    int in, out, nread ;  
    char buffer[100] ;  
    in = open(source, O_RDONLY) ;  
    out = creat(dest, 0644) ;
```

```
/* kopyalama */
```

```
while ( (nread = read(in, buffer, 100)) > 0 )  
    { write(out, buffer, nread) ; }
```

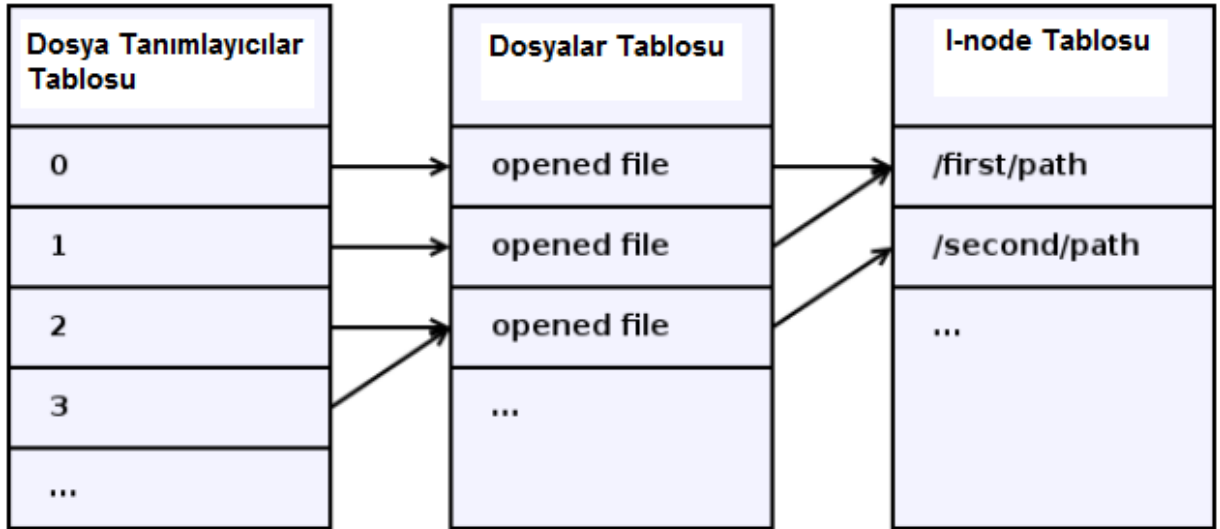
```
close(in) ;  
close(out);  
}
```

## UNIX te Processler ve Dosyalar

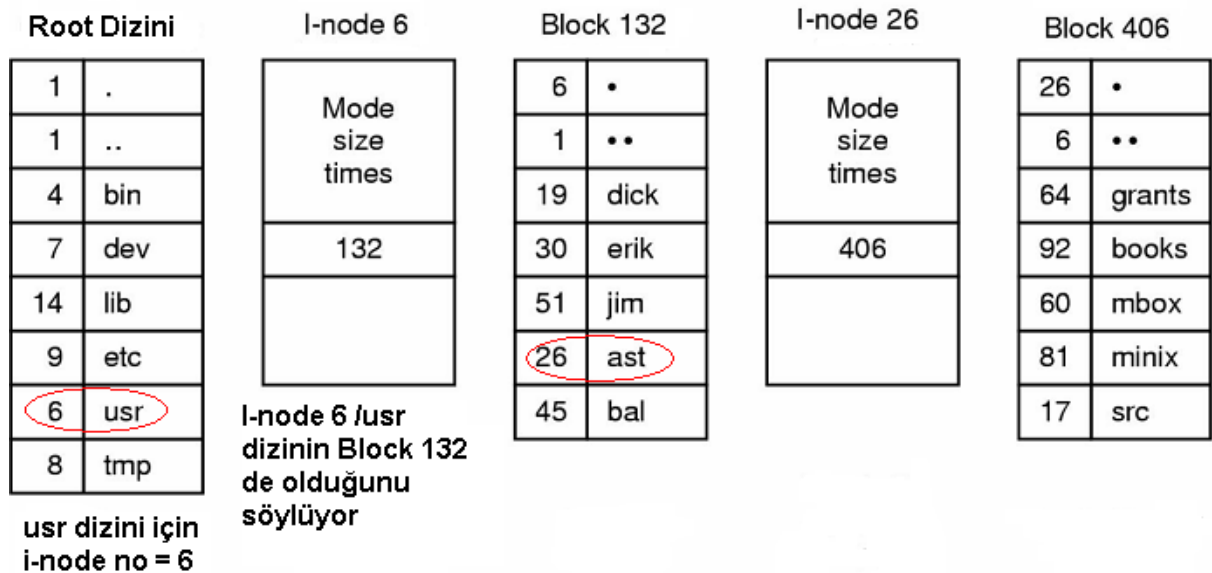


**Dosya Tanımlayıcıları, Dosya Tanımlayıcı tablosuna** sadece bir indekstir ve her process için ayrıdır. Sadece 0-2 arasındaki tanımlayıcılar tüm prosesler için ortak reserve edilmistir. **Dosya Tanımlayıcı Tablosundaki** her bir girdi ise **Sistem Dosya Tablosundaki** dosya nesnesine bir referanstır. Bu dosya nesnelerinin her biri, **i-node tablosunda** karşılık gelen i-node nesnesine bir referanstır. I-node, bir dosya sistemi nesnesidir ve ilgili girdinin disk adresini, haklarını ve diğer tanımlayıcı bilgileri (metadata) içerir. **Her bir dosyanın(dizin) tek bir i-node numarası vardır.**

**Dosyalar Tablosu**, sistem içinde açık dosyaları, bunların okuma\yazma offset durumlarını ve diğer kalıcı bilgilerini içerir. Dosyalar tablosu ve i-node tablosu, tüm prosesler için ortaktır.



## Unixte Dizin Geçişleri : Örnek



## ***open()*** system çağrısı için örnekler

```
fd1 = open("data", O_RDONLY) ;  
fd2 = open("info", O_RDWR) ;  
fd3 = open("data", O_WRONLY) ;
```

} Kul. process A

### Sonuç:

fd1 = 3, fd2 = 4 , fd3 = 5 (process A için)

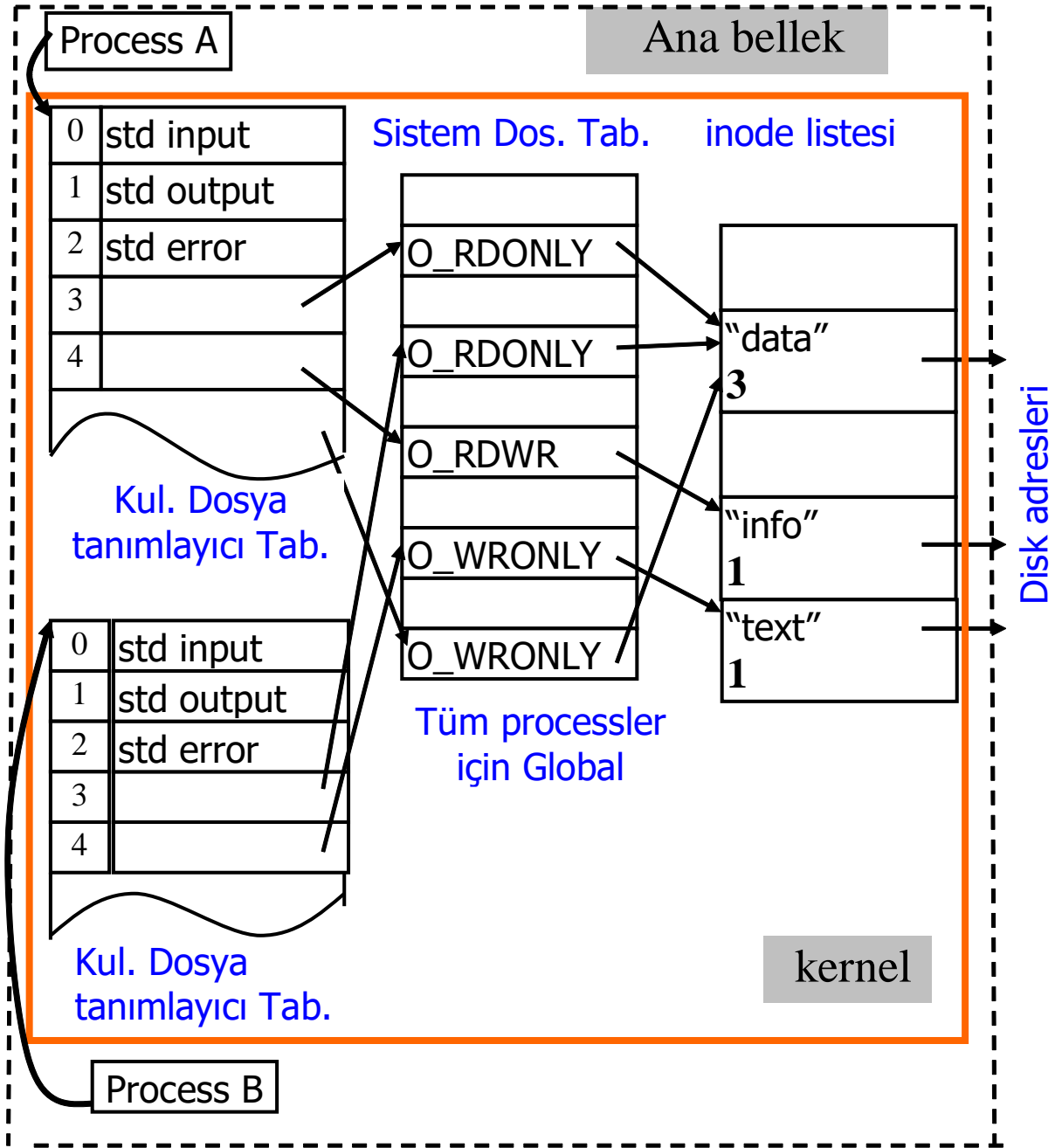
```
fd1 = open("text", O_RDONLY) ;  
fd2 = open("data", O_WRONLY) ;
```

} Kul. process B

### Sonuç:

fd1 = 3 , fd2 = 4 (process B için)

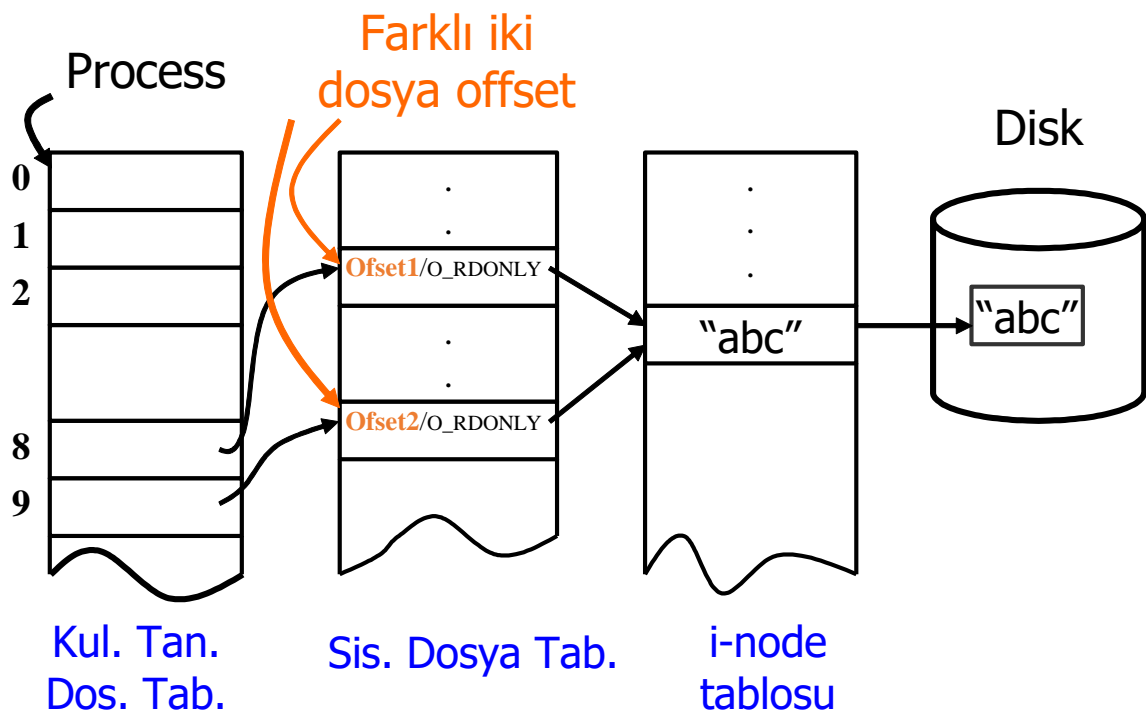




## İki dosya tanımlayıcısı ile bir dosyayı okuma

```
#include <fcntl.h>
main()
{ int fd1, fd2 ;
  char buf1[512], buf2[512] ;

  . . . . .
  fd1 = open("abc", O_RDONLY ) ;
  fd2 = open("abc", O_RDONLY ) ;
  read( fd1, buf1, 512 ) ;
  read( fd2, buf2, 512 ) ;
  . . . . .
}
```

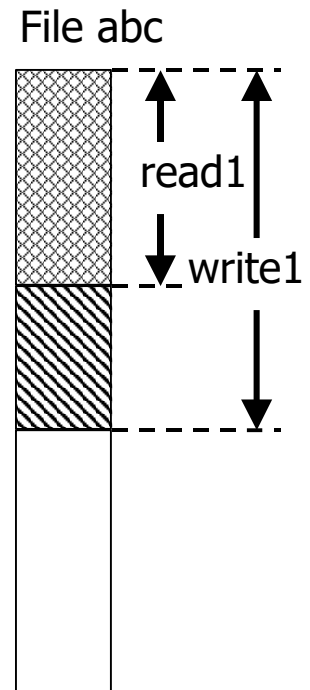


## İki processin bir dosyadan okuması ve yazması

```
#include <fcntl.h>
main()                /* process A */
{ int fd ;
  char buf[300] ;
  fd = open("abc", O_RDONLY) ;
  read( fd, buf, 300 ) ;    /* 1 */
  read( fd, buf, 300 ) ;    /* 2 */
}
```

```
#include <fcntl.h>
main()                /* process B */
{ int fd, i ;
  char buf[512] ;

  for ( i = 0 ; i < 512 ; i++ )
    buf[i] = 'C' ;
  fd = open("abc", O_WRONLY) ;
  write( fd, buf, 512 ) ; /* 1 */
  write( fd, buf, 512 ) ; /* 2 */
}
```



### Bazı mümkün read ve write sırası:

```
read1, read2, write1, write2 ; }
read1, write1, read2, write2 ; }
. . . }
```

## **Bir dizindeki dosya isimlerini listeleme**

```
#include <types.h>
#include <dir.h>
#include <stdio.h>

main( )
{
    DIR      *dp ;
    struct    direct    dir ;          /* veya dirent */
    if ( ( dp = opendir("<dir-adı>") ) == NULL )
    {
        printf("Hata: dizini açamadı.\n") ;
        exit(1) ;
    }

    while ( (dir = readdir(dp) ) != NULL )
    {
        /*d_ino=0 ise girdi silinmiş*/
        if (dir->d_ino == 0 ) continue ;
        printf("%s\n", dir->d_name) ;
    }

    closedir(dp) ;
    exit(0) ;
}
```

### **Bir Dizin Girdisi:**

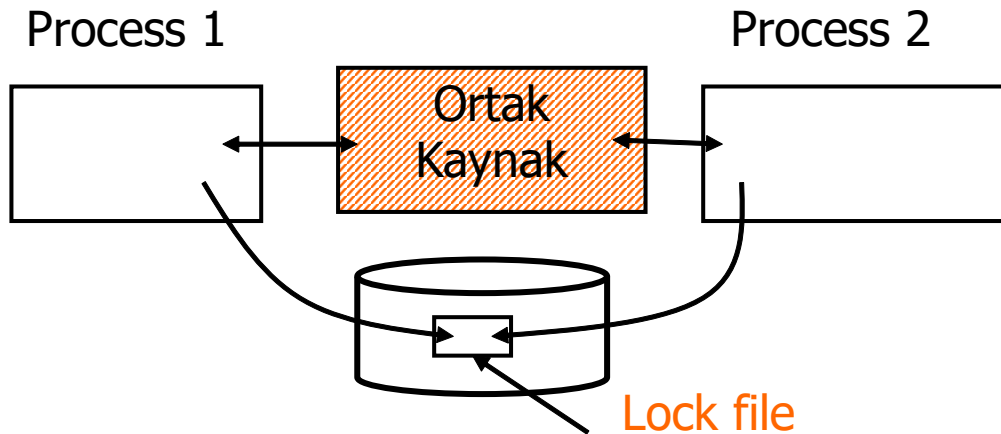
```
struct direct {
    ino_t      d_ino ; \\i-node no
    char       d_name[ ] ; \\girdi adi
};
```

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/dir.h>

main( argc, argv )
int argc ; char *argv[] ;
{
    DIR *dp ;
    struct direct *dir ;
    struct stat info ;
    if ( stat(argv[1], &info) == -1 )          /* stat() kullanımı*/
        printf("\n Stat  %s için başarısız\n", argv[1] ) ;
    else
    {
        printf("\nStat for %s is OK.\n", argv[1]) ;
        if ( info.st_mode & S_IFREG ) /* Sıradan bir dosya mı ? */
        {
            printf("%s normal bir dosya \n", argv[1]);
            printf("Dosya büyüklüğü : %d\n", info.st_size);
        }
        else
        if ( info.st_mode & S_IFDIR ) /* Yoksa dizin mi? */
        {
            printf("%s bir dizin\n", argv[1]);
            printf("Dizin büyüklüğü : %d\n", info.st_size);
            if ( (dp=opendir(argv[1])) == NULL) /*Dizini açmaya
                                                çalış */
            {
                printf("Dizin %s açılmadı\n", argv[1]);
                exit(1) ;
            }
        }
    }
}
```

```
while ( (dir=readdir(dp)) != NULL ) /* dizindeki bir
                                     sonraki dosya ismini Okumaya çalış */
{
    if ( dir->d_ino == 0 ) continue ; /* Silinmiş dosyaları
                                     atla */
    if ( stat(dir->d_name, &info) == -1 ) /* bu dosya
                                     için stat kullanımı */
    { printf("Stat %s için başarısız\n", dir->d_name);
      perror(dir->d_name);
      continue ;
    }
    else
    { printf("%-20s Dosya büyüklüğü: %d bytes \n",
            dir->d_name, info.st_size);
    }
}
closedir(dp) ;
}
}
return(0) ;
}
```

## İki yada daha fazla process ' in lock file (dosya kilitleme) kullanması



```
#include <errno.h>
#include <fcntl.h>
    /* diğer include ' lar */
#define LOCKF  "/tmp/abc"
```

```
main()
{ int fd ;
```

```
while ( (fd = creat( LOCKF, 0 )) == -1 ) sleep(1) ;
close( fd ) ;
```

*Hiçbir hak yok*

```
/* ortak kaynağı kullan */
```

```
unlink( LOCKF ) ;  /* dosyayı sil */
}
```

## Kütüphane çağrısı ***lockf()*** ` dosya kilitleme için kullanımı

**int lockf(int fd, int cmd, long size) ;**

Dosya tanım. →  
Fonksiyon: →  
Byte sayısı →

F\_LOCK  
F\_ULOCK  
...

```
#include <fcntl.h>
```

```
.....
```

```
main() /*dosya mevcut ve yazma modunda acilmali */  
{ int fd ; int i ;
```

```
    fd = open("abc", O_RDWR) ;  
                                     /* tüm dosyayı kilitle */
```

```
    i = lockf( fd, F_LOCK, 0 ) ;  
    if ( i == -1 ) { perror( ... ) ; exit( 1 ) ; }
```

```
        /* kilitli dosyayı kullan */
```

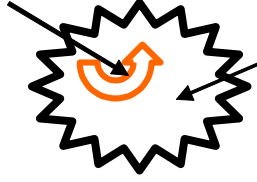
```
    lockf( fd, F_ULOCK, 0 ) ;      /* kilidi kaldır */  
}
```



## İş-parçacıkları (Thread ler)

Bir process = Tek thread li process

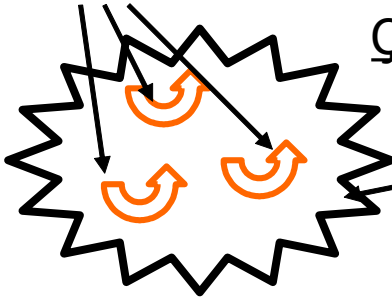
A thread



Çalışma Ortamı:

- Adres Uzayı (veri+kod)
- Semaforlar
- Ports, v.b.

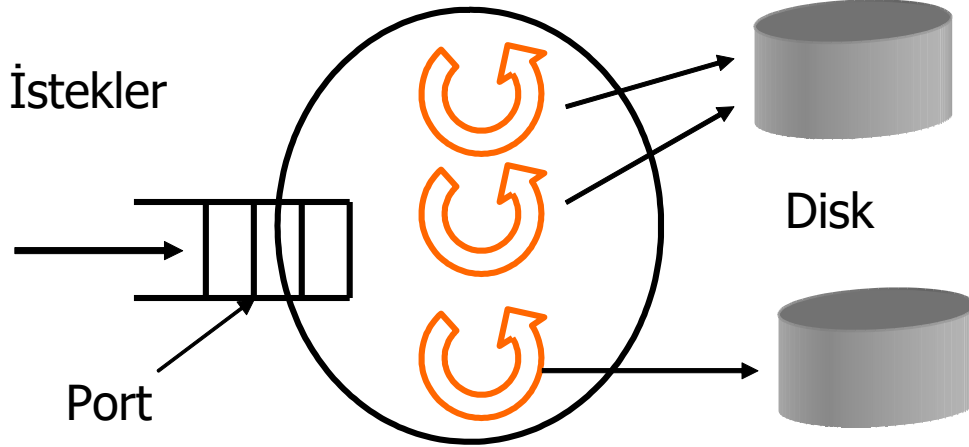
Threads



Çok-threadli Process

Çalışma ortamı

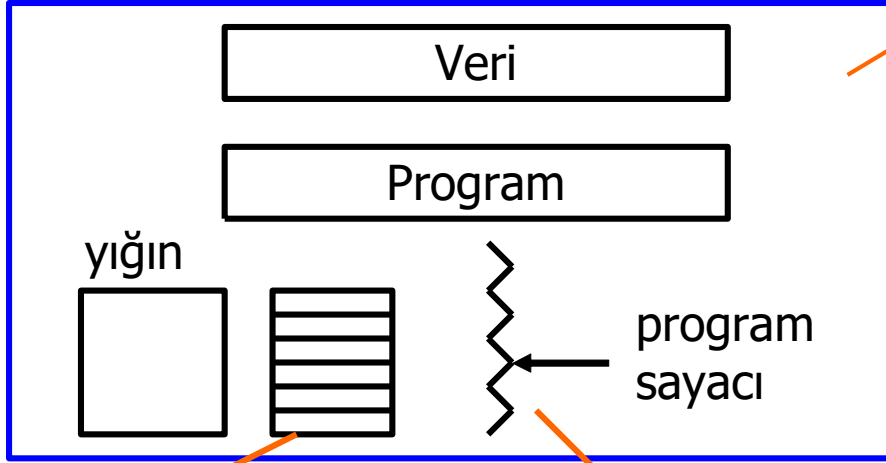
Çok-thread li bir sunucu örneği



Proces ve Thread lerin Kıyaslanması		
Özellik	Proces	Thread
Yaratma zamanı (in UNIX)	$\cong 10$ ms	$\cong 1$ ms
Anahtarlama zamanı	$\cong 1.8$ ms	$\cong 0.4$ ms
İletişim Mekanizması	Komplex	Basit
Veri paylaşımı	Hayır	Evet
Konum	Aynı yada farklı bilgisayar içinde	Aynı bilgisayarda

## Threads (İşparçacıkları)

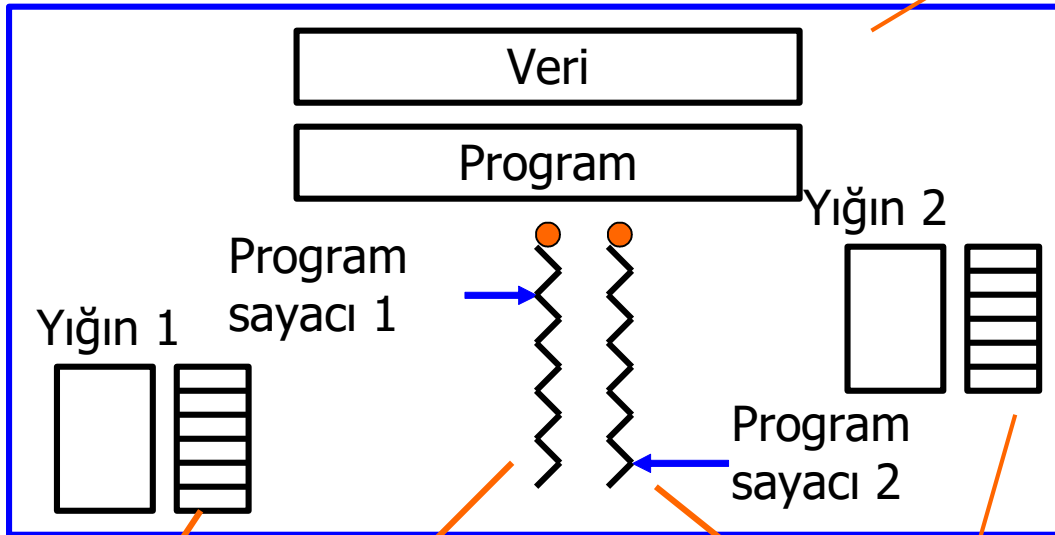
Tek thread li bir process



sayaçlar

Process'in thread çalışması

İki threadli bir process  
(çok-thread li process)



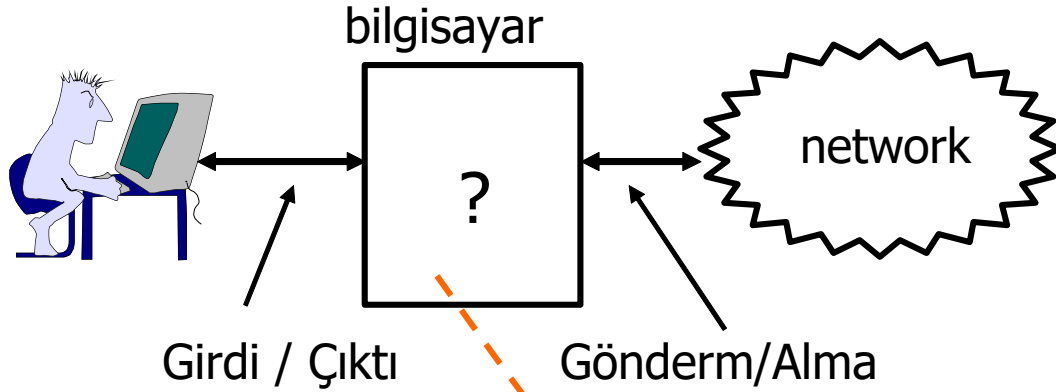
Sayaçlar 1

Thread 1

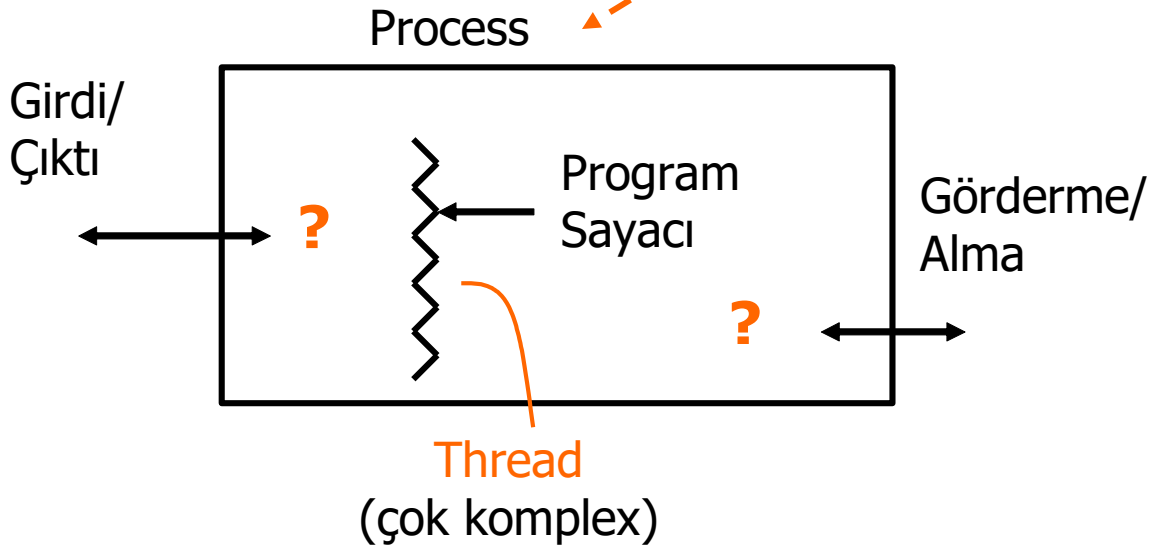
Thread 2

Sayaçlar 2

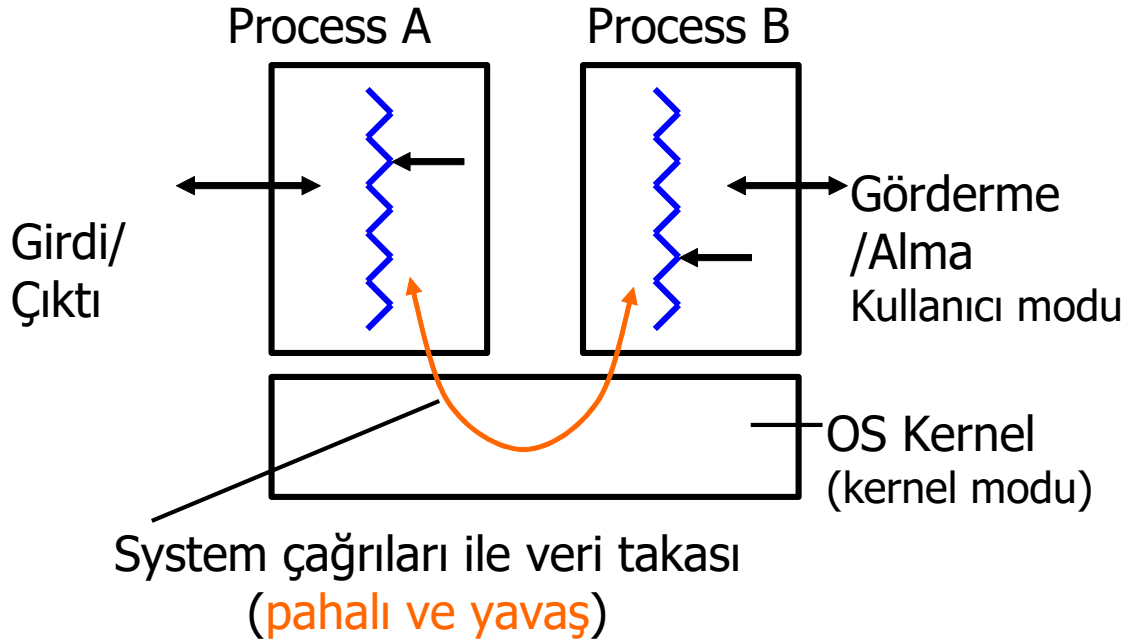
## Etkileşimli bir sistemin kurulumu için 3 yaklaşım



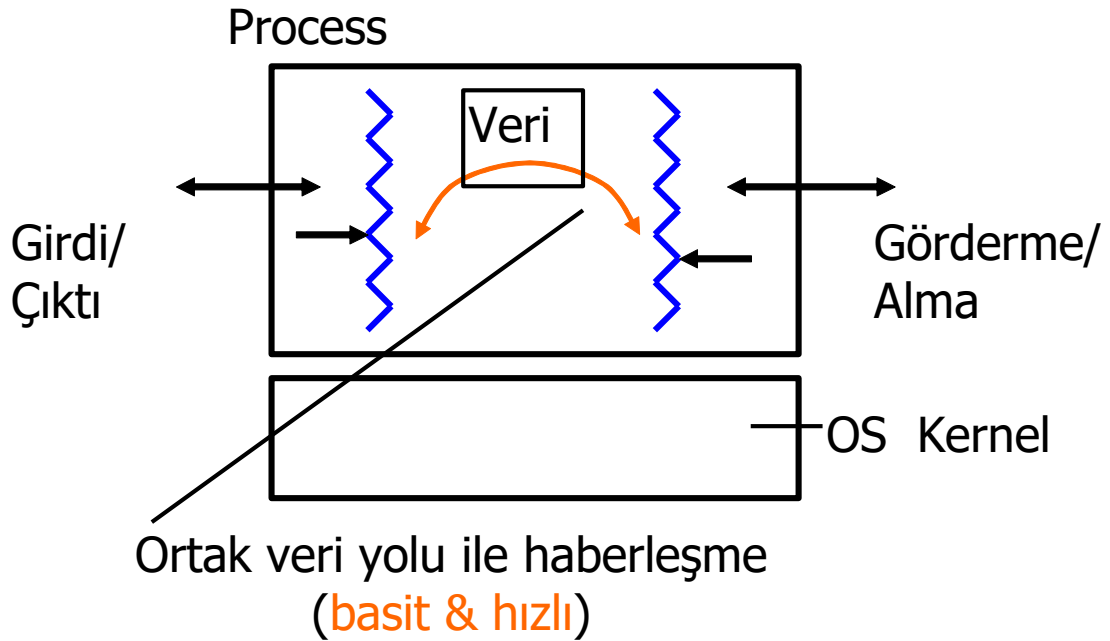
### 1. Bir thread li bir process



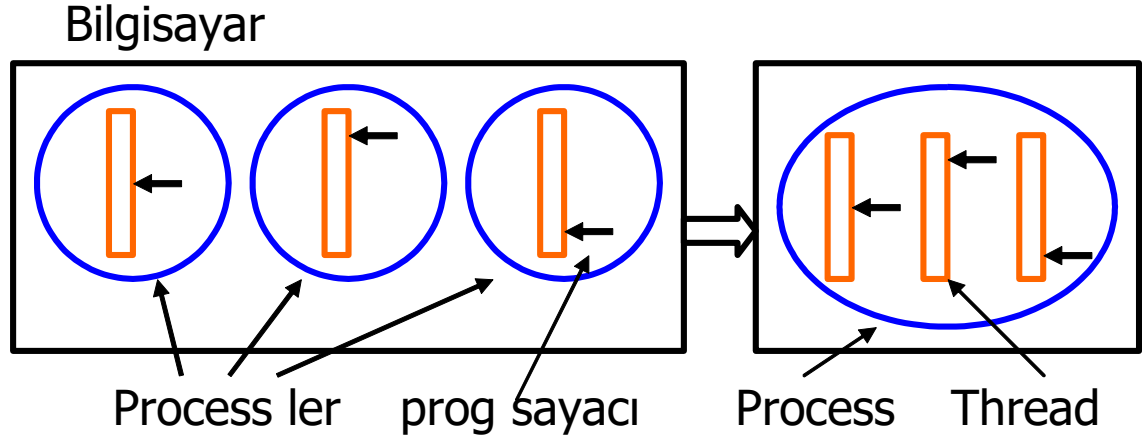
## 2. İki process (herbiri bir threadli)



## 3. İki thread li bir process



## Bir process ten bir threade



### **Her Process için**

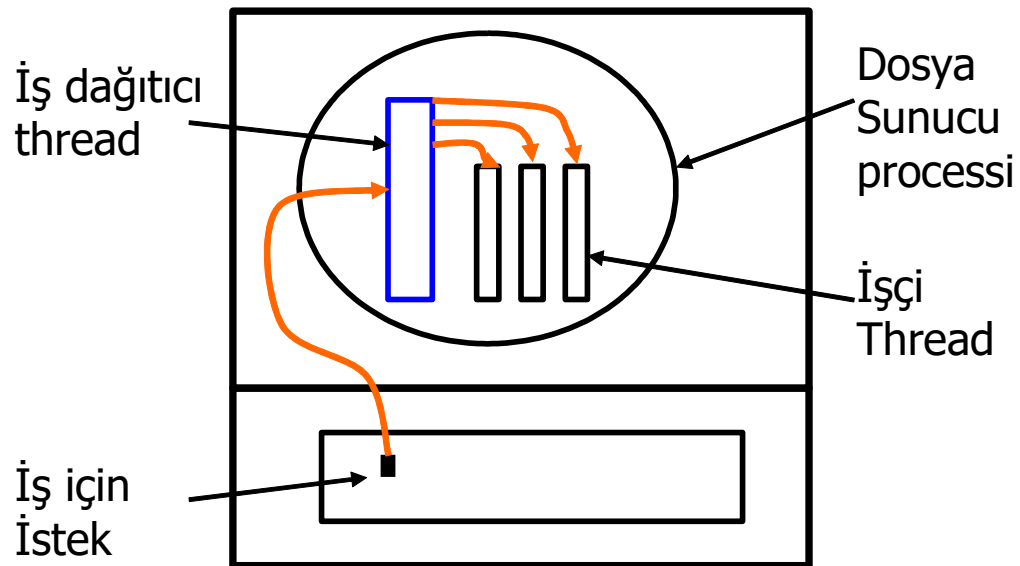
Adres Uzayı  
Açık Dosyalar  
Çocuk process ler  
Semaforlar  
Sinyaller  
. . . .

### **Her Thread için**

Prog sayacı  
Sayac takımı  
Yığın  
Durum  
Çocuk thread ler

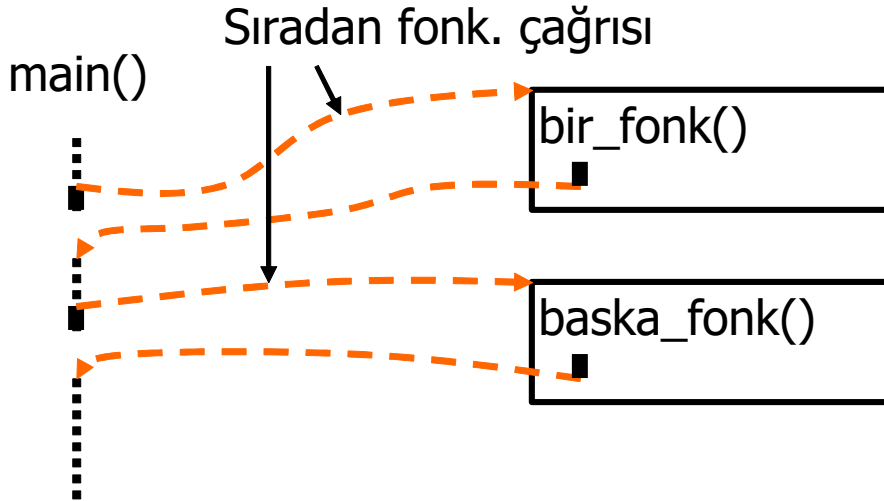
Ayrı bir adres uzayı YOK!!!

## Bir sunucunun thread lerle muhtemel bir oluşumu

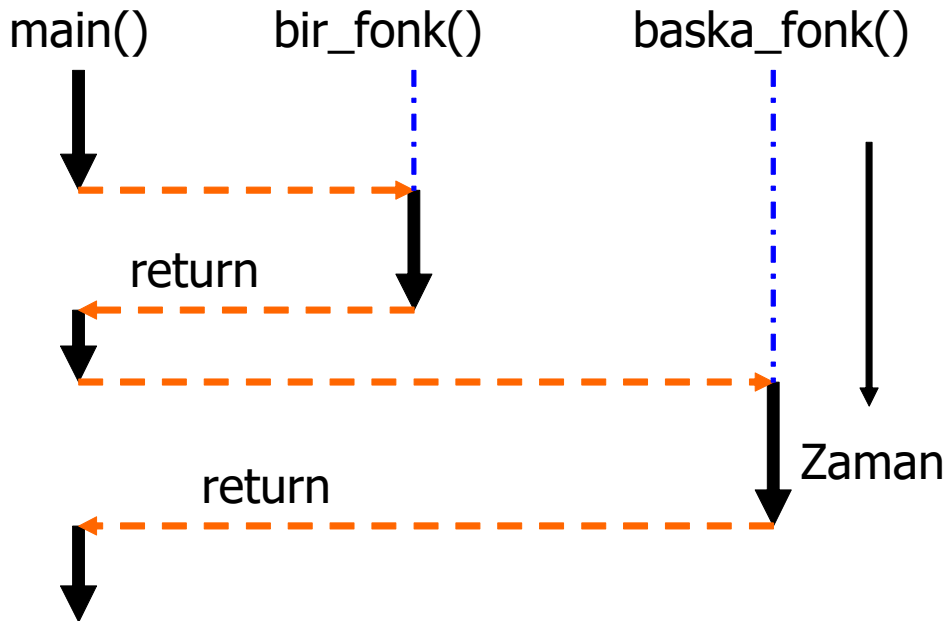


## Bir thread li bir process

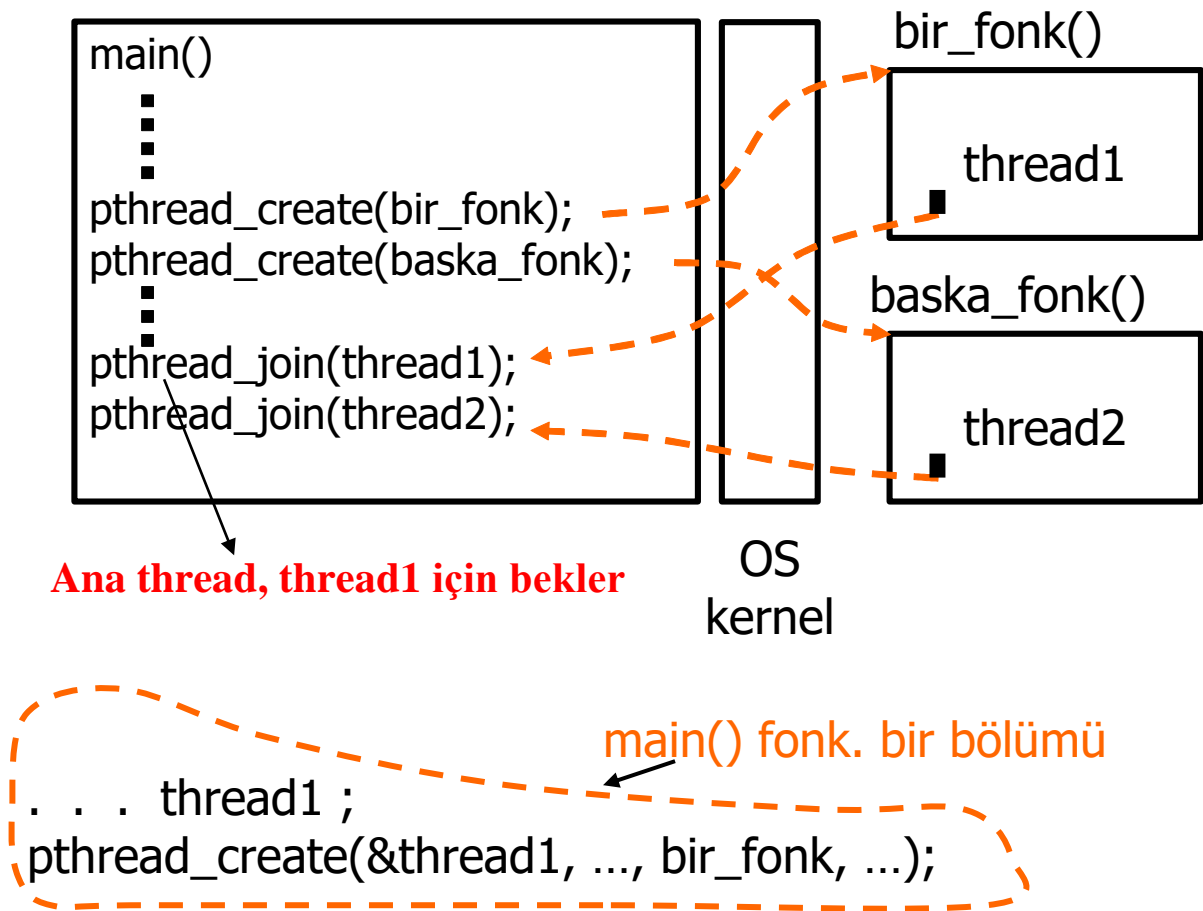
Sıralı olarak iki fonksiyonu çağırır (**aynı programda**)



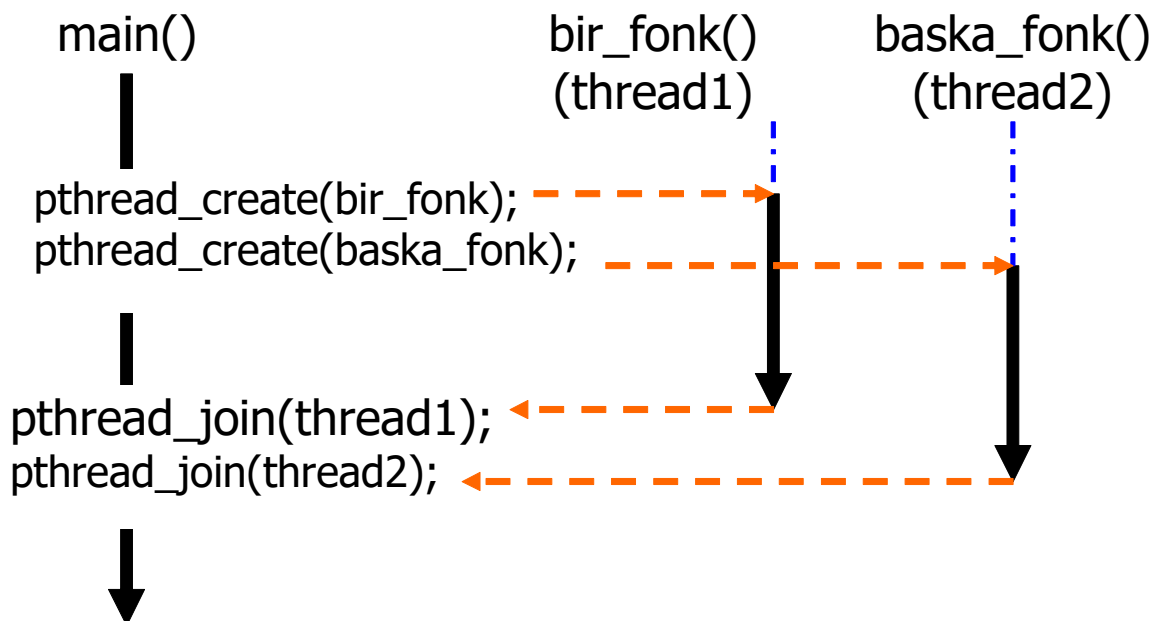
Karşılık gelen zaman diagramı:



## İki threadli bir process (Linux OS de)



## Karşılık gelen zaman diagramı



## `pthread_create(pthread, pattr, pfunc, pargs ),`

- tüm parametreler referansla çağırım şeklinde (**void \***)
- **pthread** yaratılan thread (**pthread\_t \***)
- **pattr** yaratılan threadin özellikleri (**NULL varsayılan değer için**)
- **pfunc** threadin işleteceği fonksiyon, her zaman **void \*** dönmeli
- **pargs** thread fonksiyonun parametresi (**void \***), tek parametre, çoklu parametre için struct tipinde bir adresi **void \*** olarak verin

## `pthread_join(pthread_t * pthread, void ** retval)`

- **pthread** 'in çıkış yapmasını bekler,
- thread **return (statu)** veya **pthread\_exit(statu)** ile çıkış yaptığında, thread'in statu değerini **retval**' in gösterdiği yere kopyalar (**pthread\_join fonk. içinde \*retval=statu yapılır**).

## Threadli bir Win/NT Programı

"include" & "define" yönergeleri  
değişken tanımları

```
unsigned __stdcall Thread_Adı(parametreler)
{
...Thread in iş alanı
return 0 ;
}
```

Thread  
Fonk.

```
main(...) {
. . . .
hThread= _beginthreadex(..., Thread_Adı, parametreler):
```

```
    // Ana iş tamamlandı . . .
```

```
    WaitForSingleObject(hThread, ... ) ;
    CloseHandle(hThread);
```

```
    return 0 ;
}
```



## Örnek: 2 Threadli basit bir program (thread fonksiyonu parametre almıyor)

```
#include<stdlib.h>
#include<stdio.h>
#include<pthread.h>
/* Burada tanımlanan her veri her thread tarafından
görülür*/

void * fonk1(void *); /* thread için fonk. tanımı */

int main(void)
{
pthread_t td1;
int p, j;
printf("Process main thread olarak başlar...\n");
p = pthread_create ( &td1, NULL, fonk1, NULL);
if (p != 0) { perror ("Thread problem"); exit(1); }
for(j = 1; j <= 4; ++j) printf("Ana işlem:%d\n", j);
pthread_join(td1, NULL); /*çocuk thread için bekle*/
printf("Ana thread sonlanır\n"); //NULL : td1 dönüş statüsünü alma
exit(0);
}

void * fonk1(void *)
{ int i;
  for(i = 1; i <= 3; i++)
    printf("çocuk thread:%d\n", i);
  return NULL;
}
```

option  
↙

%cc -o threads threads.c -lpthread

## Örnek: verilen iki sayıyı toplama için çok parametrelili thread çağırımları : yaklaşım-1 (tüm parametreleri bir struct ile paketle)

```
/* örnekte thread verilen iki sayıyı toplansın */
typedef struct arg {
    int a;    //in
    int b;    //in
    int sum;  //out
} argt;

void * threadfun (void * arg){//her zaman void * olmalı
    /* cast işlemi yap */
    argt *args = (argt *) arg;

    /* sonucu heap te yaratılmış olan struct içine yaz */
    args->sum = args->a + args->b;
    pthread_exit(NULL); //veya return NULL
}

void main () {
    pthread_t thid1;
    argt * args = (argt *) malloc(sizeof(argt));
    args->a=2; args->b=3;
    pthread_create(&thid1, NULL, threadfun, args);
    ...
}
```

## Örnek: çok parametrelili fonksiyon çağırımları : yaklaşım-2 (girdileri ayrı, çıktıları ayrı paketle)

```
/* örnekte thread verilen iki sayıyı toplansın */
typedef struct arg {
    int a;    //in
    int b;    //in
} argt;
```

```
void * threadfun (void * arg) { //her zaman void * olmalı

/*çıktı param için heapte yer aç */
    int * sum = (int *) malloc(sizeof(int)); //out

    /* girdi parametreleri için cast işlemi yap */
    argt *args = (argt *) args;

    /* sonucu heap te yaratılmış olan alana yaz */
    *sum = args->a + args->b;
    return (void *) sum; //her zaman void * dönmeli
}

void main () {
    pthread_t thid1;
    argt * args = (argt *) malloc(sizeof(argt));
    int *toplam; // çıktı
    args->a=2; args->b=3; // girdiler

    pthread_create(&thid1, NULL, threadfun, args);
    pthread_join(thid1,&toplam);//sonuç, verilen adrese
yazıldı, arkaalanda *retval=status işlemi ile
*(&toplam)=sum ve sonuc olarak toplam=sum oldu.
    printf("thread dönüş değeri:%d", *toplam)
    ...
}
```

### Alternatif:

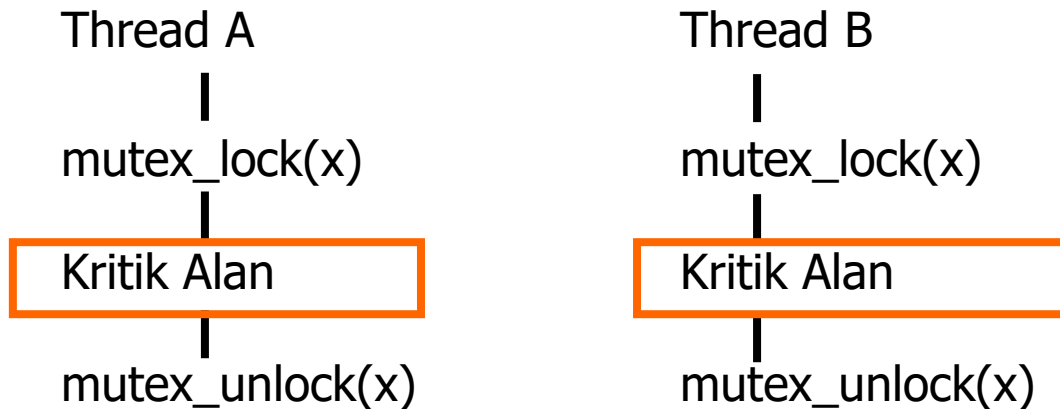
```
void * threadfun (void * arg) {
    int sum;
    . . .
    sum = args->a + args->b;
    return (void *) sum; //her zaman void * dönmeli
}

void main () {
    int sum;
    ...
    pthread_join(thid1, (void **)&sum);
    printf("thread dönüş değeri:%d", sum)
}
```

## Senkronizasyon için thread çağrıları

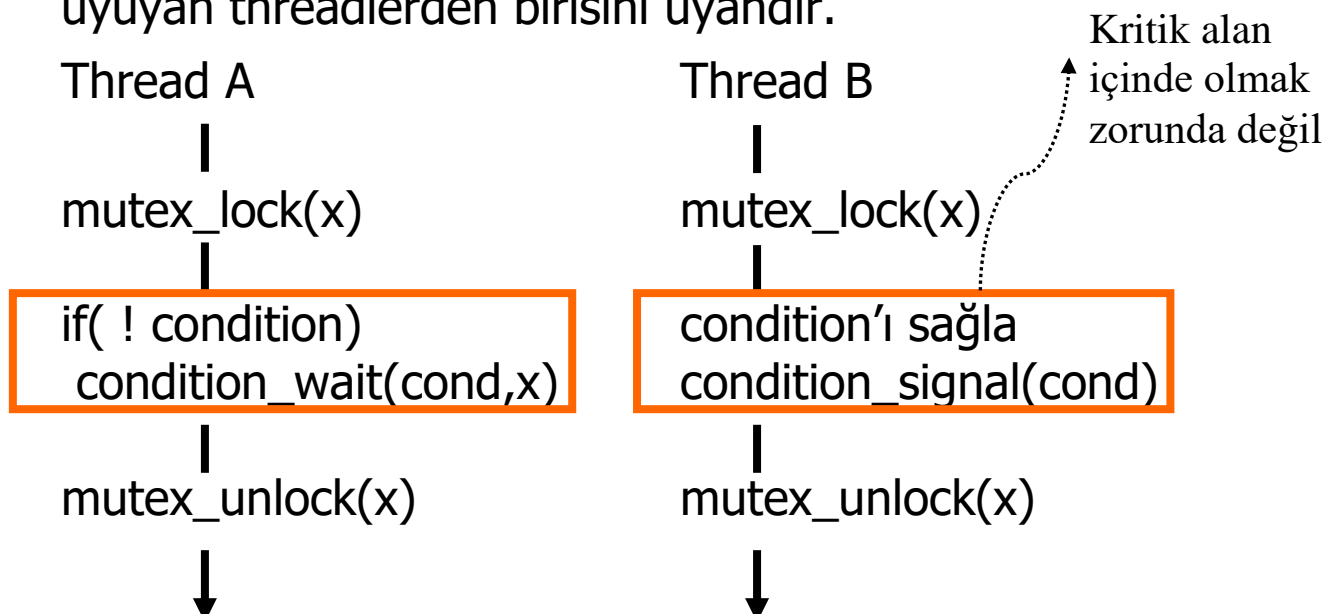
**mutex\_lock(mutexId)** : mutexId kilitli değilse kilitle, aksi halde uyu (bloke ol)

**mutex\_unlock(mutexId)** : mutexId üzerinde uyuyan threadlerden sadece birisini uyandır (OS birtane seçer)



**condition\_wait(conditionId, mutexId):** **Kilidi aç ve uyu** / **uyan ve kilitle**: mutexId kilidini aç(uyuyan birisini uyandır) ve conditionId sinyali alıncaya kadar uyu, uyanınca yeniden mutexId'yi kilitle.

**condition\_signal(conditionId):** conditionId üzerinde uyuyan threadlerden birisini uyandır.



## Örnek : Thread ve Mutex kullanımı

```
#include<stdio.h>
#include<string.h>
#include<pthread.h>
#include<stdlib.h>
#include<unistd.h>
pthread_t tid[2];
int counter;
pthread_mutex_t lock;
void* doSomething(void *arg){
    pthread_mutex_lock(&lock);
    unsigned long i = 0;
    counter += 1;
    printf("\n Job %d started\n", counter);
    for(i=0; i<(0xFFFFFFFF);i++);
    printf("\n Job %d finished\n", counter);
    pthread_mutex_unlock(&lock);
    return NULL;
}

int main(void){
    int i = 0;
    int err;
    if (pthread_mutex_init(&lock, NULL) != 0) {
        printf("\n mutex init failed\n");
        return 1;
    }
    while(i < 2) {
        err = pthread_create(&(tid[i]), NULL, &doSomething, NULL);
        if (err != 0)
            printf("\ncan't create thread :[%s]", strerror(err));
        i++;
    }
    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
    pthread_mutex_destroy(&lock);
    return 0;
}
```

Mutex değişkeni tanımla

Mutex değişkenini kilitle

Kilidi kaldır

Mutex Değişkenini (lock) Yarat

tid[i] threadini yarat

tid[0] için bekle ve statüsünü alma

Mutex Değişkenini Kaldır

```
}
```

## Örnek : condition wait ve condition signal kullanımı

(thread1, başka bir threadin üreteceği bir değeri kullanmadan iş yapamıyor. Bu değerın üretilmesi durumu bizim için condition olabilir)

```
#include <pthread.h>
#include <stdio.h>
#include <time.h>
#include <unistd.h>

pthread_cond_t cond;
pthread_mutex_t mutex;

int x1 = 2, x2 = 0, y=0;

void * threadfun1 (void *arg) {

    pthread_mutex_lock(&mutex);

    x1++;
    //burada birisinin x2 yi üretmesi için beklemesi lazım.
    if (x2==0) //eğer condition sağlanmamışsa uyu
        pthread_cond_wait(&cond, &mutex);
    y=x1+x2;

    pthread_mutex_unlock(&mutex);
    return NULL;
}

void * threadfun2 (void *arg) {

    x2++;
//simdi x2 üretildi, cond üzerinde uyuyandan birisini uyandır
    pthread_cond_signal(&cond);
    return NULL;
}

main() { ...
    pthread_t thid1, thid2;
    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&cond, NULL);

    pthread_create(&thid1, NULL, threadfun1, NULL);
    pthread_create(&thid2, NULL, threadfun2, NULL);
    pthread_join(thid1, NULL);
    pthread_join(thid2, NULL);
}
```

```
    printf("%d",y);  
    ...  
}
```