Bölüm 3: Süreçler

İşletim Sistemleri

Süreç kavramı | Bellek yerleşimi | Oluşturma: fork / exec / wait | Sonlandırma, zombi, yetim | Süreç durumları | PCB | Çizelgeleme kuyrukları | Bağlam değiştirme | Paylaşımlı bellek | İleti geçirme | Borular ve soketler | Mobil sistemlerde çoklu görev

Sercan KÜLCÜ · Giresun Üniversitesi · Bilgisayar Mühendisliği

© Sercan Külcü. Tüm hakları saklıdır. · Sunumun etkileşimli HTML sürümü: https://sercankulcu.github.io/files/os/slides/Bolum_03_Surecler.html

Öğrenme Çıktıları

Süreç modeli
Program ile süreç arasındaki farkı ve bir sürecin bellek yerleşimini açıklamak.

Yaşam döngüsü
Süreç durumlarını, geçişleri ve PCB'nin içeriğini tanımlamak.

UNIX API
`fork()`, `exec()`, `wait()`, `exit()` ve `pipe()` kullanan küçük programları izlemek.

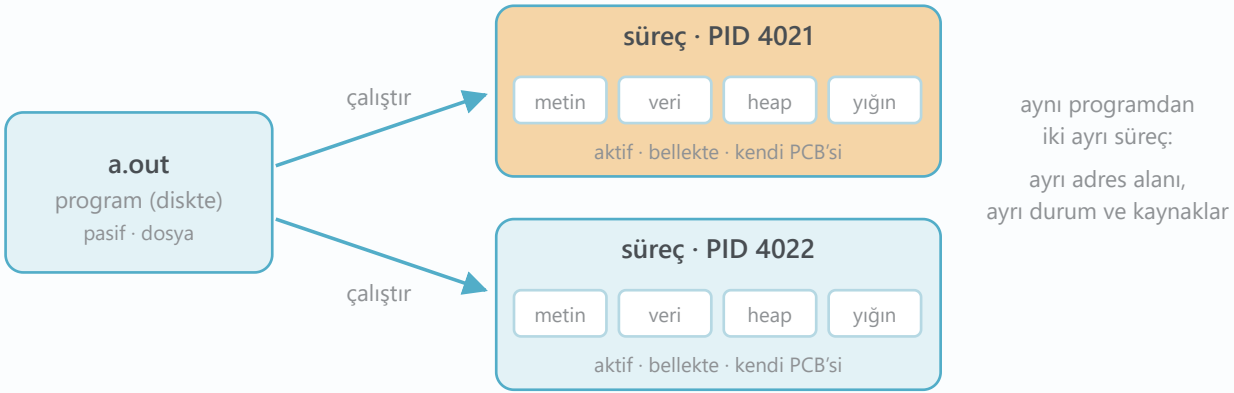
Çizelgeleme altyapısı
Hazır/bekleme kuyruklarını ve bağlam değiştirmenin maliyetini açıklamak.

IPC
Paylaşımlı bellek, ileti geçirme, boru ve soketleri karşılaştırmak.

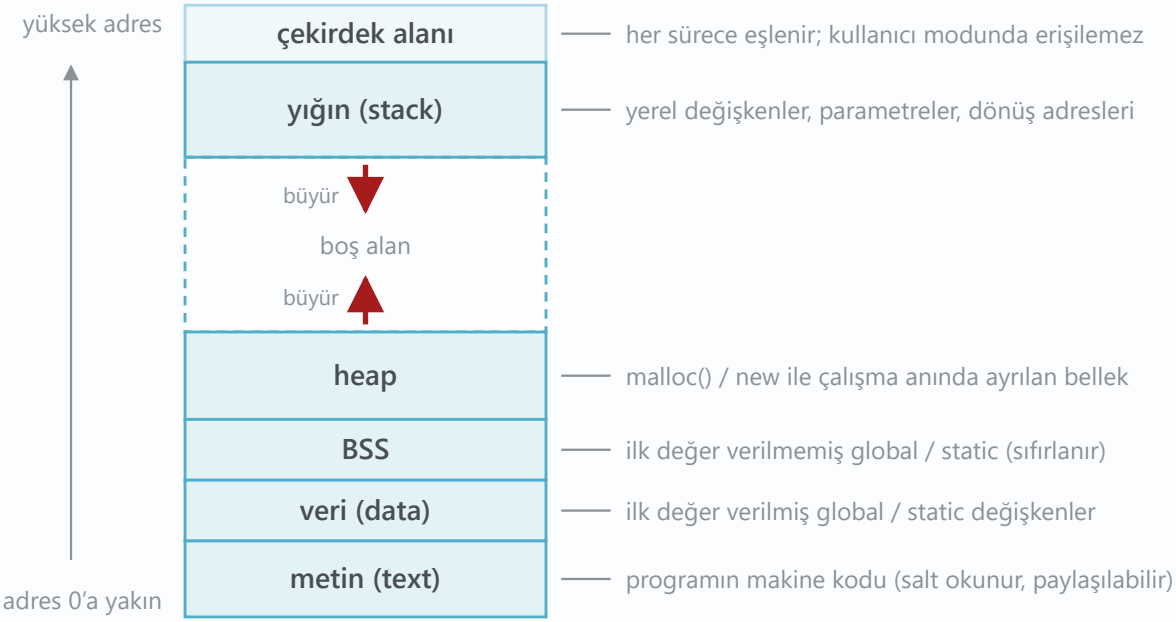
Mobil çoklu görev
iOS ve Android'in arka plan süreçlerini neden ve nasıl sınırladığını açıklamak.

Süreç Nedir?

- **Süreç (process):** yürütülmekte olan bir program örneği.
- **Program pasiftir** (diskte duran çalıştırılabilir dosya); **süreç aktiftir** (program sayacı, yazmaçlar, bellek, açık dosyalar). Program belleğe yüklenince süreç olur.
- Aynı programdan **birden çok süreç** oluşabilir (iki terminalde `vim`); her biri ayrı adres alanı ve kaynaklara sahiptir, metin bölümü paylaşılabilir.
- Tek iş parçacıklı bir süreç komutlarını **sıralı** yürütür; çok iş parçacıklı bir süreç çok çekirdekte paralel ilerleyebilir.



Bellekte Süreç Yerleşimi



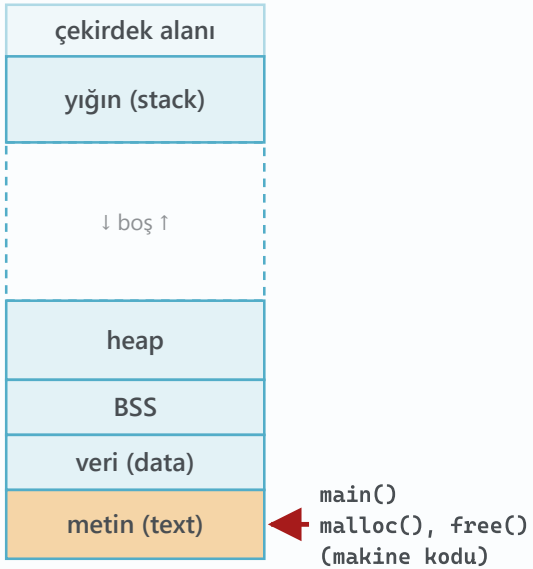
Yığın (stack) yüksek adreslerden aşağı, heap yukarı doğru büyür. Program sayacı bellekte değil, **işlemcide bir yazmaçtır**; sürecin "şu anki konumu"nu gösterir. Gerçek düzende ASLR adresleri her çalıştırmada rastgele kaydırır.

Değişkenler Nereye Yerleşir?

```c
#include <stdlib.h>
int sayac = 5;    /* data */
int tablo[1000];  /* bss */

int main(int argc, char *argv[]) {
    int yerel = 3;
    static int adet;
    int *p = malloc(4 * sizeof *p);
    p[0] = yerel + sayac;
    free(p);
    return 0;
}
```

Linux'ta `size a.out` metin/veri/BSS boyutlarını, `cat /proc/PID/maps` çalışan sürecin bölgelerini gösterir.



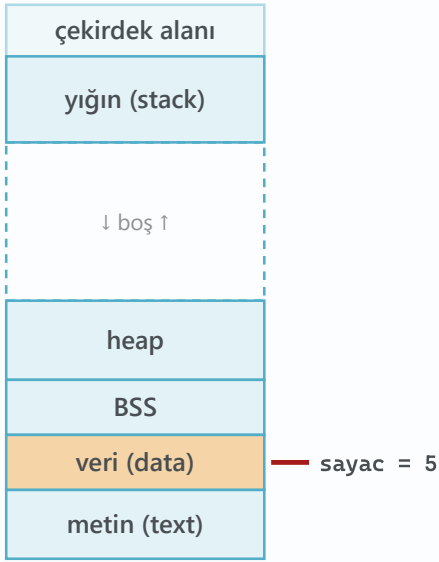
Metin: `main` fonksiyonunun derlenmiş komutları; salt okunur ve aynı programın süreçleri arasında paylaşılabilir.

Değişkenler Nereye Yerleşir?

```c
#include <stdlib.h>
int sayac = 5;    /* data */
int tablo[1000];  /* bss */

int main(int argc, char *argv[]) {
    int yerel = 3;
    static int adet;
    int *p = malloc(4 * sizeof *p);
    p[0] = yerel + sayac;
    free(p);
    return 0;
}
```

Linux'ta `size a.out` metin/veri/BSS boyutlarını, `cat /proc/PID/maps` çalışan sürecin bölgelerini gösterir.



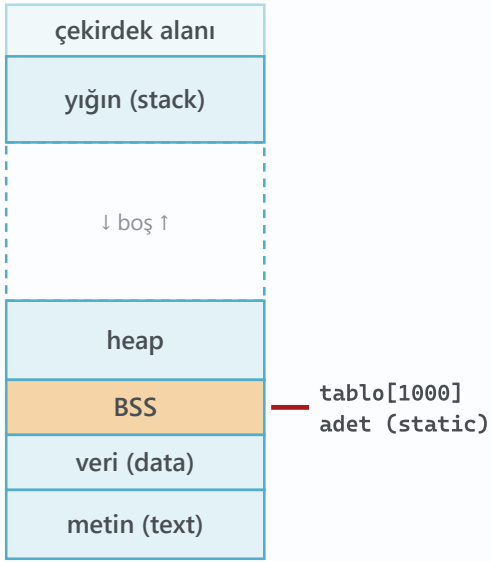
Veri: ilk değeri verilmiş global `sayac`; değeri çalıştırılabilir dosyada saklanır.

Değişkenler Nereye Yerleşir?

```c
#include <stdlib.h>
int sayac = 5;    /* data */
int tablo[1000];  /* bss */

int main(int argc, char *argv[]) {
    int yerel = 3;
    static int adet;
    int *p = malloc(4 * sizeof *p);
    p[0] = yerel + sayac;
    free(p);
    return 0;
}
```

Linux'ta `size a.out` metin/veri/BSS boyutlarını, `cat /proc/PID/maps` çalışan sürecin bölgelerini gösterir.



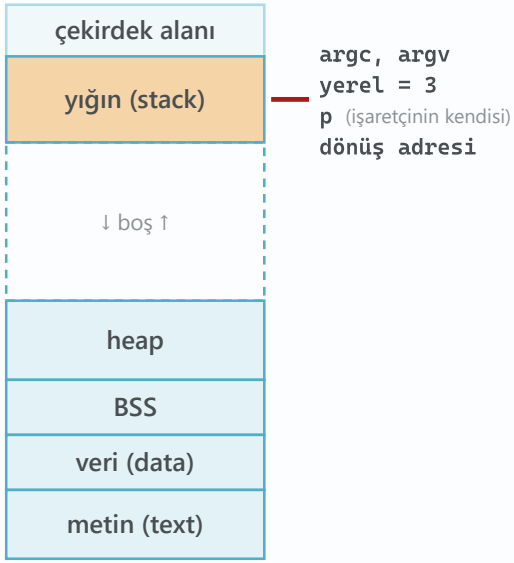
BSS: ilk değeri verilmemiş global ve `static` değişkenler; yükleyici sıfırla doldurur, dosyada yer kaplamaz.

Değişkenler Nereye Yerleşir?

```c
#include <stdlib.h>
int sayac = 5;    /* data */
int tablo[1000];  /* bss */

int main(int argc, char *argv[]) {
    int yerel = 3;
    static int adet;
    int *p = malloc(4 * sizeof *p);
    p[0] = yerel + sayac;
    free(p);
    return 0;
}
```

Linux'ta `size a.out` metin/veri/BSS boyutlarını, `cat /proc/PID/maps` çalışan sürecin bölgelerini gösterir.



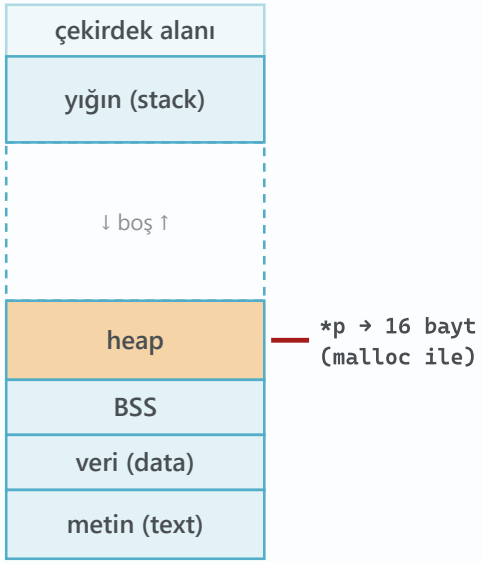
Yığın: `main` çağrı çerçevesi: parametreler, yerel değişkenler ve işaretçi `p`'nin kendisi.

Değişkenler Nereye Yerleşir?

```c
#include <stdlib.h>
int sayac = 5;    /* data */
int tablo[1000];  /* bss */

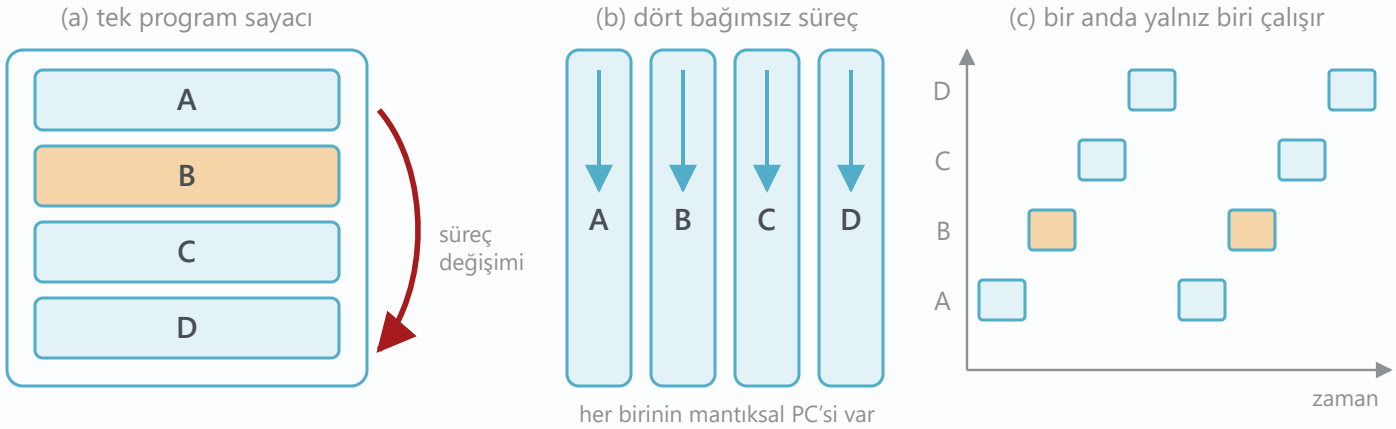
int main(int argc, char *argv[]) {
    int yerel = 3;
    static int adet;
    int *p = malloc(4 * sizeof *p);
    p[0] = yerel + sayac;
    free(p);
    return 0;
}
```

Linux'ta `size a.out` metin/veri/BSS boyutlarını, `cat /proc/PID/maps` çalışan sürecin bölgelerini gösterir.



Heap: `malloc` ile ayrılan 16 bayt; `p` yığında durur ama gösterdiği alan heap'tedir. `free` ile geri verilir.

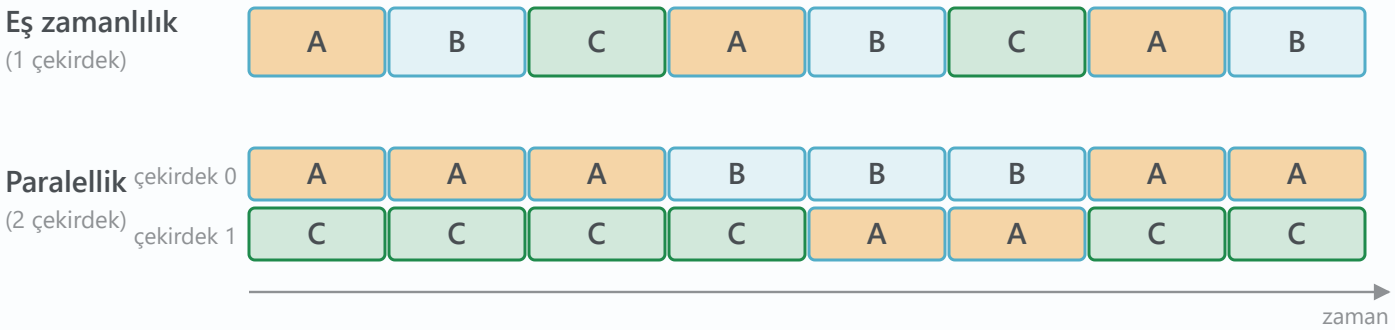
Sözde Paralellik ve Çoklu Programlama



- Tek çekirdekte herhangi bir anda yalnızca **bir** süreç yürütülür; çizelgeleyici süreçler arasında hızla geçiş yaparak **sözde paralellik** yaratır.
- Geçişler tipik olarak birkaç milisaniyelik zaman dilimleriyle olur (Linux EEVDF'de taban dilim ~3 ms; eski CFS'de hedef gecikme 6–24 ms); kullanıcıya tüm süreçler aynı anda çalışıyor gibi görünür.
- Kullanıcı için faydalıdır; ancak yönetimi zordur: süreçlerin ne zaman kesileceği **öngörülemez**, zaman varsayımı yapılmamalıdır.

SÜREÇ KAVRAMI

Eş Zamanlılık ve Paralellik



Eş zamanlılık (concurrency)

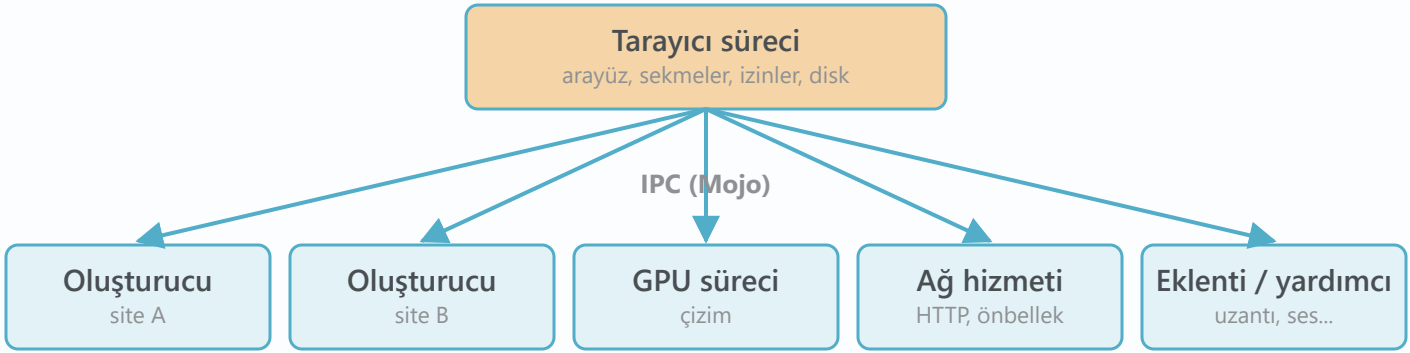
Birden çok iş aynı zaman aralığında *ilerler*; tek çekirdekte iç içe geçerek (interleaving).

Paralellik (parallelism)

Birden çok iş *aynı anda* yürütülür; birden çok çekirdek ya da işlemci gerekir.

2026 itibarıyla telefonlar bile 6–10 çekirdeklidir; yine de çalışan süreç sayısı (yüzlerce) çekirdek sayısından çok fazladır, bu yüzden çoklu programlama ve geçişler her sistemde vardır.

Örnek: Chrome'un Çok Süreçli Mimarisi



oluşturucular korumalı alanda (sandbox) çalışır;
bir sekme çökse de tarayıcı ve diğer sekmeler yaşar

Tarayıcı

Arayüzü, sekmeleri, disk ve izinleri yönetir; ayrıcalıklı tek süreç.

Oluşturucu (renderer)

HTML, CSS, JavaScript işler; site yalıtımı ile site başına ayrı süreç.

GPU / ağ / yardımcı

Çizim, ağ, uzantılar ayrı süreçlerde. Eski eklenti desteği kaldırıldı: NPAPI 2015'te, Flash 2020 sonunda.

Süreç Oluşturma

Süreç oluşturan olaylar

1. Sistem açılışı (init/systemd, arka plan hizmetleri: *daemon*).
2. Çalışan bir sürecin süreç oluşturan sistem çağrısı yapması.
3. Kullanıcının bir komut ya da uygulama başlatması.
4. Toplu (batch) bir işin başlatılması.

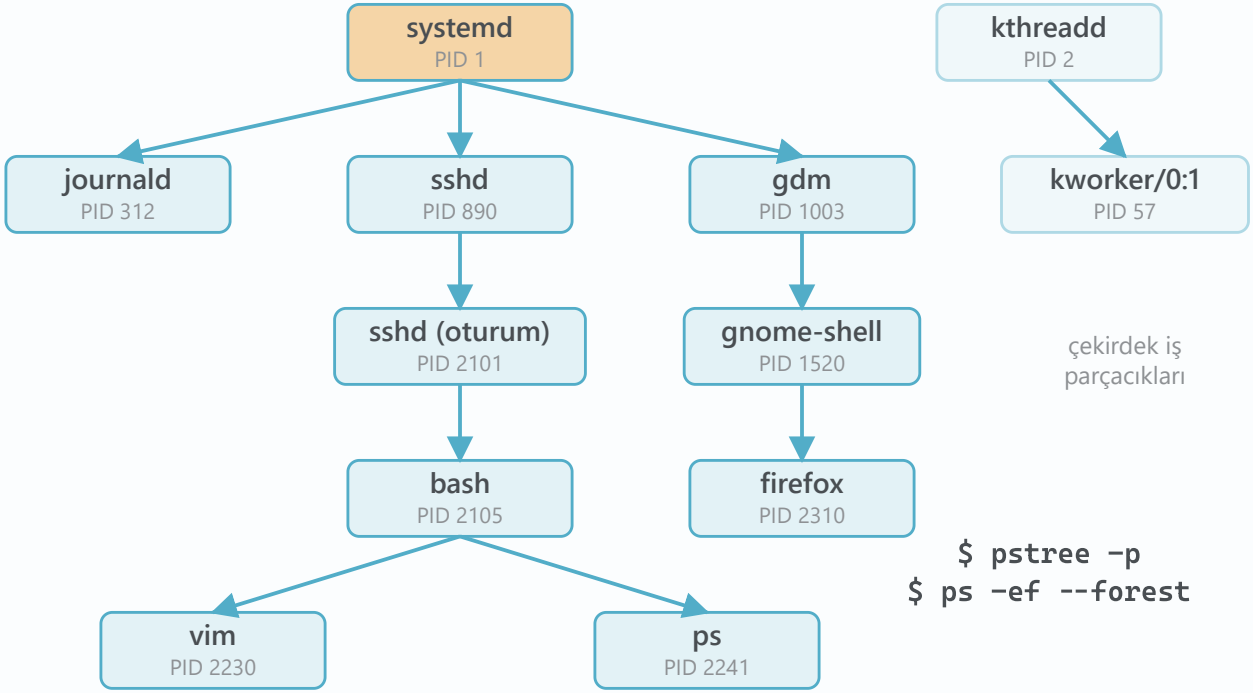
Her durumda teknik olarak aynı şey olur: var olan bir süreç, yeni süreç oluşturan bir **sistem çağrısı** yürütür.

Sistem çağrıları

Sistem	Çağrı
UNIX / Linux	`fork()` + `exec()`; ayrıca `posix_spawn()`, Linux'ta `clone()`
Windows	`CreateProcess()`: oluşturma ve program yüklemeyi tek çağrıda yapar

Ebeveyn (parent) süreç çocuk (child) süreç oluşturur; çocuklar da kendi çocuklarını oluşturarak bir **süreç ağacı** oluşur. Her süreç tekil bir **PID** taşır.

Süreç Ağacı



Linux açılışta PID 1 olarak `systemd` (ya da başka bir init) çalıştırır; PID 2 `kthreadd` çekirdek iş parçacıklarının ebeveynidir. PID değerleri örnektir.

SÜREÇ OLUŞTURMA

Ebeveyn ve Çocuk İlişkisi

Kaynak paylaşımı

Çocuk ebeveyn kaynaklarının tümünü ya da bir kısmını paylaşabilir ya da hiçbirini paylaşmayabilir. UNIX'te çocuk açık dosya tanımlayıcılarını **miras alır**.

Yürütme

Ebeveyn ve çocuk **eş zamanlı** çalışabilir ya da ebeveyn `wait()` ile çocuğun bitmesini bekleyebilir.

Adres alanı

UNIX `fork()`: çocuk ebeveynin adres alanının **kopyasını** alır (paylaşmaz). Ardından `exec()` ile yeni bir program yükleyebilir.

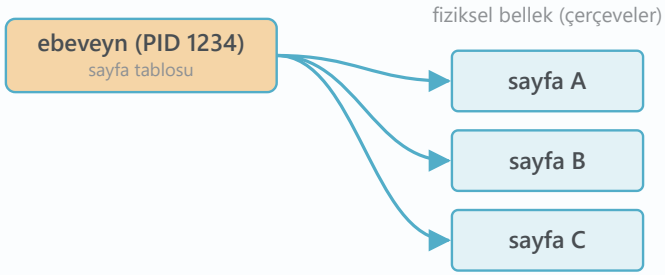
Kimlikler

`getpid()` kendi PID'ini, `getppid()` ebeveynin PID'ini döndürür.

`fork()` sonrası çocuğun adres alanı ebeveyninkinden **ayrıdır**; verimlilik için yalnızca fiziksel sayfalar yazılana kadar ortak tutulur (copy-on-write).

fork(): Bir Çağrı, İki Dönüş

```c
pid_t pid = fork();
if (pid < 0)        { perror("fork"); exit(1); }
else if (pid == 0)  { /* çocuk: fork 0 döndürdü */ }
else                { /* ebeveyn: pid = çocuğun PID'i */ }
```



Başlangıç: ebeveyn sürecin sayfa tablosu fiziksel A, B, C sayfalarını gösteriyor.

fork(): Bir Çağrı, İki Dönüş

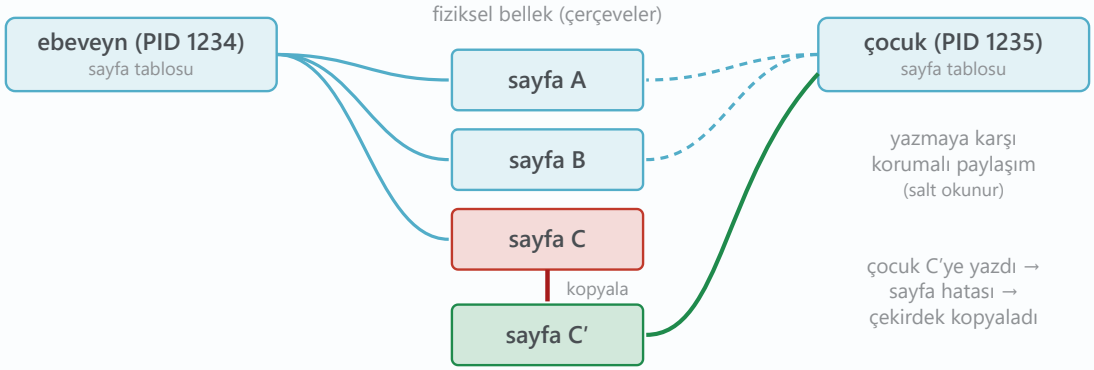
```
pid_t pid = fork();
if (pid < 0)        { perror("fork"); exit(1); }
else if (pid == 0)  { /* çocuk: fork 0 döndürdü */ }
else                { /* ebeveyn: pid = çocuğun PID'i */ }
```



`fork()` : çekirdek PCB'yi ve sayfa tablosunu kopyalar; sayfalar **kopyalanmaz**, ikisine de salt okunur eşlenir. Ebeveyn çocuğun PID'ini, çocuk 0 alır; hata olursa çocuk oluşmaz ve yalnız ebeveyne −1 döner.

fork(): Bir Çağrı, İki Dönüş

```
pid_t pid = fork();
if (pid < 0)        { perror("fork"); exit(1); }
else if (pid == 0)  { /* çocuk: fork 0 döndürdü */ }
else                { /* ebeveyn: pid = çocuğun PID'i */ }
```



Çocuk C sayfasına yazınca sayfa hatası oluşur; çekirdek yalnızca o sayfayı kopyalar (C′). Bu **yazınca kopyala (COW)** tekniğidir.

Kod İzleme: fork() + wait()

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <sys/wait.h>
4
5  int main(void) {
▶      int x = 10;
7      pid_t pid = fork();
8      if (pid == 0) {
9          x = x + 5;
10         printf("çocuk: x=%d\n", x);
11         return 0;
12     }
13     wait(NULL);
14     printf("ebeveyn: x=%d\n", x);
15     return 0;
16 }
```

DEĞİŞKENLER

`ebeveyn.x = 10`

ÇIKTI

ebeveyn · 1234
çalışıyor

Tek süreç var: ebeveyn (PID 1234), `x = 10`.

Adım 1 / 10

Kod İzleme: fork() + wait()

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

int main(void) {
    int x = 10;
    pid_t pid = fork();
    if (pid == 0) {
        x = x + 5;
        printf("çocuk: x=%d\n", x);
        return 0;
    }
    wait(NULL);
    printf("ebeveyn: x=%d\n", x);
    return 0;
}
```

DEĞİŞKENLER

`ebeveyn.x = 10` `ebeveyn.pid = 1235`

`çocuk.x = 10` `çocuk.pid = 0`

ÇIKTI

ebeveyn · 1234
fork() → 1235

çocuk · 1235
fork() → 0

`fork()` çocuk 1235'i oluşturur. Ebeveynde `pid = 1235`, çocukta `pid = 0`; iki ayrı `x` var.

Kod İzleme: fork() + wait()

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <sys/wait.h>
4
5  int main(void) {
6      int x = 10;
7      pid_t pid = fork();
8      if (pid == 0) {
9          x = x + 5;
10         printf("çocuk: x=%d\n", x);
11         return 0;
12     }
▶      wait(NULL);
14     printf("ebeveyn: x=%d\n", x);
15     return 0;
16 }
```

DEĞİŞKENLER

`ebeveyn.x = 10` `ebeveyn.pid = 1235`

`çocuk.x = 10` `çocuk.pid = 0`

ÇIKTI

ebeveyn · 1234
wait() → bekliyor

çocuk · 1235
hazır

Ebeveynde `pid ≠ 0` : `if` atlanır, `wait(NULL)` çocuk bitene kadar ebeveyni **bekliyor** durumuna sokar.

Kod İzleme: fork() + wait()

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <sys/wait.h>
4
5  int main(void) {
6      int x = 10;
7      pid_t pid = fork();
▶      if (pid == 0) {
9          x = x + 5;
10         printf("çocuk: x=%d\n", x);
11         return 0;
12     }
13     wait(NULL);
14     printf("ebeveyn: x=%d\n", x);
15     return 0;
16 }
```

DEĞİŞKENLER

`ebeveyn.x = 10` `ebeveyn.pid = 1235`

`çocuk.x = 10` `çocuk.pid = 0`

ÇIKTI

ebeveyn · 1234
bekliyor

çocuk · 1235
çalışıyor

Çocukta `pid == 0` doğru: çocuk dalına girer.
(Çocuk önce çalışmış olsaydı da sonuç aynı olurdu.)

Kod İzleme: fork() + wait()

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <sys/wait.h>
4
5  int main(void) {
6      int x = 10;
7      pid_t pid = fork();
8      if (pid == 0) {
▶          x = x + 5;
10         printf("çocuk: x=%d\n", x);
11         return 0;
12     }
13     wait(NULL);
14     printf("ebeveyn: x=%d\n", x);
15     return 0;
16 }
```

DEĞİŞKENLER

`ebeveyn.x = 10` `ebeveyn.pid = 1235`

`çocuk.x = 15` `çocuk.pid = 0`

ÇIKTI

ebeveyn · 1234
bekliyor

çocuk · 1235
çalışıyor

Çocuk kendi kopyasını değiştirir: `x = 15`.
Ebeveynin `x`'i **10** kalır.

Kod İzleme: fork() + wait()

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <sys/wait.h>
4
5  int main(void) {
6      int x = 10;
7      pid_t pid = fork();
8      if (pid == 0) {
9          x = x + 5;
▶          printf("çocuk: x=%d\n", x);
11         return 0;
12     }
13     wait(NULL);
14     printf("ebeveyn: x=%d\n", x);
15     return 0;
16 }
```

DEĞİŞKENLER

`ebeveyn.x = 10` `ebeveyn.pid = 1235`

`çocuk.x = 15` `çocuk.pid = 0`

ÇIKTI

```
çocuk: x=15
```

ebeveyn · 1234
bekliyor

çocuk · 1235
çalışıyor

Çocuk yazdırır.

Kod İzleme: fork() + wait()

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

int main(void) {
    int x = 10;
    pid_t pid = fork();
    if (pid == 0) {
        x = x + 5;
        printf("çocuk: x=%d\n", x);
        return 0;
    }
    wait(NULL);
    printf("ebeveyn: x=%d\n", x);
    return 0;
}
```

DEĞİŞKENLER

`ebeveyn.x = 10` `ebeveyn.pid = 1235`

`çocuk.x = 15` `çocuk.pid = 0`

ÇIKTI

```
çocuk: x=15
```

ebeveyn · 1234
uyandı (SIGCHLD)

çocuk · 1235
zombi

Çocuk `return 0` ile çıkar (`exit(0)`); kısa süre zombi olur, çekirdek ebeveyni uyandırır.

Kod İzleme: fork() + wait()

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <sys/wait.h>
4
5  int main(void) {
6      int x = 10;
7      pid_t pid = fork();
8      if (pid == 0) {
9          x = x + 5;
10         printf("çocuk: x=%d\n", x);
11         return 0;
12     }
▶      wait(NULL);
14     printf("ebeveyn: x=%d\n", x);
15     return 0;
16 }
```

DEĞİŞKENLER

`ebeveyn.x = 10` `ebeveyn.pid = 1235`

ÇIKTI

```
çocuk: x=15
```

ebeveyn · 1234
çalışıyor

çocuk · 1235
biçildi

`wait` 1235 döndürür; çocuğun girdisi süreç tablosundan silinir.

Kod İzleme: fork() + wait()

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <sys/wait.h>
4
5  int main(void) {
6      int x = 10;
7      pid_t pid = fork();
8      if (pid == 0) {
9          x = x + 5;
10         printf("çocuk: x=%d\n", x);
11         return 0;
12     }
13     wait(NULL);
▶      printf("ebeveyn: x=%d\n", x);
15     return 0;
16 }
```

DEĞİŞKENLER

`ebeveyn.x = 10` `ebeveyn.pid = 1235`

ÇIKTI

```
çocuk: x=15
ebeveyn: x=10
```

ebeveyn · 1234
çalışıyor

çocuk · 1235
biçildi

Ebeveyn yazdırır: kendi `x`'i hâlâ 10.

Kod İzleme: fork() + wait()

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <sys/wait.h>
4
5  int main(void) {
6      int x = 10;
7      pid_t pid = fork();
8      if (pid == 0) {
9          x = x + 5;
10         printf("çocuk: x=%d\n", x);
11         return 0;
12     }
13     wait(NULL);
14     printf("ebeveyn: x=%d\n", x);
▶      return 0;
16 }
```

DEĞİŞKENLER

`ebeveyn.x = 10` `ebeveyn.pid = 1235`

ÇIKTI

```
çocuk: x=15
ebeveyn: x=10
```

ebeveyn · 1234
sonlandı

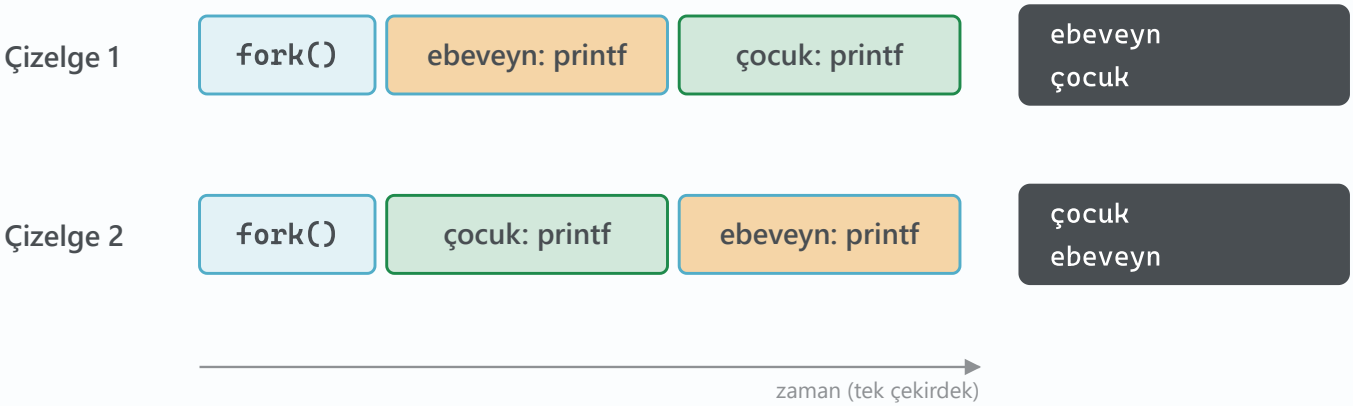
çocuk · 1235
biçildi

Ebeveyn de sonlanır. `wait()` sayesinde çıktı sırası **her çalıştırmada aynıdır**.

Adım 10 / 10

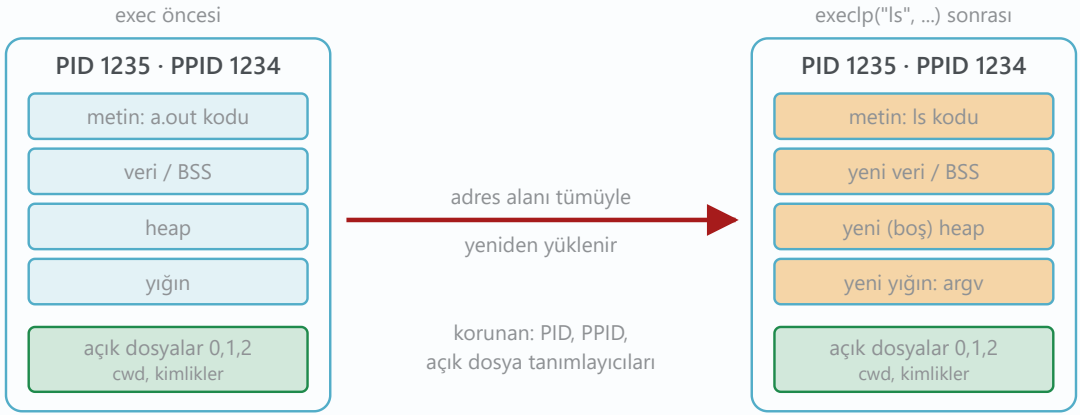
wait() Olmadan Çıktı Sırası Belirsizdir

```c
pid_t pid = fork();
if (pid == 0) printf("çocuk\n");
else          printf("ebeveyn\n");
```



fork sonrası hangi sürecin önce çalışacağını **çizelgeleyici** belirler; çok çekirdekte ikisi aynı anda bile çalışabilir. Programın doğruluğu bu sıraya bağlıysa `wait()` (ya da başka bir senkronizasyon) kullanılmalıdır.

exec() Ailesi



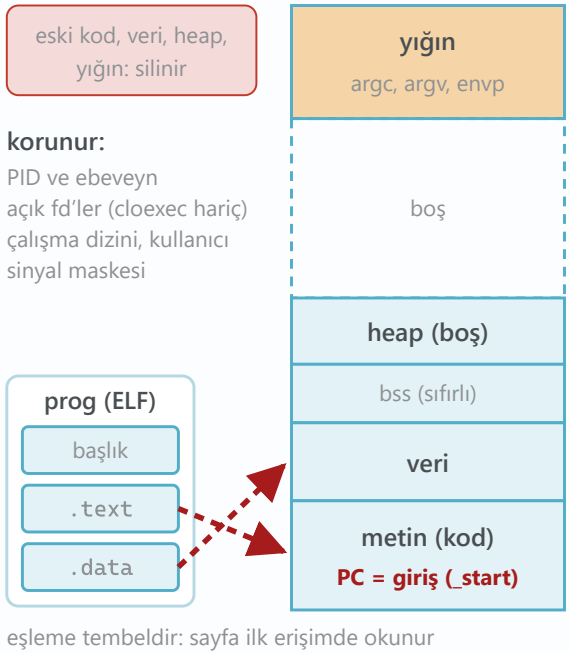
Çağrı	Argümanlar	PATH araması	Ortam
`execl` / `execv`	liste / dizi	hayır	devralınır
`execlp` / `execvp`	liste / dizi	evet	devralınır
`execle` / `execve`	liste / dizi	hayır	açıkça verilir

Hepsi `execve()` sistem çağrısına varır. Başarılı exec **geri dönmez**; dönüyorsa hata vardır.

exec Çekirdekte Ne Yapar?

1. Dosyayı aç; izni ve biçimi (ELF başlığı ya da `#!` satırı) denetle.
   Dosya yoksa ya da çalıştırılamıyorsa `exec` −1 döner ve eski program kaldığı yerden sürer.

Adım 1 / 6



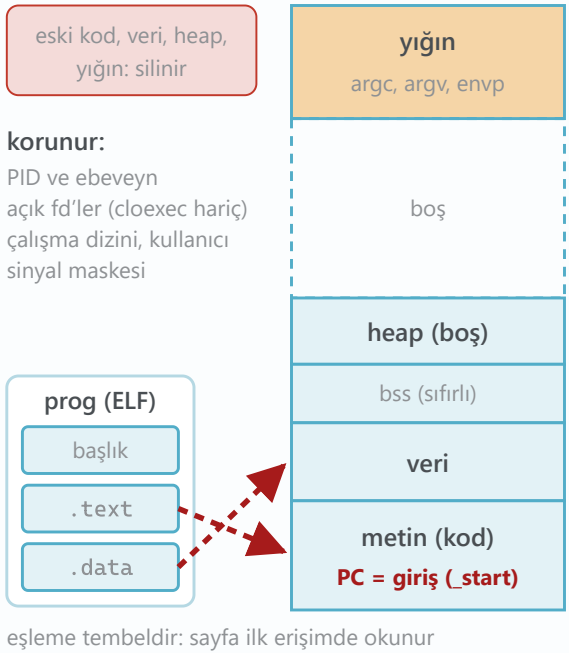
eşleme tembeldir: sayfa ilk erişimde okunur

Diskteki program pasif bir dosyadır; exec onu çalışan bir sürecin adres alanına dönüştürür. Linux'ta tüm exec ailesi sonunda `execve()`'ye iner.

exec Çekirdekte Ne Yapar?

1. Dosyayı aç; izni ve biçimi (ELF başlığı ya da `#!` satırı) denetle.

2. Eski adres alanını bırak.
Kod, veri, heap ve yığın eşlemeleri silinir; süreçteki diğer iş parçacıkları sonlandırılır. Bu noktadan sonra geri dönülecek program yoktur.



eşleme tembeldir: sayfa ilk erişimde okunur

Diskteki program pasif bir dosyadır; exec onu çalışan bir sürecin adres alanına dönüştürür. Linux'ta tüm exec ailesi sonunda `execve()`'ye iner.

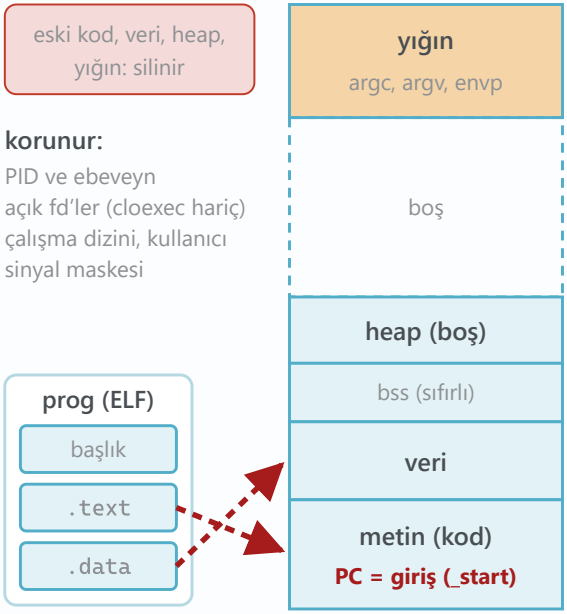
exec Çekirdekte Ne Yapar?

1. Dosyayı aç; izni ve biçimi (ELF başlığı ya da `#!` satırı) denetle.

2. Eski adres alanını bırak.

3. close-on-exec fd'leri kapat, sinyal işleyicilerini sıfırla.
   Yakalanan sinyaller varsayılan eyleme döner, çünkü işleyici kodu artık bellekte değildir; yok sayılanlar yok sayılmaya devam eder.

Adım 3 / 6



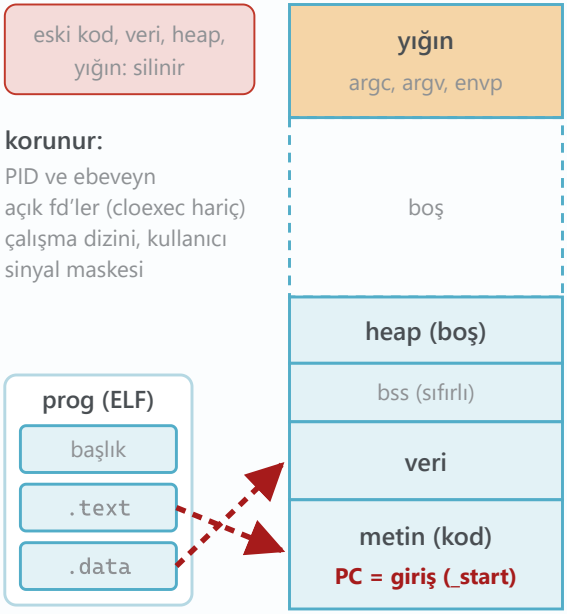
eşleme tembeldir: sayfa ilk erişimde okunur

Diskteki program pasif bir dosyadır; exec onu çalışan bir sürecin adres alanına dönüştürür. Linux'ta tüm exec ailesi sonunda `execve()`'ye iner.

exec Çekirdekte Ne Yapar?

1. Dosyayı aç; izni ve biçimi (ELF başlığı ya da `#!` satırı) denetle.
2. Eski adres alanını bırak.
3. close-on-exec fd'leri kapat, sinyal işleyicilerini sıfırla.
4. Kod ve veri bölümlerini dosyadan eşle.
   Kopyalama yapılmaz: bir sayfa ilk kez kullanıldığında sayfa hatası olur ve o sayfa diskten getirilir (tembel yükleme).

Adım 4 / 6



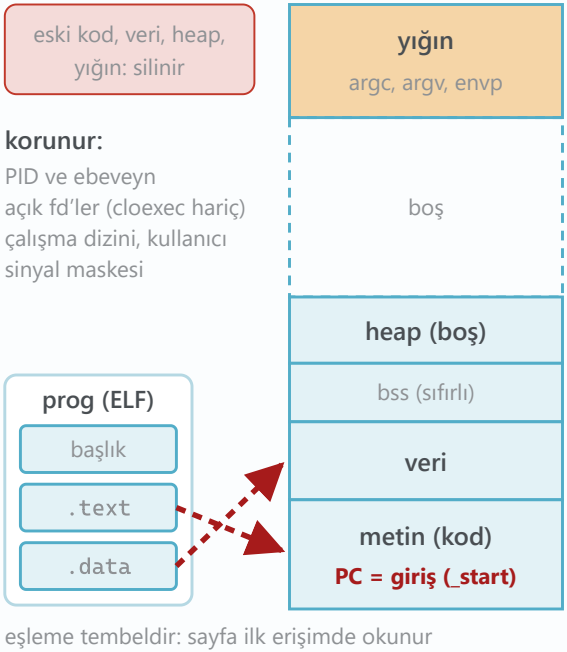
eşleme tembeldir: sayfa ilk erişimde okunur

Diskteki program pasif bir dosyadır; exec onu çalışan bir sürecin adres alanına dönüştürür. Linux'ta tüm exec ailesi sonunda `execve()`'ye iner.

exec Çekirdekte Ne Yapar?

1. Dosyayı aç; izni ve biçimi (ELF başlığı ya da `#!` satırı) denetle.
2. Eski adres alanını bırak.
3. close-on-exec fd'leri kapat, sinyal işleyicilerini sıfırla.
4. Kod ve veri bölümlerini dosyadan eşle.
5. bss, heap ve yığın için sıfırlı bellek ayır.
   Yığının tepesine argv ve envp dizgileri, bunları gösteren işaretçi dizileri ve argc yerleştirilir.

Adım 5 / 6



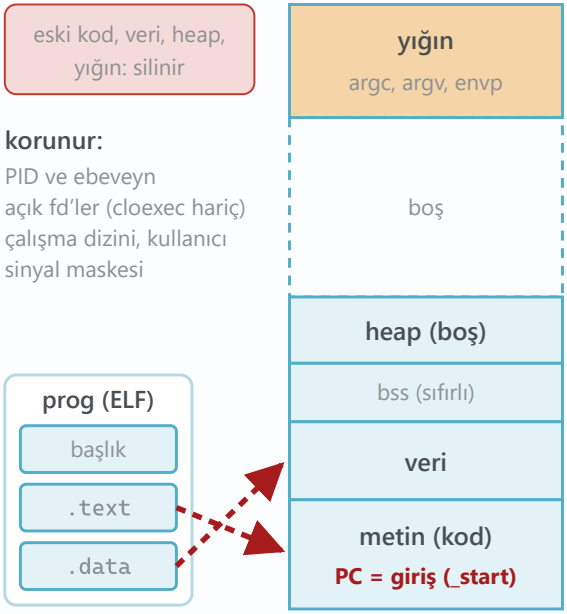
eşleme tembeldir: sayfa ilk erişimde okunur

Diskteki program pasif bir dosyadır; exec onu çalışan bir sürecin adres alanına dönüştürür. Linux'ta tüm exec ailesi sonunda `execve()` 'ye iner.

exec Çekirdekte Ne Yapar?

1. Dosyayı aç; izni ve biçimi (ELF başlığı ya da `#!` satırı) denetle.
2. Eski adres alanını bırak.
3. close-on-exec fd'leri kapat, sinyal işleyicilerini sıfırla.
4. Kod ve veri bölümlerini dosyadan eşle.
5. bss, heap ve yığın için sıfırlı bellek ayır.
6. PC = giriş noktası; kullanıcı moduna dön.
   `_start` C çalışma zamanını kurar ve `main(argc, argv, envp)` 'yi çağırır. PID hiç değişmemiştir.

Adım 6 / 6



eşleme tembeldir: sayfa ilk erişimde okunur

Diskteki program pasif bir dosyadır; exec onu çalışan bir sürecin adres alanına dönüştürür. Linux'ta tüm exec ailesi sonunda `execve()` 'ye iner.

Kod İzleme: fork + exec + waitpid

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/wait.h>
5
6  int main(void) {
▶    pid_t pid = fork();
8    if (pid == 0) {
9      execlp("ls", "ls", "-l", NULL);
10     perror("execlp");  /* hata */
11     exit(1);
12   }
13   int st;
14   waitpid(pid, &st, 0);
15   printf("bitti: %d\n", WEXITSTATUS(st));
16   return 0;
17 }
```

DEĞİŞKENLER

`ebeveyn.pid = 1235` `çocuk.pid = 0`

ÇIKTI



ebeveyn · 1234
fork() → 1235

çocuk · 1235
fork() → 0

`fork()` : iki süreç aynı `a.out` görüntüsünü çalıştırıyor.

Kod İzleme: fork + exec + waitpid

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/wait.h>
5
6  int main(void) {
7    pid_t pid = fork();
8    if (pid == 0) {
9      execlp("ls", "ls", "-l", NULL);
10     perror("execlp");  /* hata */
11     exit(1);
12   }
13   int st;
▶    waitpid(pid, &st, 0);
15   printf("bitti: %d\n", WEXITSTATUS(st));
16   return 0;
17 }
```

DEĞİŞKENLER

`ebeveyn.pid = 1235` `çocuk.pid = 0`

ÇIKTI

ebeveyn · 1234
waitpid → bekliyor

çocuk · 1235
hazır

Ebeveyn çocuk dalını atlar ve `waitpid(1235, ...)` ile bloke olur.

Kod İzleme: fork + exec + waitpid

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/wait.h>
5
6  int main(void) {
7    pid_t pid = fork();
8    if (pid == 0) {
▶      execlp("ls", "ls", "-l", NULL);
10     perror("execlp");  /* hata */
11     exit(1);
12   }
13   int st;
14   waitpid(pid, &st, 0);
15   printf("bitti: %d\n", WEXITSTATUS(st));
16   return 0;
17 }
```

DEĞİŞKENLER

`ebeveyn.pid = 1235` `çocuk = ls -l`

ÇIKTI



ebeveyn · 1234
bekliyor

çocuk · 1235
exec: ls

Çocuk `execlp` çağırır: PATH içinde `ls` bulunur, adres alanı `/usr/bin/ls` ile **değiştirilir**. PID hâlâ 1235.

Kod İzleme: fork + exec + waitpid

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/wait.h>
5
6  int main(void) {
7    pid_t pid = fork();
8    if (pid == 0) {
▶      execlp("ls", "ls", "-l", NULL);
10     perror("execlp");  /* hata */
11     exit(1);
12   }
13   int st;
14   waitpid(pid, &st, 0);
15   printf("bitti: %d\n", WEXITSTATUS(st));
16   return 0;
17 }
```

DEĞİŞKENLER

`ebeveyn.pid = 1235` `çocuk = ls -l`

ÇIKTI

```
(ls -l çıktısı)
```

ebeveyn · 1234
bekliyor

çocuk · 1235
ls çalışıyor

Artık çocuk `ls` programıdır; 10–11. satırlar **hiç çalışmaz** (exec başarılı). Miras alınan stdout'a listeyi yazar.

Kod İzleme: fork + exec + waitpid

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/wait.h>
5
6  int main(void) {
7    pid_t pid = fork();
8    if (pid == 0) {
9      execlp("ls", "ls", "-l", NULL);
10     perror("execlp");  /* hata */
11     exit(1);
12   }
13   int st;
▶    waitpid(pid, &st, 0);
15   printf("bitti: %d\n", WEXITSTATUS(st));
16   return 0;
17 }
```

DEĞİŞKENLER

`ebeveyn.pid = 1235` `st = 0 (kodlanmış)`

ÇIKTI

```
(ls -l çıktısı)
```

ebeveyn · 1234
uyandı

çocuk · 1235
çıktı: 0

`ls` `exit(0)` ile biter; `waitpid` döner ve `st` çıkış durumunu taşır.

Kod İzleme: fork + exec + waitpid

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/wait.h>
5
6  int main(void) {
7    pid_t pid = fork();
8    if (pid == 0) {
9      execlp("ls", "ls", "-l", NULL);
10     perror("execlp");  /* hata */
11     exit(1);
12   }
13   int st;
14   waitpid(pid, &st, 0);
▶    printf("bitti: %d\n", WEXITSTATUS(st));
16   return 0;
17 }
```

DEĞİŞKENLER

`ebeveyn.pid = 1235` `WEXITSTATUS = 0`

ÇIKTI

```
(ls -l çıktısı)
bitti: 0
```

ebeveyn · 1234
çalışıyor

çocuk · 1235
biçildi

`WEXITSTATUS(st)` = 0 yazdırılır.

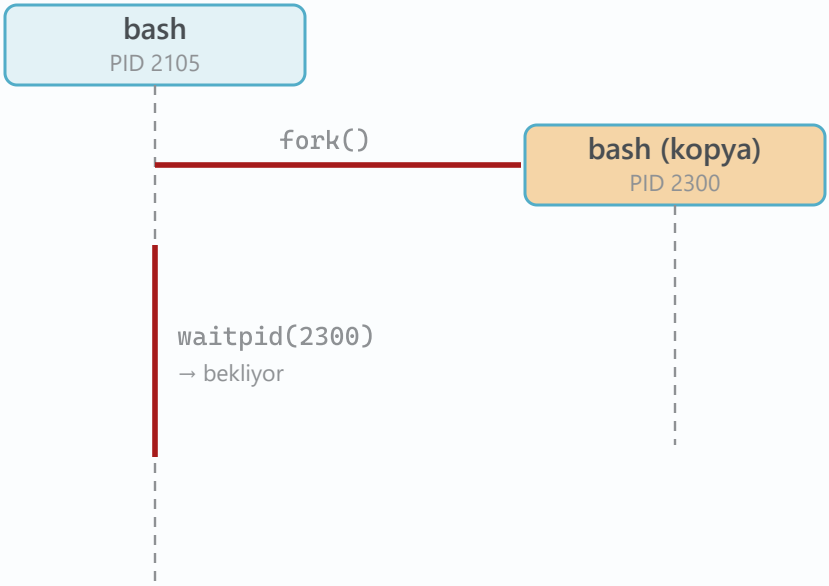
Adım 6 / 6

Kabuk Döngüsü: fork → exec → wait



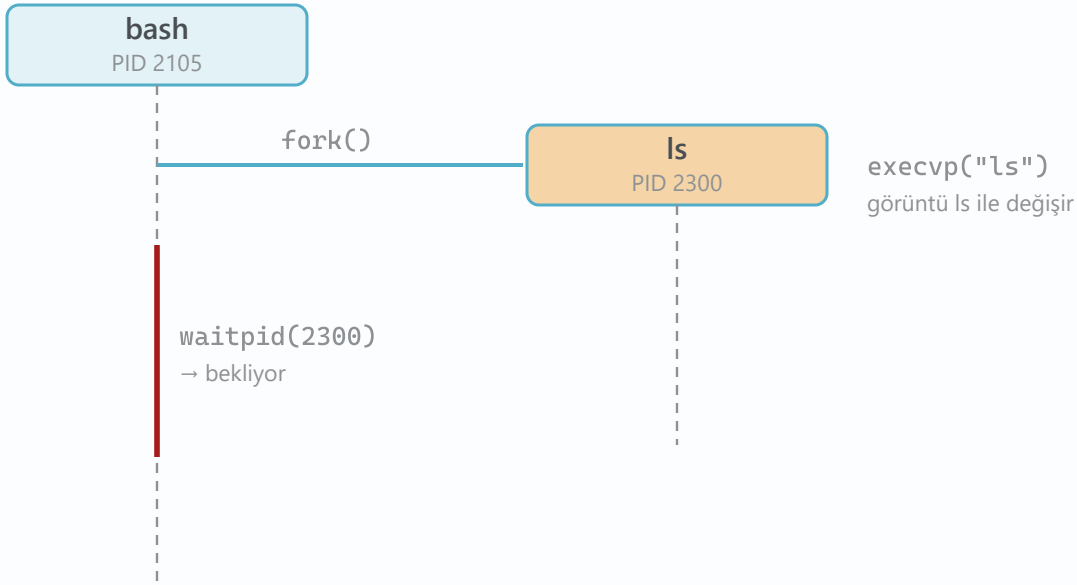
Kabuk istemi gösterir ve `ls` komutunu okur.

Kabuk Döngüsü: fork → exec → wait



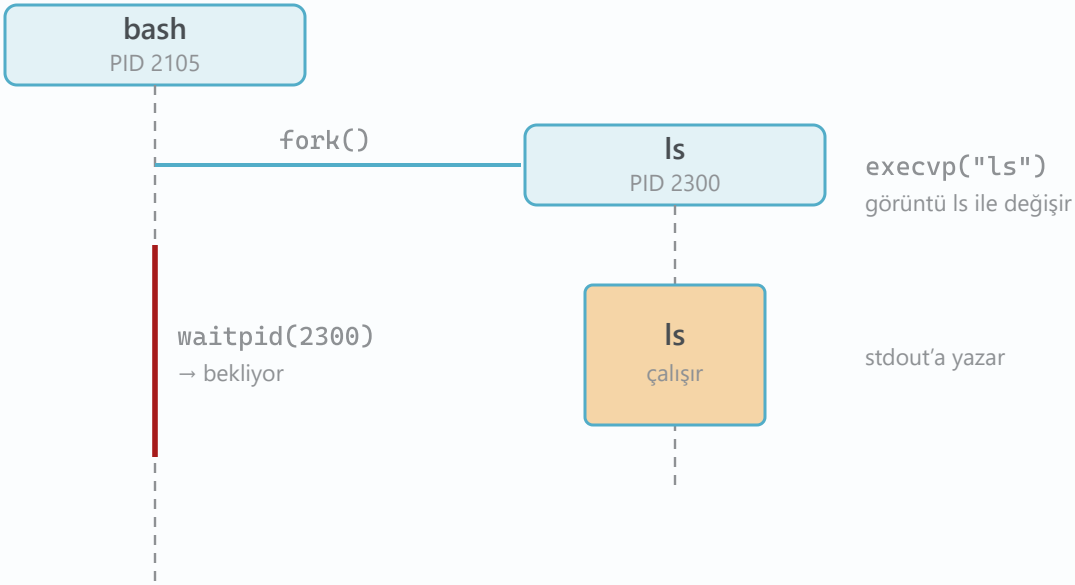
`fork()` : kabuğun bir kopyası (PID 2300) oluşur; ebeveyn `waitpid` ile beklemeye geçer.

Kabuk Döngüsü: fork → exec → wait



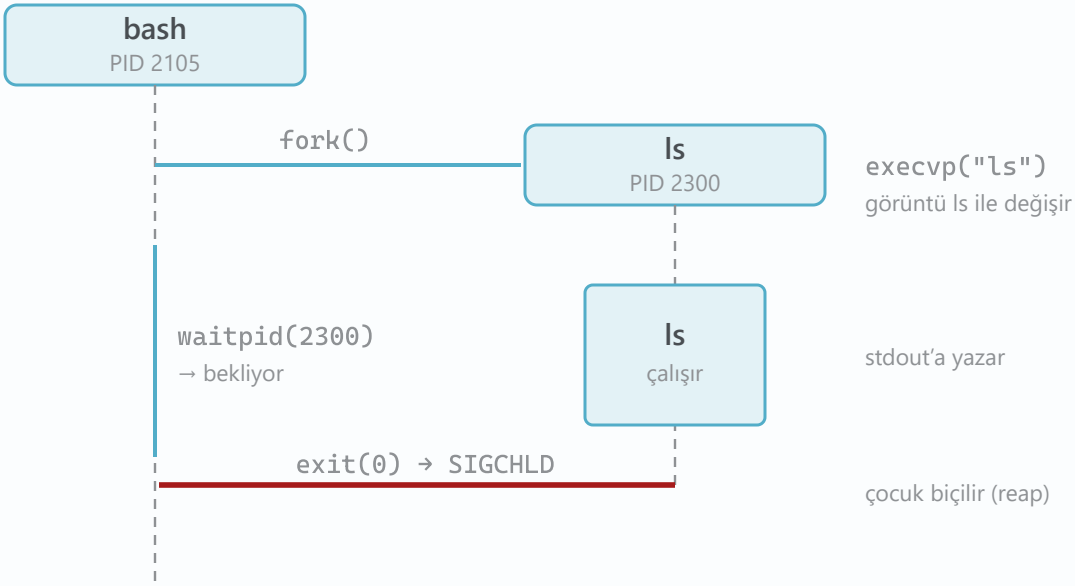
Çocuk `execvp("ls", argv)` çağırır; görüntüsü `ls` ile değişir.

Kabuk Döngüsü: fork → exec → wait



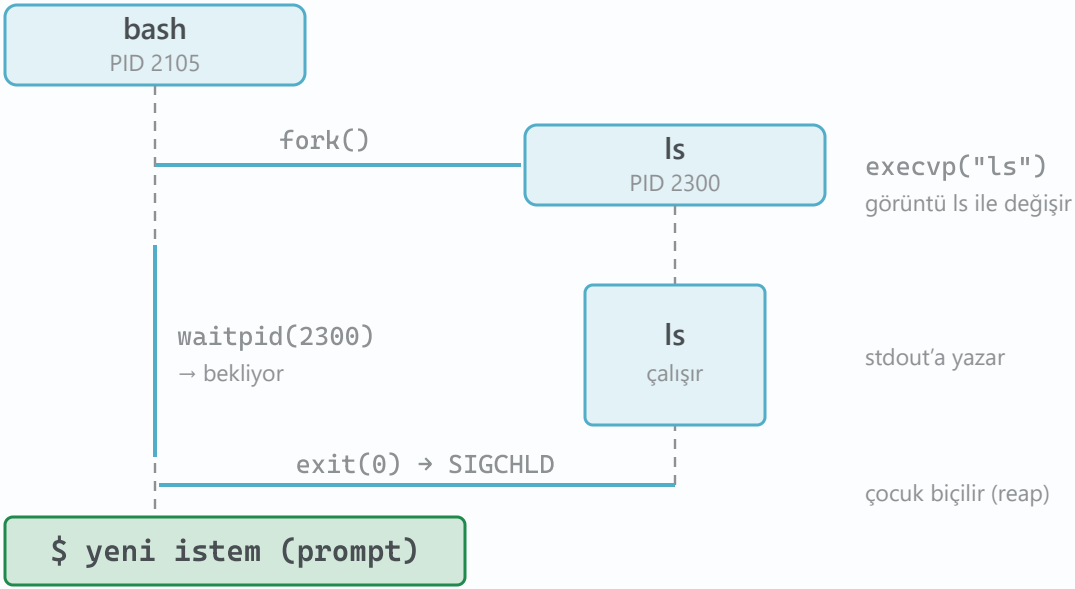
`ls` çalışır, miras aldığı terminal stdout'una yazar.

Kabuk Döngüsü: fork → exec → wait



`ls` çıkar; çekirdek ebeveyni SIGCHLD ile bilgilendirir, `waitpid` çocuğu biçer.

Kabuk Döngüsü: fork → exec → wait



Kabuk yeni istemi gösterir. Komut sonuna `&` eklenirse kabuk beklemeden devam eder (arka plan işi).

Neden fork ve exec Ayrı? Çıktı Yönlendirme

```
1  int main(void) {
▶    pid_t pid = fork();
3    if (pid == 0) {    /* çocuk */
4      close(1);        /* stdout boşalır */
5      int fd = open("sayim.txt",
6               O_WRONLY | O_CREAT | O_TRUNC, 0644);
7      /* en küçük boş numara: fd == 1 */
8      execlp("wc", "wc", "-l", "notlar.txt", NULL);
9      perror("execlp");
10     exit(1);
11   }
12   waitpid(pid, NULL, 0);
13   printf("bitti\n");
14   return 0;
15 }
```

DEĞİŞKENLER

`ebeveyn.pid` = **1500**

`çocuk.pid` = **1501**

ÇIKTI

çocuğun fd tablosu (ebeveynin kopyası)

0 → tty	1 → tty	2 → tty

ebeveyn: 0, 1, 2 → terminal (değişmez)

`fork()` : çocuk ebeveynin fd tablosunu **kopyalar**; 0, 1, 2 terminaldir.

Adım 1 / 6

Kabuk `> sayim.txt` 'i böyle yapar: çocuk fork ile exec arasında ortamını (fd, dizin, öncelik) hazırlar. Windows'un `CreateProcess` 'i bunları parametre ister.

Neden fork ve exec Ayrı? Çıktı Yönlendirme

```
1  int main(void) {
2    pid_t pid = fork();
3    if (pid == 0) {    /* çocuk */
4      close(1);        /* stdout boşalır */
5      int fd = open("sayim.txt",
6               O_WRONLY | O_CREAT | O_TRUNC, 0644);
7      /* en küçük boş numara: fd == 1 */
8      execlp("wc", "wc", "-l", "notlar.txt", NULL);
9      perror("execlp");
10     exit(1);
11   }
▶  waitpid(pid, NULL, 0);
13   printf("bitti\n");
14   return 0;
15 }
```

DEĞİŞKENLER

`ebeveyn.pid = 1500`

`çocuk.pid = 1501`

ÇIKTI

çocuğun fd tablosu

0 → tty	1 → tty	2 → tty

ebeveyn: 0, 1, 2 → terminal (değişmez)

Ebeveyn `waitpid` ile bekler; sonraki adımlar yalnız çocuğun tablosunu değiştirir.

Kabuk `> sayim.txt`'i böyle yapar: çocuk fork ile exec arasında ortamını (fd, dizin, öncelik) hazırlar. Windows'un `CreateProcess`'i bunları parametre ister.

Neden fork ve exec Ayrı? Çıktı Yönlendirme

```
1  int main(void) {
2    pid_t pid = fork();
3    if (pid == 0) {     /* çocuk */
▶      close(1);         /* stdout boşalır */
5      int fd = open("sayim.txt",
6               O_WRONLY | O_CREAT | O_TRUNC, 0644);
7      /* en küçük boş numara: fd == 1 */
8      execlp("wc", "wc", "-l", "notlar.txt", NULL);
9      perror("execlp");
10     exit(1);
11   }
12   waitpid(pid, NULL, 0);
13   printf("bitti\n");
14   return 0;
15 }
```

DEĞİŞKENLER

`ebeveyn.pid = 1500`

`çocuk.pid = 1501`

ÇIKTI

çocuğun fd tablosu

0 → tty	1 → (boş)	2 → tty

ebeveyn: 0, 1, 2 → terminal (değişmez)

`close(1)` : çocuğun 1 numaralı girişi boşalır. Ebeveynin stdout'u hâlâ terminaldir.

Kabuk `> sayim.txt` 'i böyle yapar: çocuk fork ile exec arasında ortamını (fd, dizin, öncelik) hazırlar. Windows'un `CreateProcess` 'i bunları parametre ister.

Neden fork ve exec Ayrı? Çıktı Yönlendirme

```
1  int main(void) {
2    pid_t pid = fork();
3    if (pid == 0) {     /* çocuk */
4      close(1);         /* stdout boşalır */
▶      int fd = open("sayim.txt",
▶               O_WRONLY | O_CREAT | O_TRUNC, 0644);
7      /* en küçük boş numara: fd == 1 */
8      execlp("wc", "wc", "-l", "notlar.txt", NULL);
9      perror("execlp");
10     exit(1);
11   }
12   waitpid(pid, NULL, 0);
13   printf("bitti\n");
14   return 0;
15 }
```

DEĞİŞKENLER

`ebeveyn.pid = 1500`

`çocuk.pid = 1501` `fd = 1`

ÇIKTI

çocuğun fd tablosu

0 → tty	1 → sayim.txt	2 → tty

ebeveyn: 0, 1, 2 → terminal (değişmez)

`open` her zaman **en küçük boş** numarayı verir: 1. Artık çocuğun stdout'u dosyadır.

Kabuk `> sayim.txt` 'i böyle yapar: çocuk fork ile exec arasında ortamını (fd, dizin, öncelik) hazırlar. Windows'un `CreateProcess` 'i bunları parametre ister.

Neden fork ve exec Ayrı? Çıktı Yönlendirme

```c
int main(void) {
  pid_t pid = fork();
  if (pid == 0) {    /* çocuk */
    close(1);        /* stdout boşalır */
    int fd = open("sayim.txt",
              O_WRONLY | O_CREAT | O_TRUNC, 0644);
    /* en küçük boş numara: fd == 1 */
    execlp("wc", "wc", "-l", "notlar.txt", NULL);
    perror("execlp");
    exit(1);
  }
  waitpid(pid, NULL, 0);
  printf("bitti\n");
  return 0;
}
```

DEĞİŞKENLER

`ebeveyn.pid = 1500`

`çocuk = wc -l`

ÇIKTI

çocuk = wc · fd tablosu korundu

0 → tty	1 → sayim.txt	2 → tty

ebeveyn: 0, 1, 2 → terminal (değişmez)

`exec` kodu ve veriyi değiştirir, fd tablosunu **korur**: habersiz wc fd 1'e, yani dosyaya yazar.

Kabuk `> sayim.txt`'i böyle yapar: çocuk fork ile exec arasında ortamını (fd, dizin, öncelik) hazırlar. Windows'un `CreateProcess`'i bunları parametre ister.

Neden fork ve exec Ayrı? Çıktı Yönlendirme

```
1  int main(void) {
2    pid_t pid = fork();
3    if (pid == 0) {    /* çocuk */
4      close(1);        /* stdout boşalır */
5      int fd = open("sayim.txt",
6               O_WRONLY | O_CREAT | O_TRUNC, 0644);
7      /* en küçük boş numara: fd == 1 */
8      execlp("wc", "wc", "-l", "notlar.txt", NULL);
9      perror("execlp");
10     exit(1);
11   }
12   waitpid(pid, NULL, 0);
▶  13   printf("bitti\n");
14   return 0;
15 }
```

DEĞİŞKENLER

`ebeveyn.pid = 1500`

ÇIKTI

```
bitti
```

çocuk bitti; dosyada: 42 notlar.txt

0 → tty	1 → sayim.txt	2 → tty

$ cat sayim.txt → 42 notlar.txt

wc biter, `waitpid` döner; ebeveyn terminale yazar: ekranda yalnız **bitti**.

Adım 6 / 6

Kabuk `> sayim.txt` 'i böyle yapar: çocuk fork ile exec arasında ortamını (fd, dizin, öncelik) hazırlar. Windows'un `CreateProcess` 'i bunları parametre ister.

Süreç Sonlandırma

Neden	Tür	Örnek
Normal çıkış	gönüllü	`return 0`, `exit(0)`; derleyici işi bitti
Hata sonrası çıkış	gönüllü	dosya bulunamadı → `exit(1)`
Ölümcül hata	istemsiz	geçersiz bellek erişimi (SIGSEGV), sıfıra bölme (SIGFPE)
Başka süreç tarafından	istemsiz	`kill(pid, SIGTERM)`, Windows'ta `TerminateProcess()`

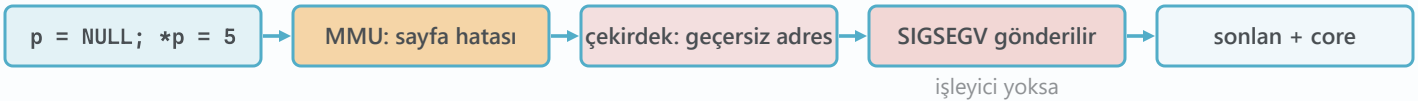
- `exit(durum)`: C kütüphanesi tamponları boşaltır, `atexit` işleyicilerini çalıştırır, sonra `_exit` sistem çağrısı.
- `abort()`: çağıran süreç kendine SIGABRT gönderir, anormal biter (çekirdek dökümü oluşabilir). Başka süreci sonlandırmak için `kill()` kullanılır.
- Çekirdek sürecin belleğini, açık dosyalarını ve diğer kaynaklarını geri alır; çıkış durumu ebeveyn `wait()` çağırana kadar saklanır.
- Bazı sistemlerde ebeveyn sonlanınca çocuklar da sonlandırılır (*basamaklı sonlandırma*); UNIX'te ise çocuklar yaşamaya devam eder.

Ölümcül Hatalar

Hata	Sinyal
Geçersiz/izinsiz bellek erişimi	SIGSEGV
Tam sayı sıfıra bölme	SIGFPE
Geçersiz komut	SIGILL
Hizalama / veri yolu hatası	SIGBUS
Kapalı boruya yazma	SIGPIPE

- İşletim sistemi donanım istisnasını yakalar, sürece ilgili **sinyali** gönderir; işlenmezse süreç sonlanır.
- Süreç bellekten temizlenir, kaynakları serbest bırakılır.
- **Çekirdek dökümü (core dump)** alınabilir: `coredumpctl` , `gdb a.out core` ile hatanın nedeni incelenir (crash dump analizi).

Bellek sızıntısı ve kaynakları serbest bırakmamak kendi başına ölümcül hata değildir; süreç çalışmayı sürdürür. Bellek tükenirse Linux OOM killer süreci **SIGKILL** ile sonlandırabilir.



SÜREÇ SONLANDIRMA

Sinyallerle Sonlandırma: kill, pkill, killall

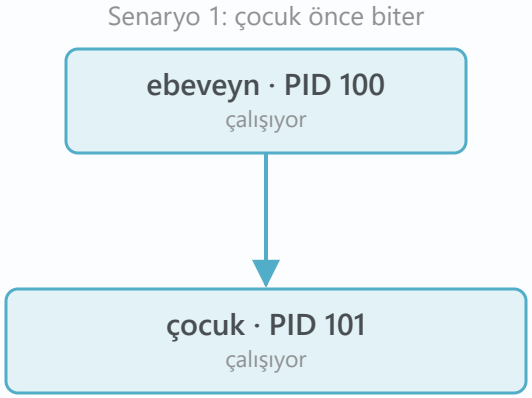
Sinyal	No	Anlamı
SIGTERM	15	kibarca bitir; süreç yakalayıp temizlik yapabilir (`kill` 'in varsayılanı)
SIGKILL	9	zorla bitir; yakalanamaz, yok sayılamaz, temizlik yapılamaz
SIGINT	2	terminalde `Ctrl+C`
SIGTSTP	20	terminalde `Ctrl+Z` (duraklat); `fg` / `bg` sürdürür
SIGSTOP / SIGCONT	19 / 18	durdur (yakalanamaz) / sürdür

Numaralar x86/ARM Linux içindir; taşınabilir kod numaraları değil adları kullanır.

```
$ ps -e | grep yaz
 1234 ?  00:00:03 yaz
$ kill 1234       # SIGTERM
$ kill -9 1234    # SIGKILL
$ pkill yaz       # ada göre
$ killall yaz
```

- Önce SIGTERM, yanıt yoksa SIGKILL.
- Sinyal göndermek için aynı kullanıcı ya da yetkili (root) olmak gerekir.

Zombi ve Yetim Süreçler



Ebeveyn (100) ve çocuk (101) çalışıyor.

Adım 1 / 6

Zombi kaynak harcamaz ama PID tüketir; `wait` çağırmayan uzun ömürlü bir sunucu binlerce zombi biriktirip PID'leri tüketebilir. Çözüm: `wait` / `waitpid` ya da SIGCHLD işleyicisi.

SÜREÇ SONLANDIRMA

Zombi ve Yetim Süreçler

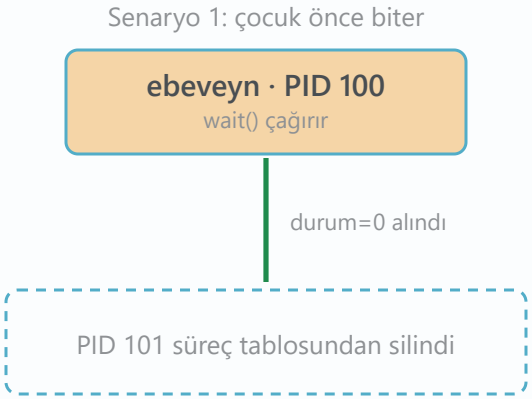


Çocuk `exit()` ile bitti ama ebeveyn henüz `wait()` çağırmadı: çocuk **zombi**dir. Belleği serbest; süreç tablosunda PID ve çıkış durumu bekler (`ps` 'te `Z` / <defunct>).

Adım 2 / 6

Zombi kaynak harcamaz ama PID tüketir; `wait` çağırmayan uzun ömürlü bir sunucu binlerce zombi biriktirip PID'leri tüketebilir. Çözüm: `wait` / `waitpid` ya da SIGCHLD işleyicisi.

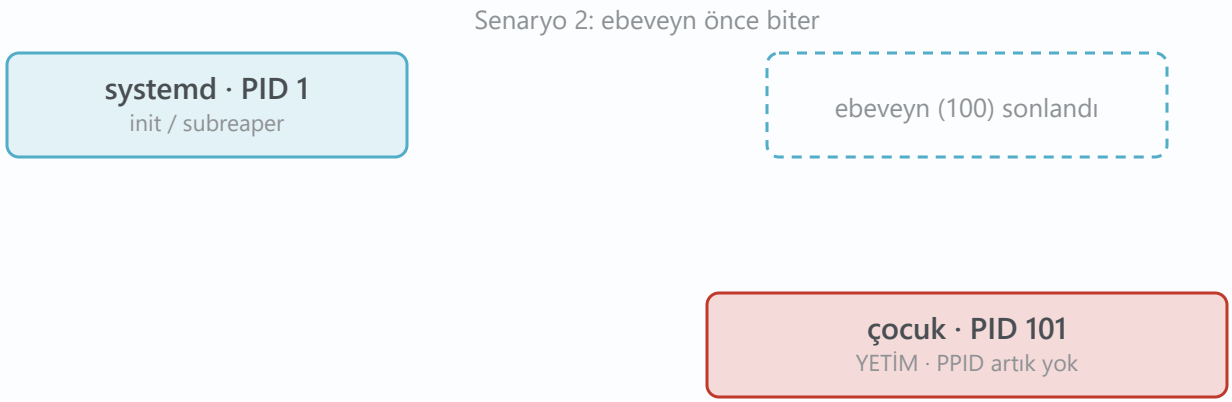
Zombi ve Yetim Süreçler



Ebeveyn `wait()` çağırır, çıkış durumunu alır; çekirdek girdiyi siler (**biçme**, reaping). Zombi kalmaz.

Zombi kaynak harcamaz ama PID tüketir; `wait` çağırmayan uzun ömürlü bir sunucu binlerce zombi biriktirip PID'leri tüketebilir. Çözüm: `wait` / `waitpid` ya da SIGCHLD işleyicisi.

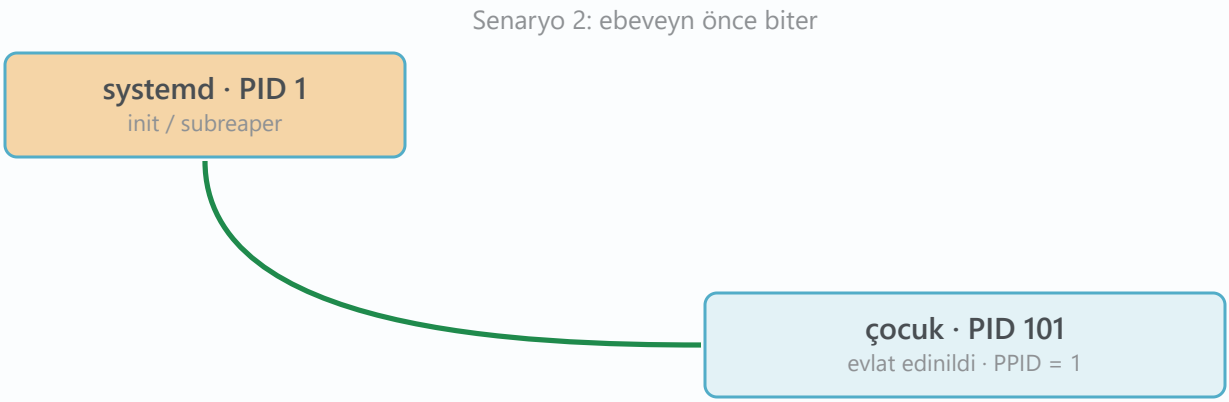
Zombi ve Yetim Süreçler



Senaryo 2: ebeveyn (100) çocuktan önce sonlandı. Çocuk (101) hâlâ çalışıyor: **yetim** süreç.

Zombi kaynak harcamaz ama PID tüketir; `wait` çağırmayan uzun ömürlü bir sunucu binlerce zombi biriktirip PID'leri tüketebilir. Çözüm: `wait` / `waitpid` ya da SIGCHLD işleyicisi.

Zombi ve Yetim Süreçler

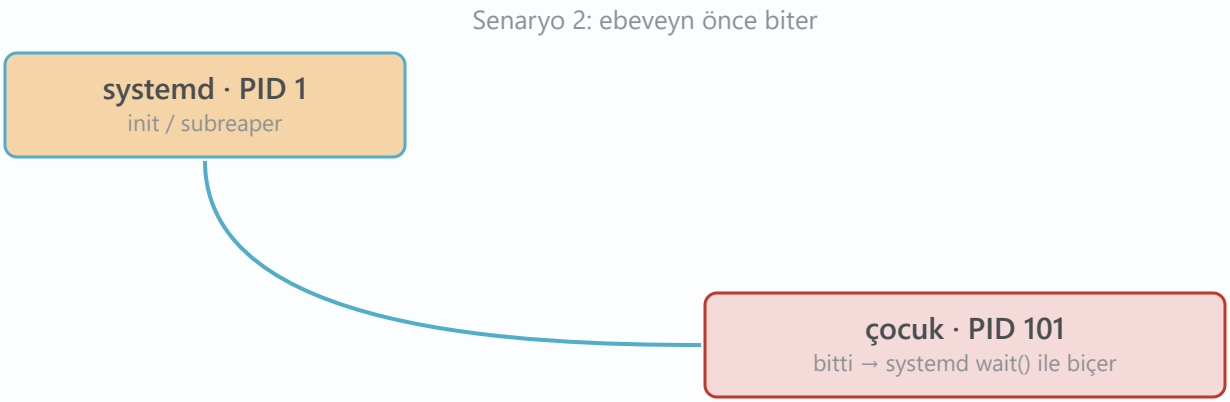


Çekirdek yetimi `init` / `systemd`'ye (PID 1) ya da tanımlı bir alt-biçici (subreaper) sürece bağlar; `getppid()` artık 1 döndürür.

Adım 5 / 6

Zombi kaynak harcamaz ama PID tüketir; `wait` çağırmayan uzun ömürlü bir sunucu binlerce zombi biriktirip PID'leri tüketebilir. Çözüm: `wait` / `waitpid` ya da SIGCHLD işleyicisi.

Zombi ve Yetim Süreçler

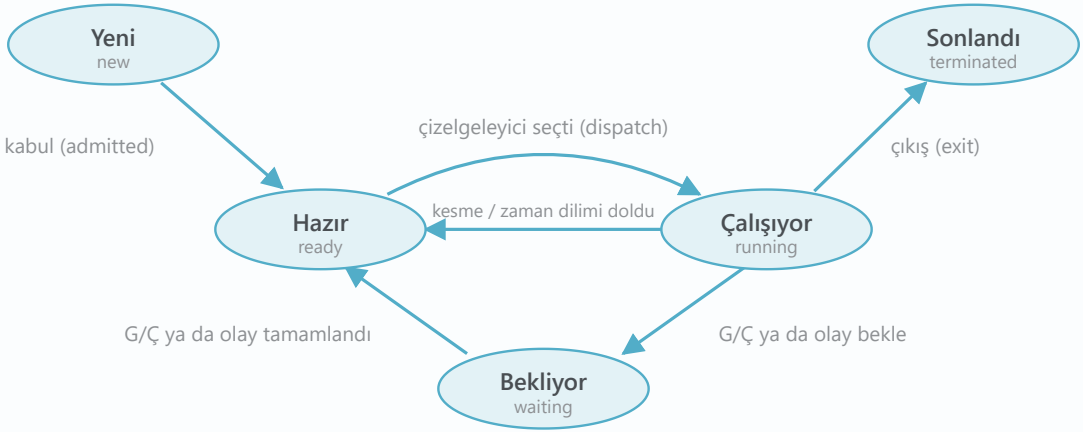


Yetim bitince yeni ebeveyni onu `wait()` ile biçer; böylece zombi birikmez.

Zombi kaynak harcamaz ama PID tüketir; `wait` çağırmayan uzun ömürlü bir sunucu binlerce zombi biriktirip PID'leri tüketebilir. Çözüm: `wait` / `waitpid` ya da SIGCHLD işleyicisi.

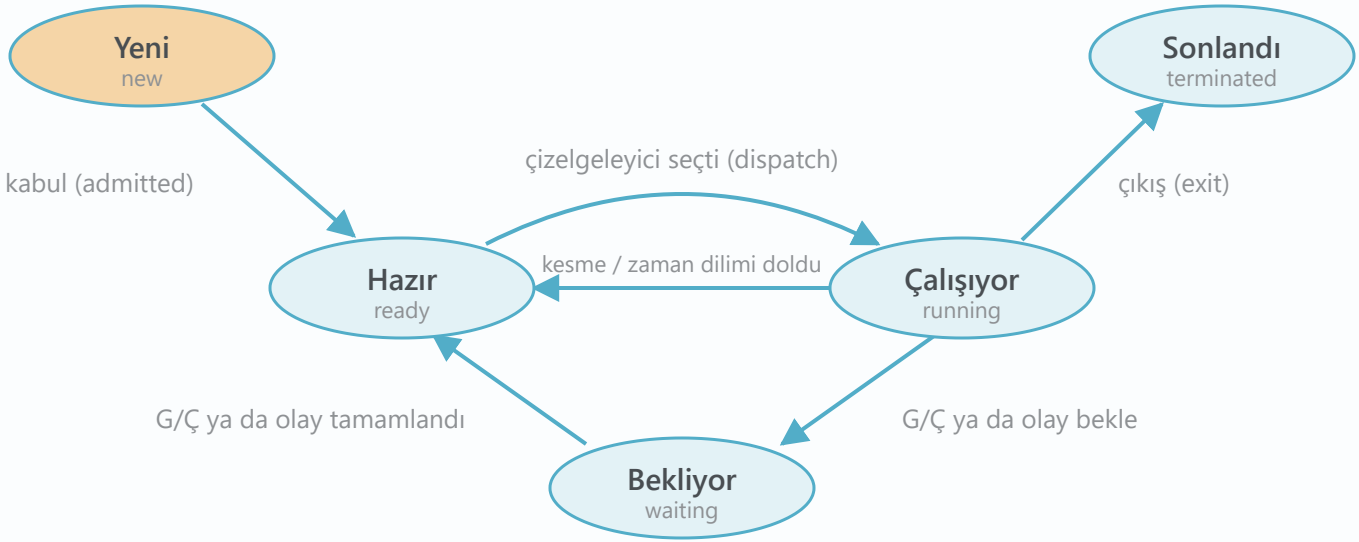
Süreç Durumları (Beş Durum Modeli)

Yeni	Hazır	Çalışıyor	Bekliyor	Sonlandı
new: oluşturuluyor	ready: CPU'yu bekliyor	running: komutlar yürütülüyor	waiting/blocked: bir olayı bekliyor	terminated: yürütme bitti



Bir çekirdekte aynı anda yalnız bir süreç *çalışıyor* olabilir; pek çoğu hazır ya da bekliyor durumundadır.

Durum Geçişleri Adım Adım

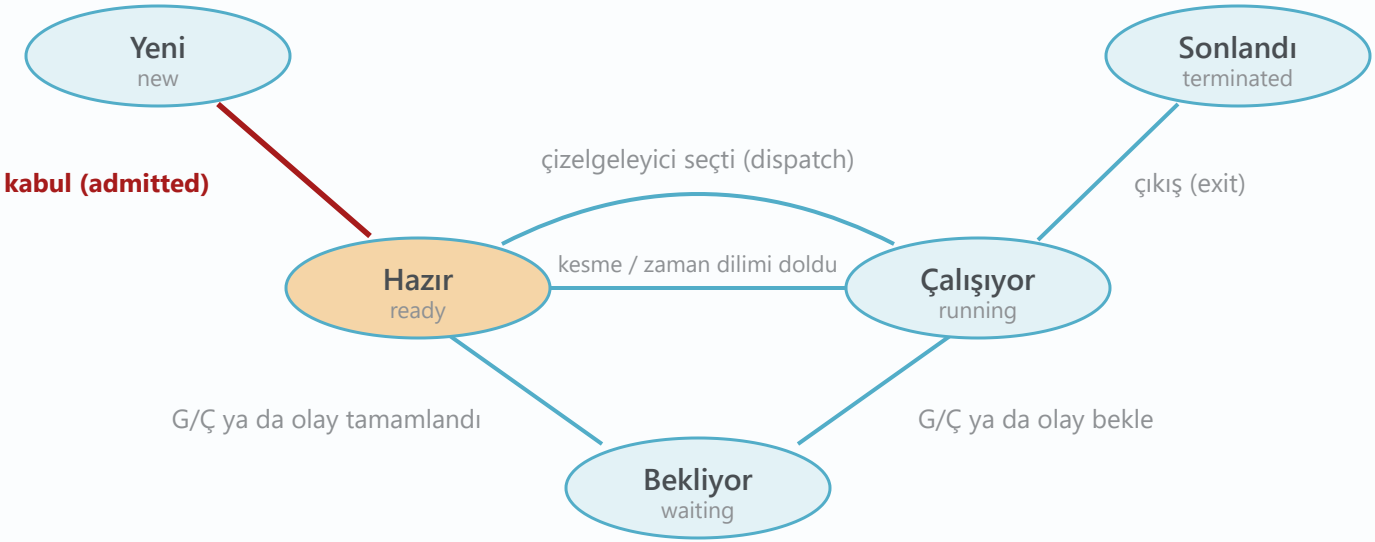


Kullanıcı programı başlattı: PCB oluşturuluyor, bellek ayrılıyor → **Yeni**.

Adım 1 / 9

Bekliyor → Çalışıyor geçişi **yoktur**: olay tamamlanan süreç önce hazır kuyruğuna girer.

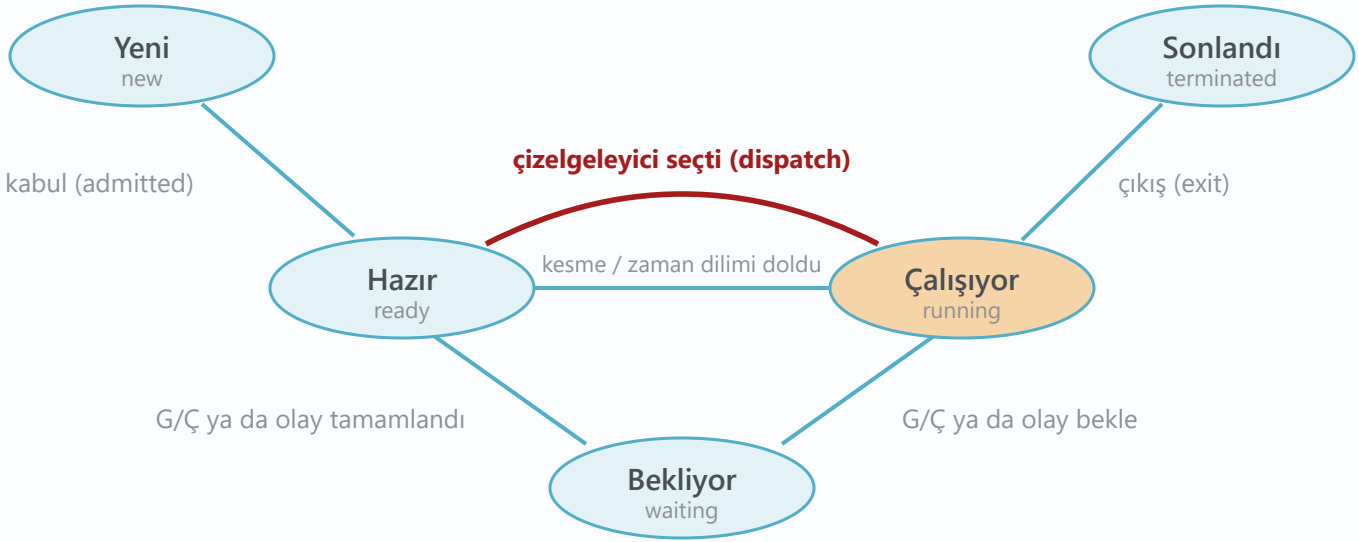
Durum Geçişleri Adım Adım



Çekirdek süreci kabul etti ve hazır kuyruğuna koydu → **Hazır**.

Bekliyor → Çalışıyor geçişi **yoktur**: olay tamamlanan süreç önce hazır kuyruğuna girer.

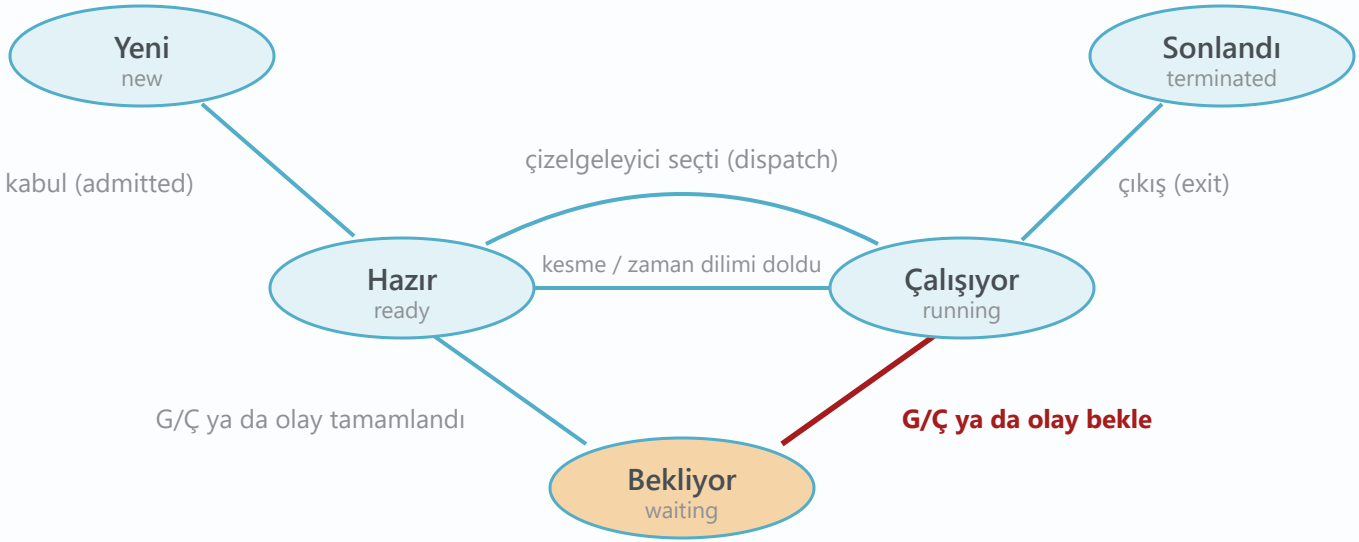
Durum Geçişleri Adım Adım



Çizelgeleyici bu süreci seçti, bağlamı yüklendi → **Çalışıyor**.

Bekliyor → Çalışıyor geçişi **yoktur**: olay tamamlanan süreç önce hazır kuyruğuna girer.

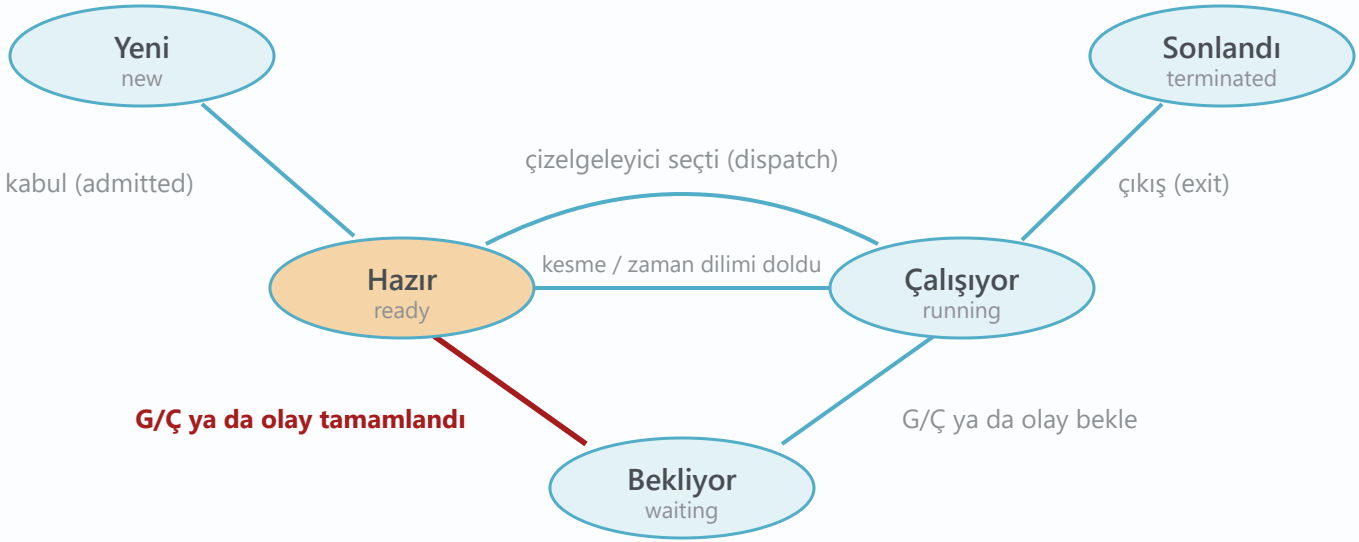
Durum Geçişleri Adım Adım



Süreç diskten `read()` istedi; veri gelene kadar CPU'yu bırakır → **Bekliyor**.

Bekliyor → Çalışıyor geçişi **yoktur**: olay tamamlanan süreç önce hazır kuyruğuna girer.

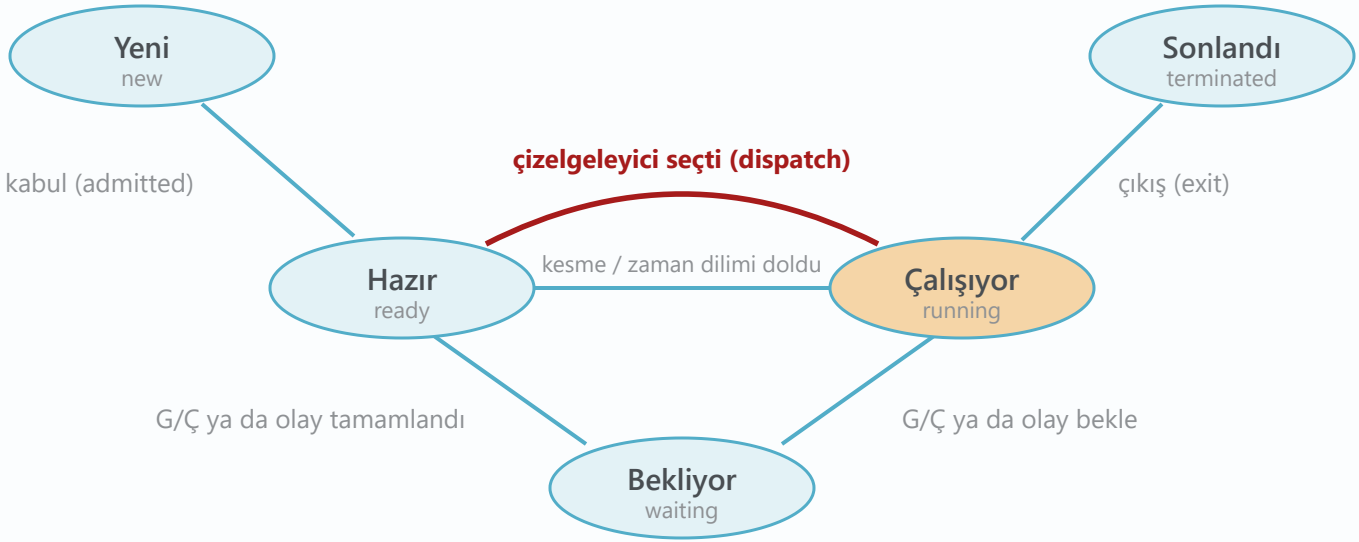
Durum Geçişleri Adım Adım



Disk kesmesi: veri hazır. Süreç doğrudan çalışmaz, hazır kuyruğuna döner → **Hazır**.

Bekliyor → Çalışıyor geçişi **yoktur**: olay tamamlanan süreç önce hazır kuyruğuna girer.

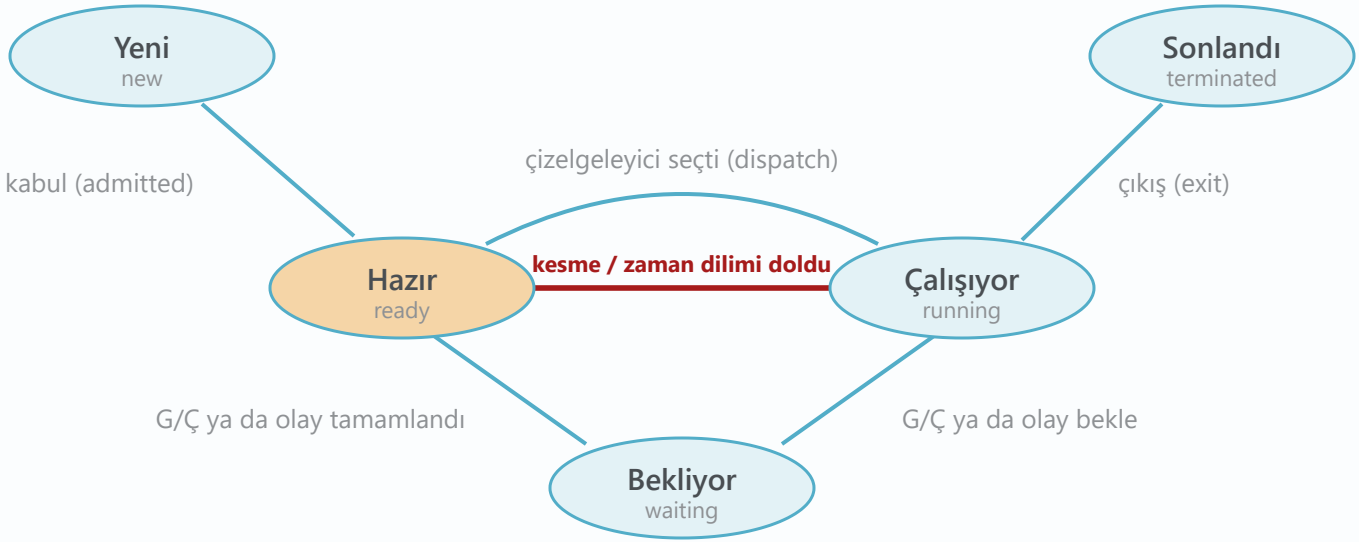
Durum Geçişleri Adım Adım



Yeniden seçildi → **Çalışıyor**.

Bekliyor → Çalışıyor geçişi **yoktur**: olay tamamlanan süreç önce hazır kuyruğuna girer.

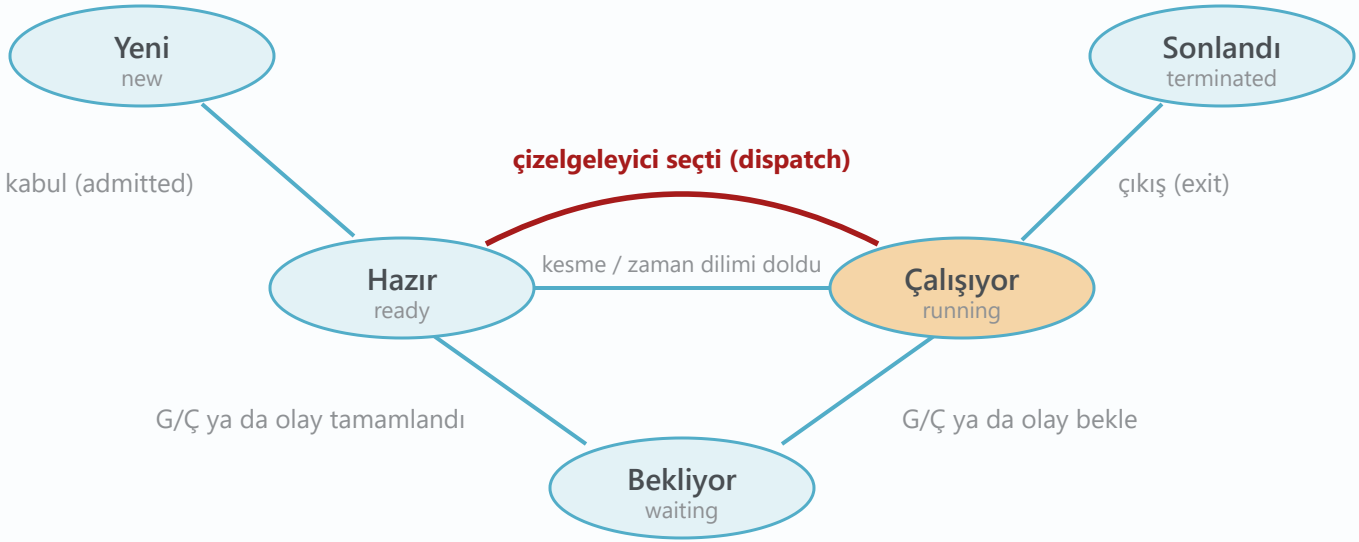
Durum Geçişleri Adım Adım



Zamanlayıcı kesmesi: zaman dilimi doldu (kesmeli çizelgeleme) → **Hazır**.

Bekliyor → Çalışıyor geçişi **yoktur**: olay tamamlanan süreç önce hazır kuyruğuna girer.

Durum Geçişleri Adım Adım



Tekrar seçildi → **Çalışıyor**.

Bekliyor → Çalışıyor geçişi **yoktur**: olay tamamlanan süreç önce hazır kuyruğuna girer.

Durum Geçişleri Adım Adım



`exit()` çağırdı → **Sonlandı**; ebeveyn `wait()` edene kadar zombi.

Bekliyor → Çalışıyor geçişi **yoktur**: olay tamamlanan süreç önce hazır kuyruğuna girer.

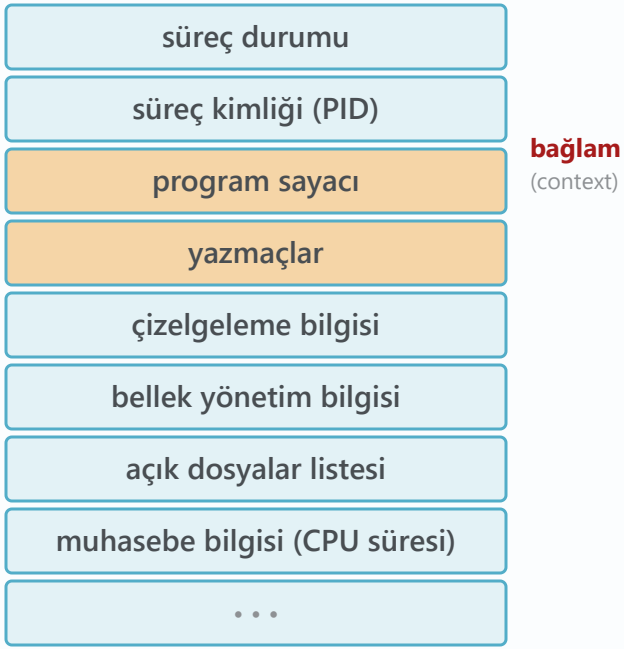
Linux'ta Süreç Durumları

ps STAT	Anlam	Modeldeki karşılığı
`R`	çalışıyor ya da çalışmaya hazır	Çalışıyor + Hazır
`S`	kesilebilir uyku (olay bekliyor)	Bekliyor
`D`	kesilemez uyku (genelde disk G/Ç)	Bekliyor
`T`	durdurulmuş (SIGSTOP, hata ayıklayıcı)	askıya alınmış
`Z`	zombi	Sonlandı (biçilmedi)
`I`	boşta çekirdek iş parçacığı	—

```
$ ps -eo pid,stat,comm
  PID STAT COMMAND
    1 Ss   systemd
  890 Ss   sshd
 2105 Ss   bash
 2241 R+   ps
 2307 Z    yaz <defunct>
```

Ek harfler: `s` oturum lideri, `+` ön plan grubu. Linux hazır ile çalışıyor durumunu ayırmaz; ikisi de `R`.

Süreç Kontrol Bloğu (PCB)



- Çekirdek her süreç için bir **PCB** (process control block) tutar; sürece dair bilinmesi gereken her şey buradadır.
- Tüm PCB'lerin oluşturduğu yapıya **süreç tablosu** denir (Tanenbaum'un tablosu bu girdilerin alanlarını listeler).
- Süreç CPU'dan ayrıldığında program sayacı ve yazmaçlar PCB'ye kaydedilir; geri döndüğünde buradan yüklenir. Bu bilgiye **bağlam** denir.
- PCB'ler çekirdek belleğinde durur; kullanıcı süreçleri doğrudan erişemez.

PCB Alanları (Tanenbaum'a göre)

Süreç yönetimi	Bellek yönetimi	Dosya yönetimi
yazmaçlar · program sayacı · program durum sözcüğü (PSW) · yığın işaretçisi · süreç durumu · öncelik · çizelgeleme parametreleri · PID · ebeveyn süreç · süreç grubu · sinyaller · başlama zamanı · kullanılan CPU zamanı · çocukların CPU zamanı · sonraki alarm zamanı	metin, veri ve yığın bölümlerinin (segment) işaretçileri; günümüzde sayfa tablosu işaretçisi ve bellek bölgeleri listesi	kök dizin · çalışma dizini · dosya tanımlayıcıları · kullanıcı kimliği (UID) · grup kimliği (GID)

PSW

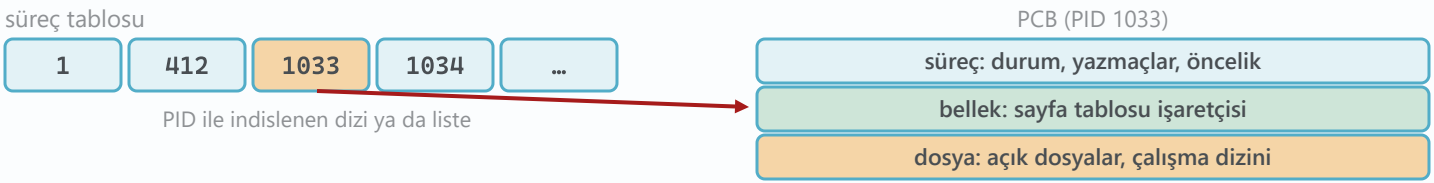
Durum bayrakları (sıfır, taşma...), çalışma modu (kullanıcı/çekirdek), kesme izinleri.

Süreç grubu

Birlikte yönetilen süreçler: kabuktaki bir boru hattı tek grup olur, Ctrl+C hepsine gider.

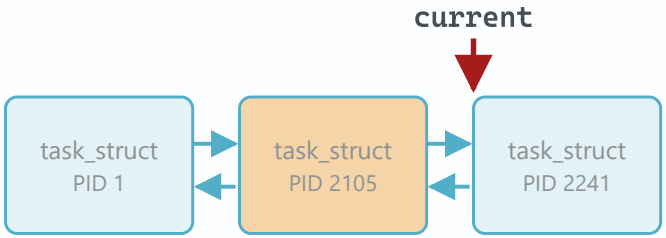
Sonraki alarm

`alarm()` ile istenen SIGALRM'nin teslim zamanı.



Linux'ta PCB: task_struct

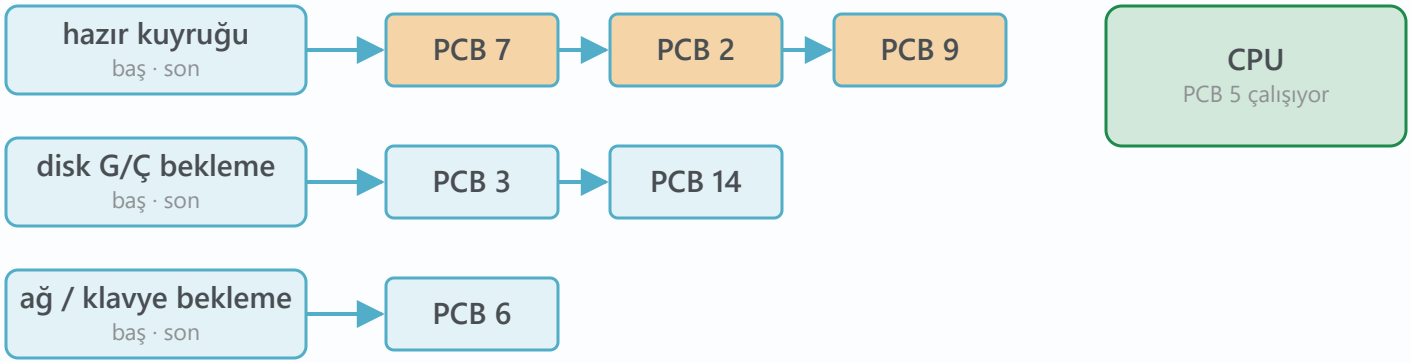
```
struct task_struct {          /* <linux/sched.h> */
    pid_t pid;                    /* süreç kimliği */
    unsigned int __state;         /* süreç durumu */
    int prio;                     /* öncelik */
    struct sched_entity se;       /* CFS/EEVDF bilgisi */
    struct task_struct *parent;   /* ebeveyn süreç */
    struct list_head children;    /* çocuklar */
    struct files_struct *files;   /* açık dosyalar */
    struct mm_struct *mm;         /* adres uzayı */
    /* ... yüzlerce alan daha ... */
};
```



tasks: çift yönlü bağlı liste

- Tüm görevler çift yönlü bağlı listededir; `current` o çekirdekte çalışan görevi gösterir.
- Eski çekirdeklerdeki `long state` / `time_slice` alanları güncel çekirdeklerde `__state` ve `sched_entity` içine taşınmıştır.

Hazır ve Bekleme Kuyrukları

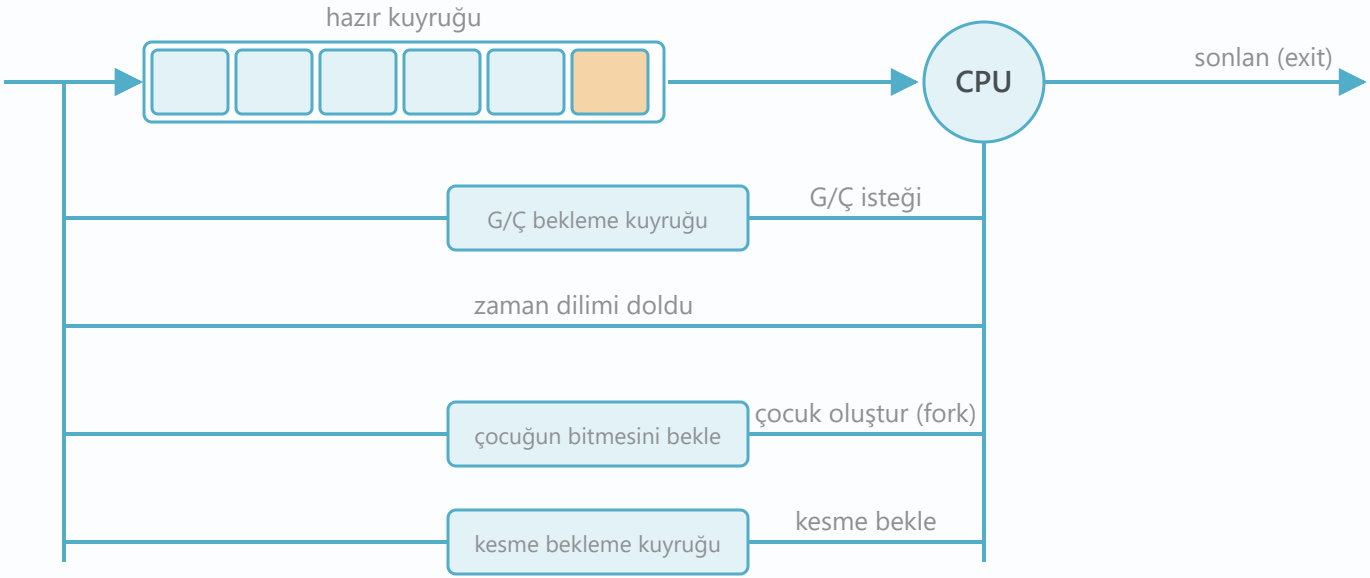


kuyruklar PCB'lere bağlı listeler olarak tutulur

- **Çizelgeleyici** (scheduler) hazır süreçler arasından CPU'ya geçecek olanı seçer; amaç CPU'yu meşgul tutmak ve kullanıcıya hızlı yanıt vermektir.
- **Hazır kuyruğu:** bellekte olup yürütülmeyi bekleyen süreçler. **Bekleme kuyrukları:** belirli bir olayı (G/Ç bitişi, çocuğun sonlanması...) bekleyenler; her olay türü için ayrı kuyruk vardır.
- Linux her çekirdeğe ayrı bir çalışma kuyruğu (runqueue) atar; CFS/EEVDF bunu kırmızı-siyah ağaçla tutar.

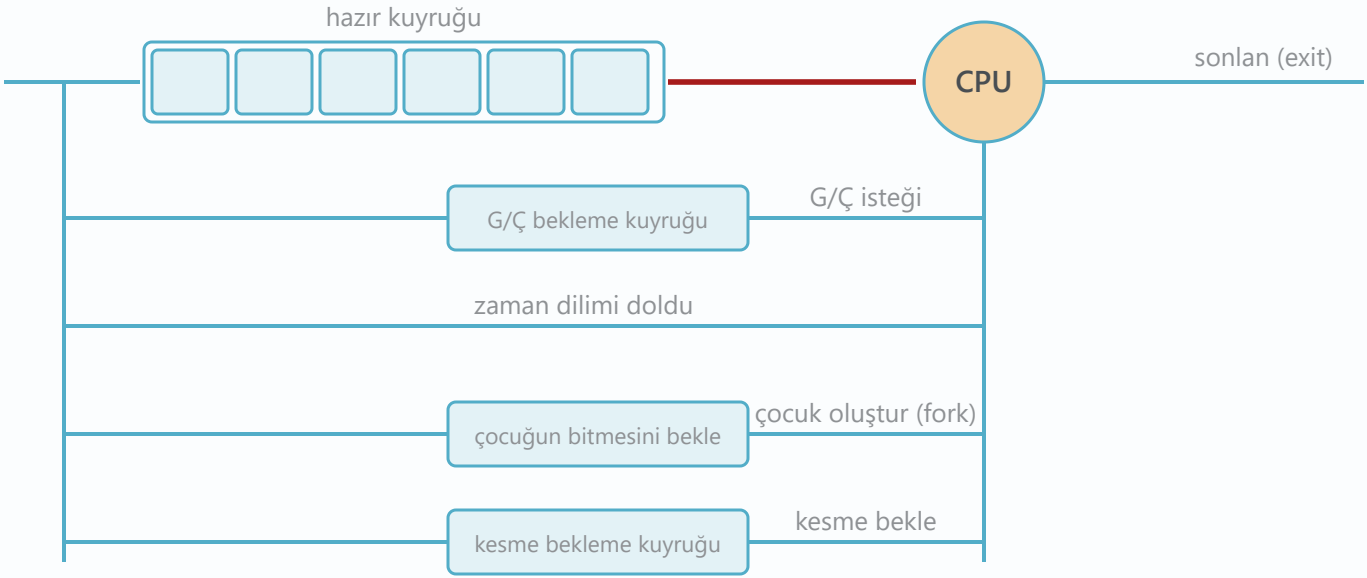
ÇİZELGELEME

Kuyruk Diyagramı: Bir Sürecin Yolculuğu



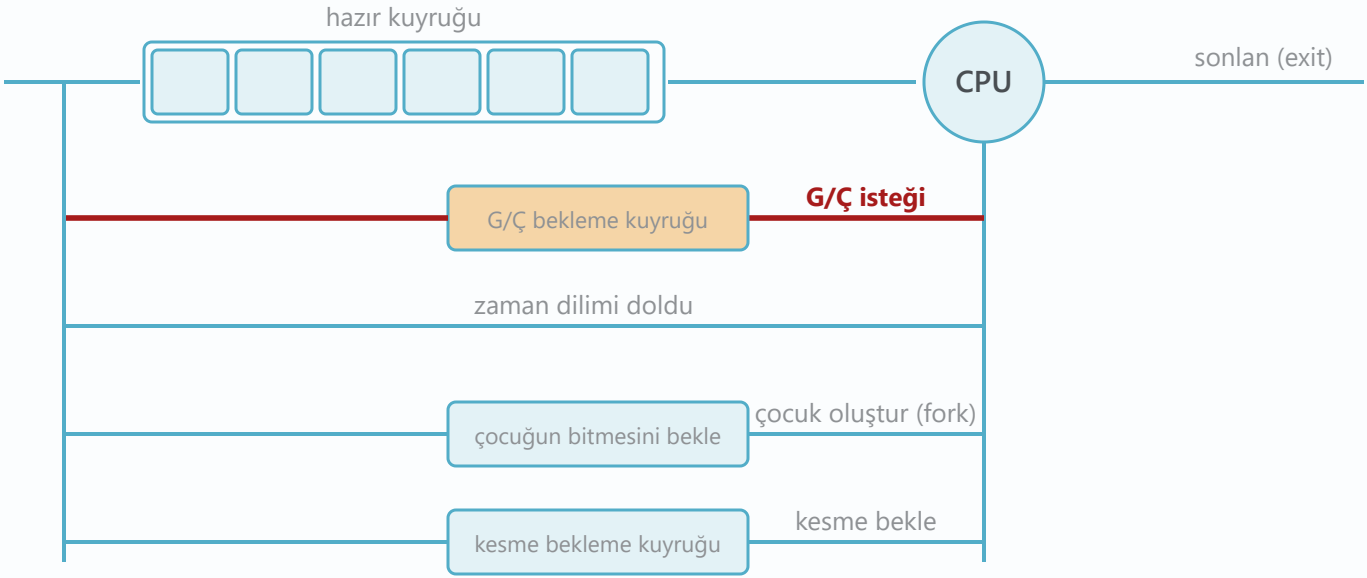
Yeni süreç hazır kuyruğunun sonuna eklenir ve sırasını bekler.

Kuyruk Diyagramı: Bir Sürecin Yolculuğu



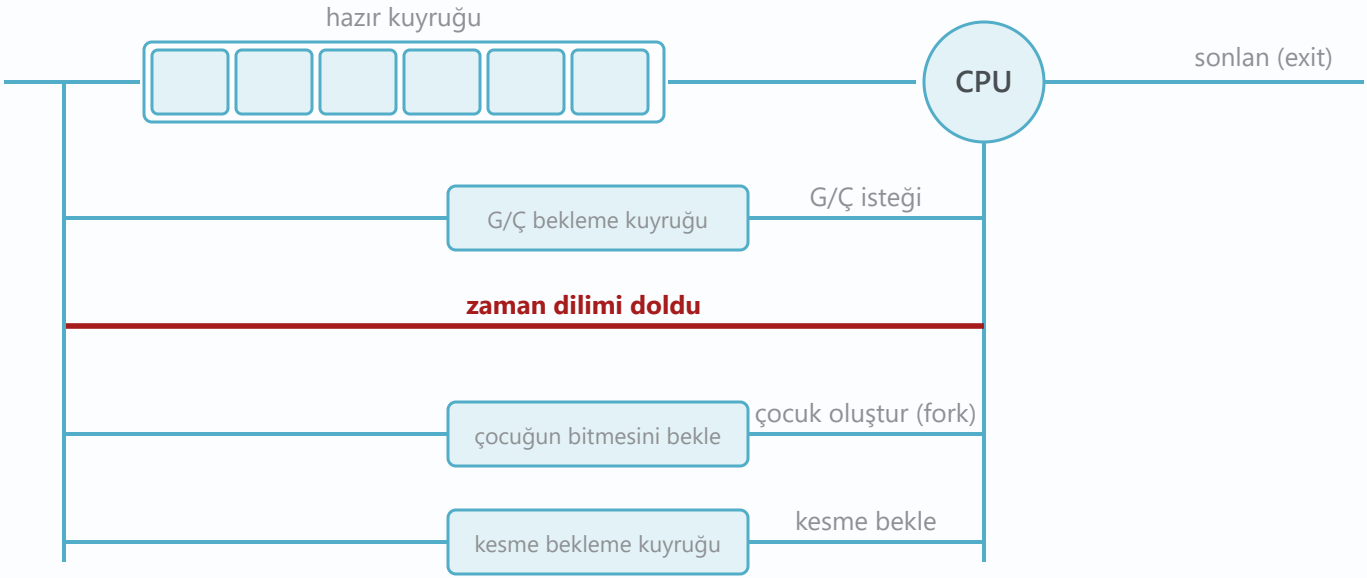
Çizelgeleyici onu seçer (dispatch): süreç CPU'da çalışır.

Kuyruk Diyagramı: Bir Sürecin Yolculuğu



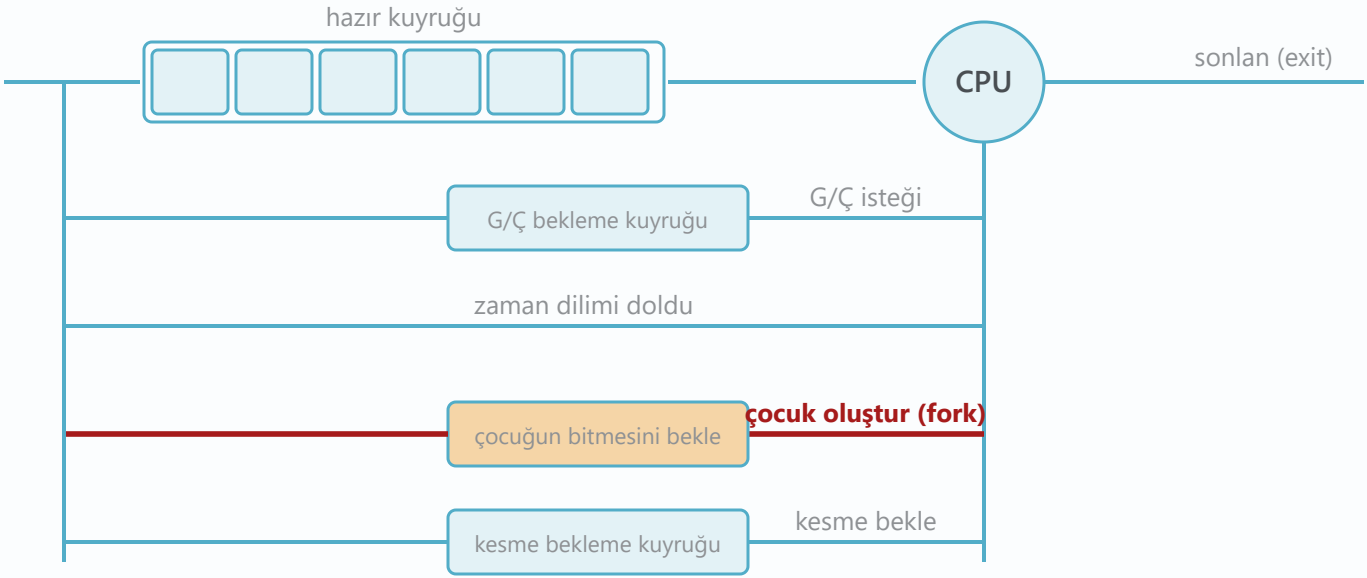
G/Ç isteği yaptı: G/Ç bekleme kuyruğuna geçer; G/Ç bitince hazır kuyruğuna döner.

Kuyruk Diyagramı: Bir Sürecin Yolculuğu



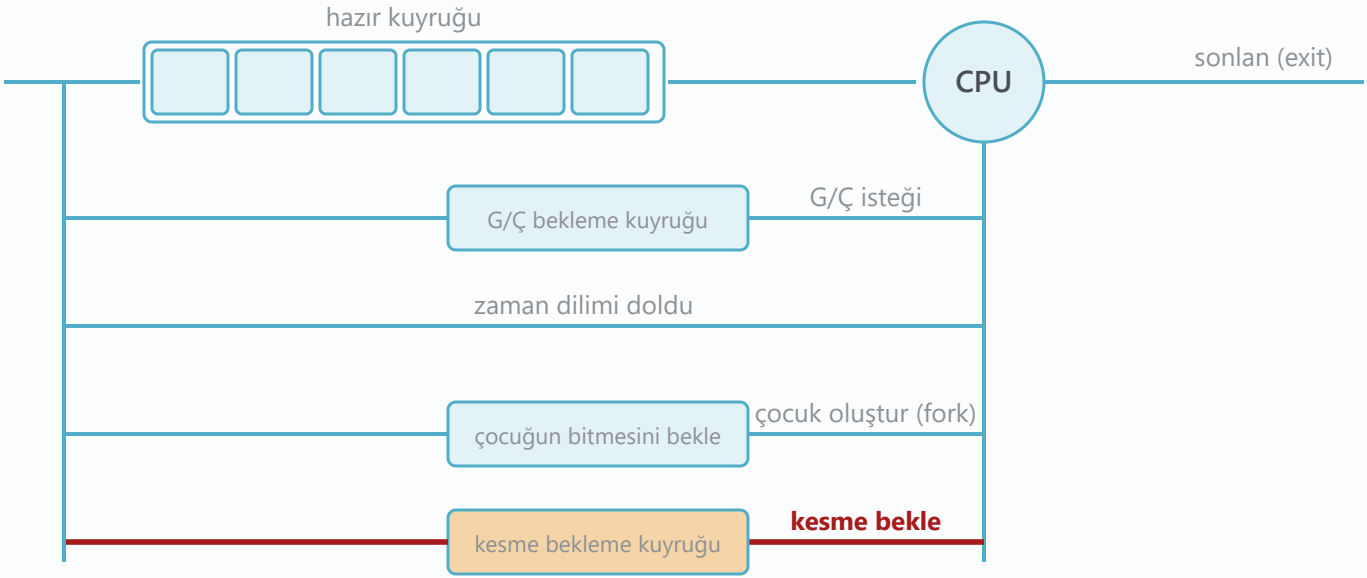
Zaman dilimi doldu: zamanlayıcı kesmesiyle CPU'dan alınır, doğrudan hazır kuyruğuna döner.

Kuyruk Diyagramı: Bir Sürecin Yolculuğu



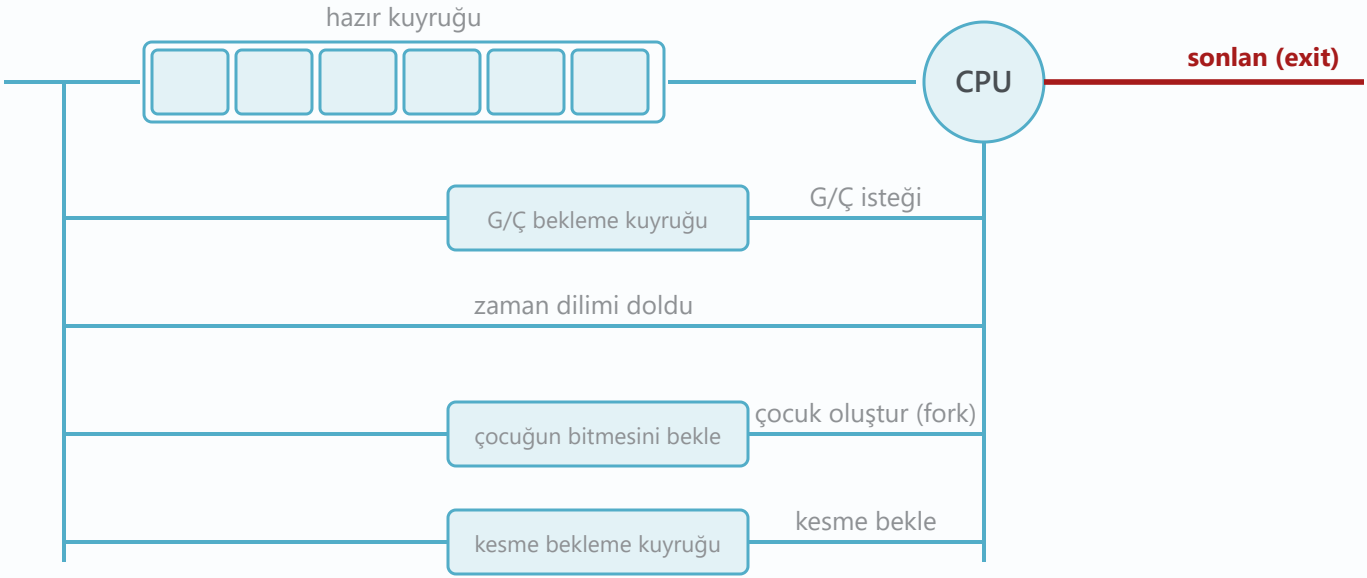
Çocuk oluşturup `wait()` çağırdı: çocuk bitene kadar bekler.

Kuyruk Diyagramı: Bir Sürecin Yolculuğu



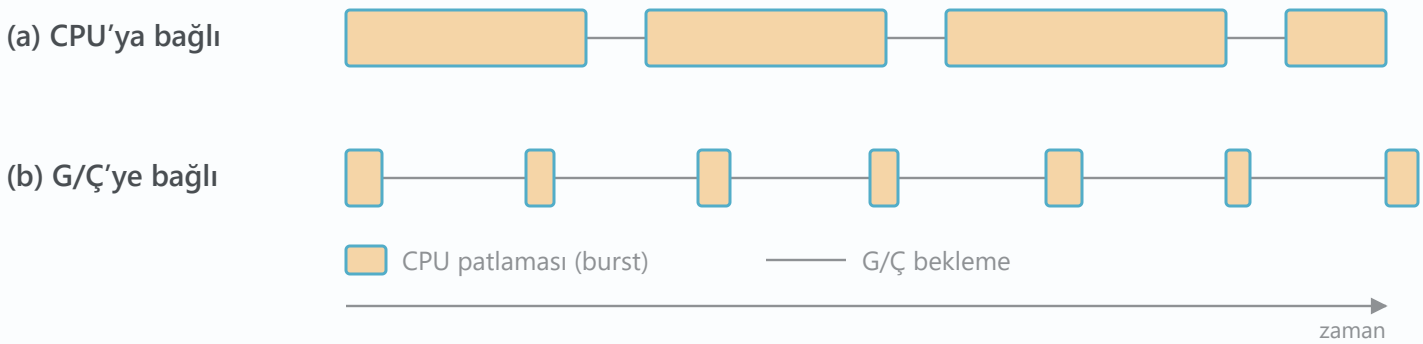
Bir kesmeyi/olayı bekliyor (örn. `sleep()`, ağ paketi).

Kuyruk Diyagramı: Bir Sürecin Yolculuğu



İşi bitti: `exit()` ile sistemden çıkar; PCB'si ebeveyn `wait` edince silinir.

CPU’ya Bağlı ve G/Ç’ye Bağlı Süreçler



CPU’ya bağlı (CPU-bound)

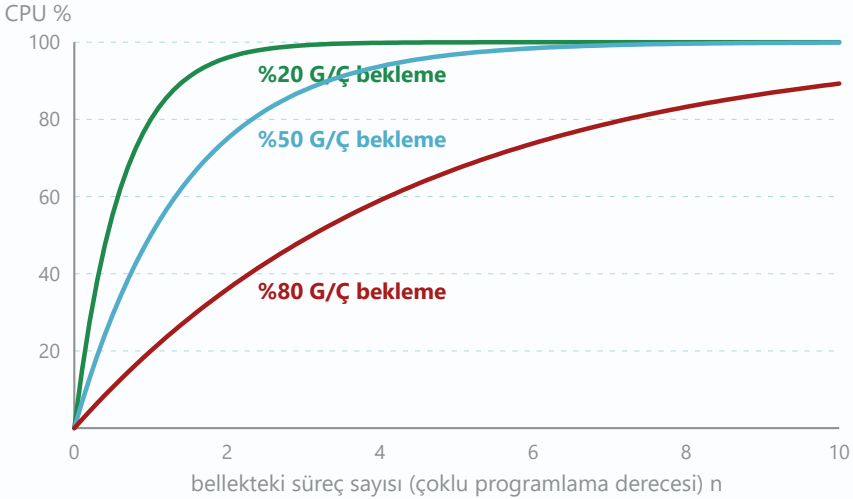
Uzun hesaplama patlamaları, seyrek G/Ç: derleme, video kodlama, bilimsel hesap.

G/Ç’ye bağlı (I/O-bound)

Kısa CPU patlamaları, sık bekleme: editör, web sunucusu, veritabanı istemcisi.

- CPU’lar disklerden çok daha hızlı geliştiği için süreçler giderek **G/Ç’ye bağlı** hâle gelir.
- G/Ç’ye bağlı süreç hazır olunca hemen CPU almalı (hızlı bağlam değiştirme); iki tür dengeli karıştırılmalıdır.

Çoklu Programlamanın Modellenmesi



Bir süreç zamanının **p** kesrini G/Ç bekleyerek geçirsin. Bellekte **n** bağımsız süreç varsa hepsinin aynı anda beklemesi olasılığı $\mathbf{p^n}$'dir:

$$\textbf{CPU kullanımı} = 1 - p^n$$

Örn. p = 0,8: n = 1 → %20, n = 4 → %59, n = 10 → %89.

Kaba bir model: süreçlerin bağımsız olduğu varsayılır.

Kod İzleme: CPU Kullanımı 1 – p^n

```
▶ double p = 0.8;      /* G/Ç oranı */
▶ double pn = 1.0;
3 for (int n = 1; n <= 5; n++) {
4   pn = pn * p;       /* hepsi bekler */
5   double kul = 1 - pn;
6   printf("n=%d %%%.0f\n", n, kul * 100);
7 }
```



Her süreç zamanının **%80**'ini G/Ç bekleyerek geçiriyor. CPU yalnızca bellekteki süreçlerin **hepsi** aynı anda beklerken boştur.

Adım 1 / 11

DEĞİŞKENLER

`p = 0.8` `pn = 1.0`

Kod İzleme: CPU Kullanımı 1 – p^n

```
1 double p = 0.8;      /* G/Ç oranı */
2 double pn = 1.0;
▶ for (int n = 1; n <= 5; n++) {
▶   pn = pn * p;       /* hepsi bekler */
5   double kul = 1 - pn;
6   printf("n=%d %%%.0f\n", n, kul * 100);
7 }
```



n = 1: tek süreç; CPU'nun boş kalma olasılığı **p^n = 0,8**.

Adım 2 / 11

DEĞİŞKENLER

`p = 0.8` `pn = 0.8` `n = 1`

Kod İzleme: CPU Kullanımı 1 – p^n

```
1 double p = 0.8;       /* G/Ç oranı */
2 double pn = 1.0;
3 for (int n = 1; n <= 5; n++) {
4   pn = pn * p;        /* hepsi bekler */
▶   double kul = 1 - pn;
▶   printf("n=%d %%%.0f\n", n, kul * 100);
7 }
```



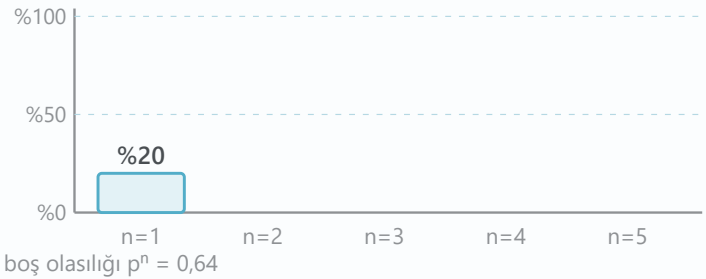
kul = 1 − 0,8 = **%20**: tek programlı sistemde CPU zamanın beşte dördünde boş.

DEĞİŞKENLER

`p = 0.8` `pn = 0.8` `n = 1` `kul = 0.2`

Kod İzleme: CPU Kullanımı 1 – p^n

```
1 double p = 0.8;      /* G/Ç oranı */
2 double pn = 1.0;
▶ for (int n = 1; n <= 5; n++) {
▶   pn = pn * p;       /* hepsi bekler */
5   double kul = 1 - pn;
6   printf("n=%d %%%.0f\n", n, kul * 100);
7 }
```



n = 2: hepsinin birden beklemesi olasılığı $0{,}8^2$ = **0,64**.

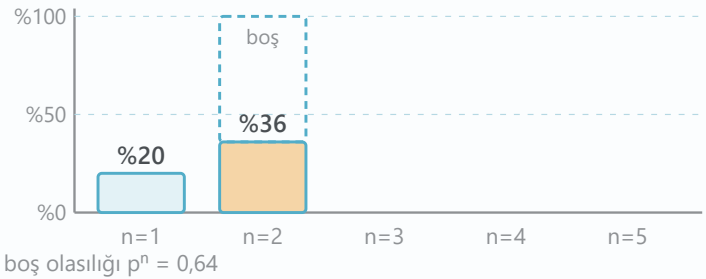
Adım 4 / 11

DEĞİŞKENLER

`p = 0.8` `pn = 0.64` `n = 2`

Kod İzleme: CPU Kullanımı 1 – p^n

```
1 double p = 0.8;       /* G/Ç oranı */
2 double pn = 1.0;
3 for (int n = 1; n <= 5; n++) {
4   pn = pn * p;        /* hepsi bekler */
▶   double kul = 1 - pn;
▶   printf("n=%d %%%.0f\n", n, kul * 100);
7 }
```



boş olasılığı p^n = 0,64

kul = **%36**.

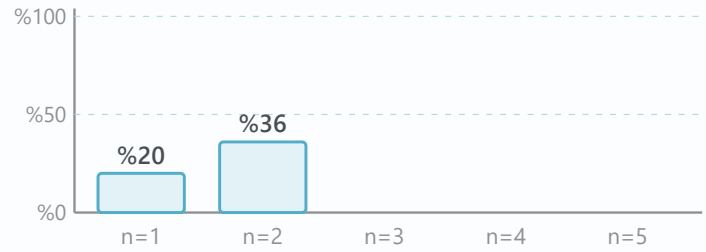
Adım 5 / 11

DEĞİŞKENLER

`p = 0.8` `pn = 0.64` `n = 2` `kul = 0.36`

Kod İzleme: CPU Kullanımı 1 – p^n

```
1 double p = 0.8;       /* G/Ç oranı */
2 double pn = 1.0;
▶ for (int n = 1; n <= 5; n++) {
▶   pn = pn * p;        /* hepsi bekler */
5   double kul = 1 - pn;
6   printf("n=%d %%%.0f\n", n, kul * 100);
7 }
```



boş olasılığı p^n = 0,512

n = 3: hepsinin birden beklemesi olasılığı $0{,}8^3$ = **0,512**.

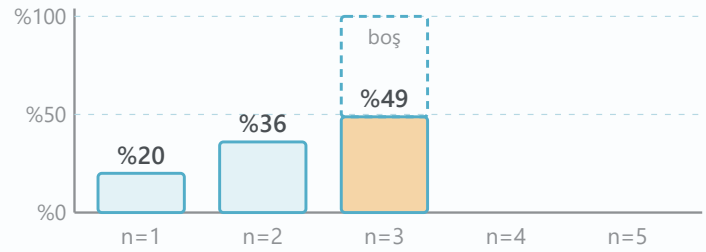
Adım 6 / 11

DEĞİŞKENLER

`p = 0.8` `pn = 0.512` `n = 3`

Kod İzleme: CPU Kullanımı 1 – p^n

```
1 double p = 0.8;       /* G/Ç oranı */
2 double pn = 1.0;
3 for (int n = 1; n <= 5; n++) {
4   pn = pn * p;        /* hepsi bekler */
▶   double kul = 1 - pn;
▶   printf("n=%d %%%.0f\n", n, kul * 100);
7 }
```



boş olasılığı p^n = 0,512

kul = **%48,8**.

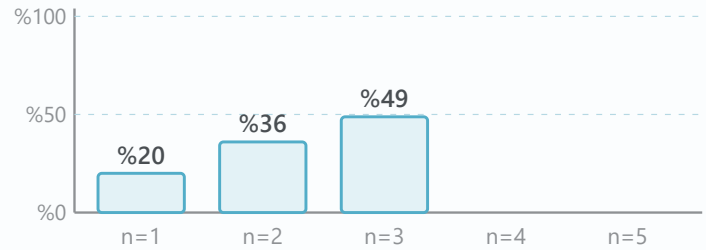
Adım 7 / 11

DEĞİŞKENLER

`p = 0.8` `pn = 0.512` `n = 3` `kul = 0.488`

Kod İzleme: CPU Kullanımı 1 – p^n

```
1 double p = 0.8;       /* G/Ç oranı */
2 double pn = 1.0;
▶ for (int n = 1; n <= 5; n++) {
▶   pn = pn * p;        /* hepsi bekler */
5   double kul = 1 - pn;
6   printf("n=%d %%%.0f\n", n, kul * 100);
7 }
```



boş olasılığı p^n = 0,4096

n = 4: hepsinin birden beklemesi olasılığı $0{,}8^4$ = **0,4096**.

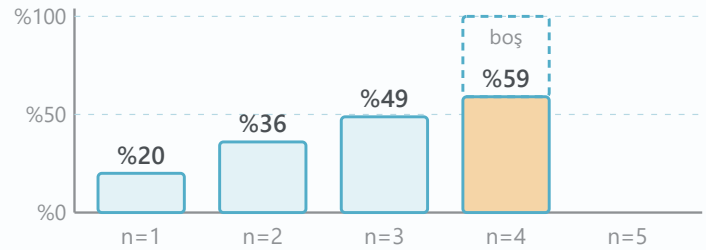
Adım 8 / 11

DEĞİŞKENLER

`p = 0.8` `pn = 0.4096` `n = 4`

Kod İzleme: CPU Kullanımı 1 – p^n

```
1 double p = 0.8;       /* G/Ç oranı */
2 double pn = 1.0;
3 for (int n = 1; n <= 5; n++) {
4   pn = pn * p;        /* hepsi bekler */
▶   double kul = 1 - pn;
▶   printf("n=%d %%%.0f\n", n, kul * 100);
7 }
```



boş olasılığı p^n = 0,4096

kul = **%59,04**.

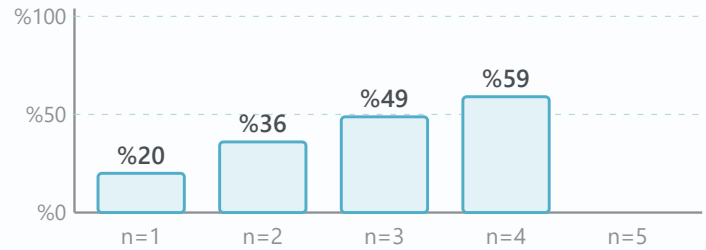
Adım 9 / 11

DEĞİŞKENLER

`p = 0.8` `pn = 0.4096` `n = 4` `kul = 0.5904`

Kod İzleme: CPU Kullanımı 1 – p^n

```
1 double p = 0.8;       /* G/Ç oranı */
2 double pn = 1.0;
▶ for (int n = 1; n <= 5; n++) {
▶   pn = pn * p;        /* hepsi bekler */
5   double kul = 1 - pn;
6   printf("n=%d %%%.0f\n", n, kul * 100);
7 }
```



boş olasılığı p^n = 0,3277

n = 5: hepsinin birden beklemesi olasılığı $0{,}8^5$ = **0,32768**.

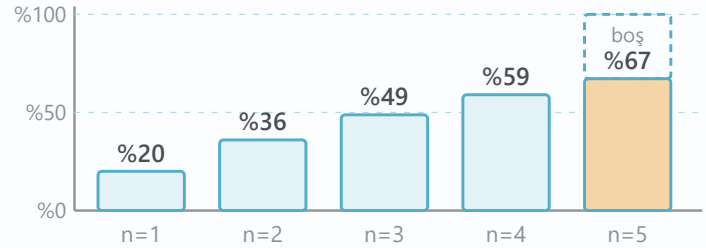
Adım 10 / 11

DEĞİŞKENLER

`p = 0.8` `pn = 0.32768` `n = 5`

Kod İzleme: CPU Kullanımı 1 – p^n

```
1 double p = 0.8;      /* G/Ç oranı */
2 double pn = 1.0;
3 for (int n = 1; n <= 5; n++) {
4   pn = pn * p;       /* hepsi bekler */
▶   double kul = 1 - pn;
▶   printf("n=%d %%%.0f\n", n, kul * 100);
7 }
```



boş olasılığı p^n = 0,3277

kul = **%67,23**. Her yeni süreç bir öncekinden **daha az** kazandırıyor: artış 8 puan.

Adım 11 / 11

DEĞİŞKENLER

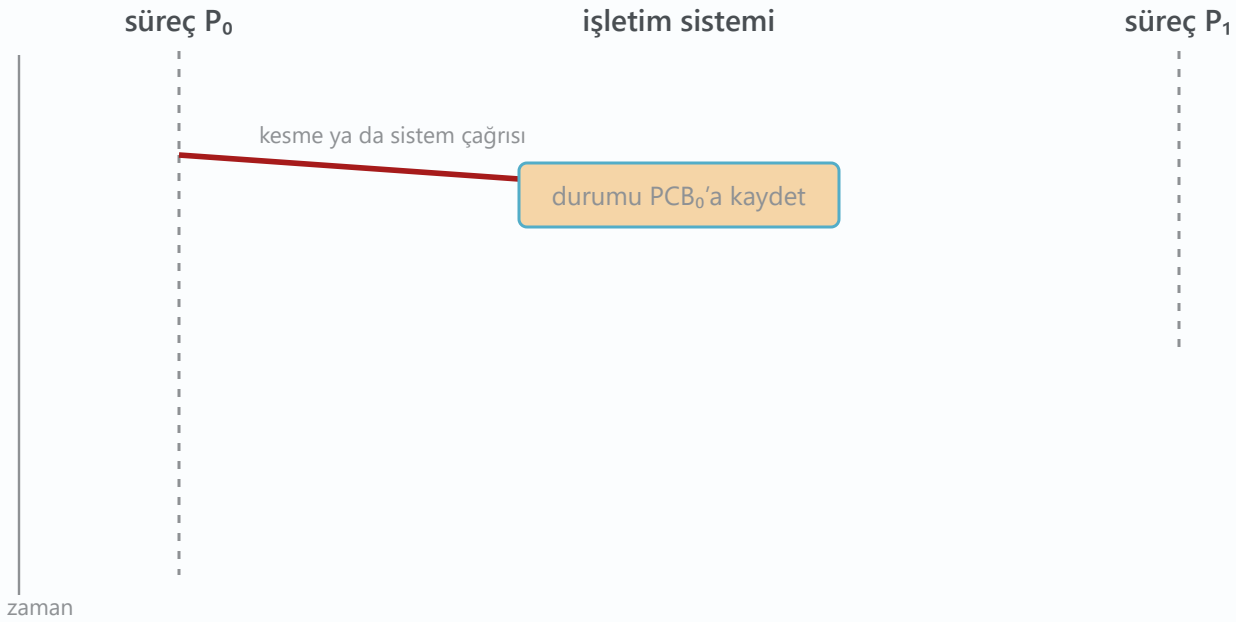
`p = 0.8` `pn = 0.32768` `n = 5` `kul = 0.67232`

Bağlam Değiştirme (Context Switch)



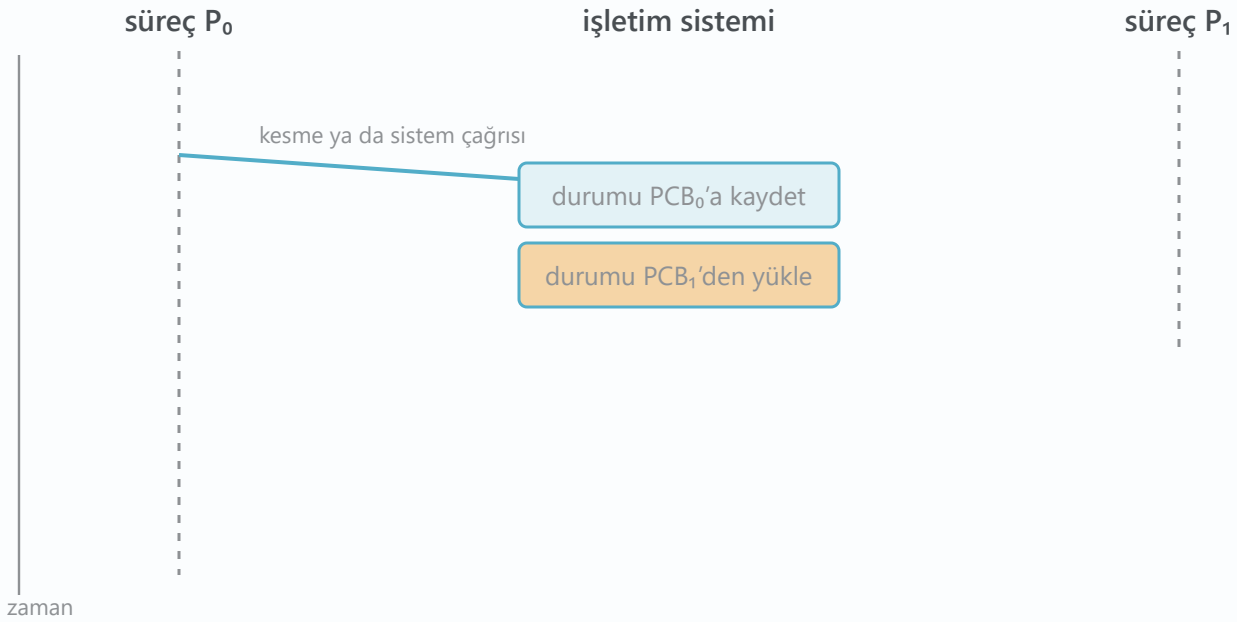
P_0 çalışıyor (kalın çizgi); P_1 hazır kuyruğunda bekliyor.

Bağlam Değiştirme (Context Switch)



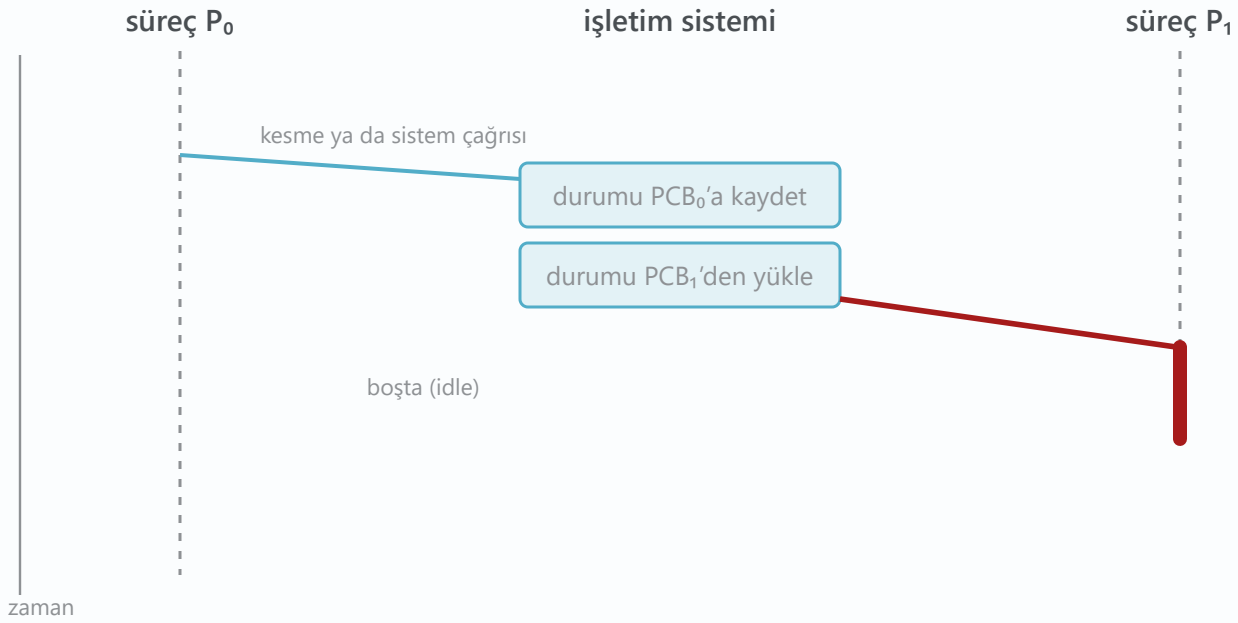
Kesme ya da sistem çağrısı: CPU çekirdek moduna geçer, P_0'ın program sayacı ve yazmaçları **PCB_0**'a kaydedilir.

Bağlam Değiştirme (Context Switch)



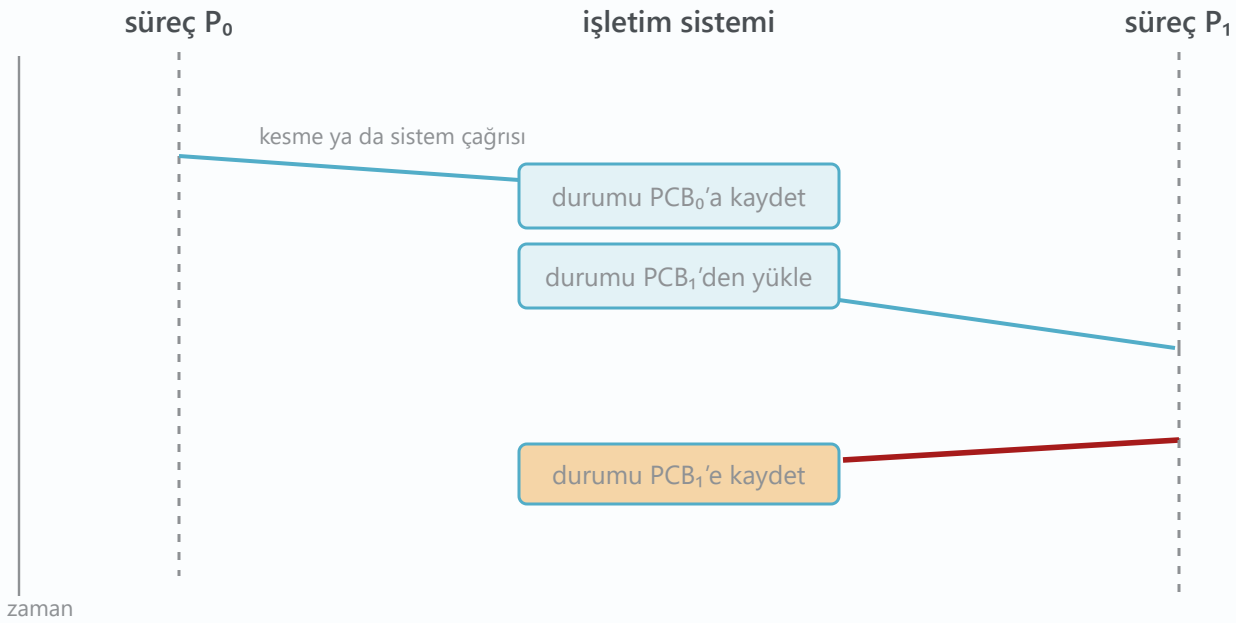
Çizelgeleyici P_1'i seçer; P_1'in bağlamı **PCB_1**'den yüklenir (adres alanı da değişir).

Bağlam Değiştirme (Context Switch)



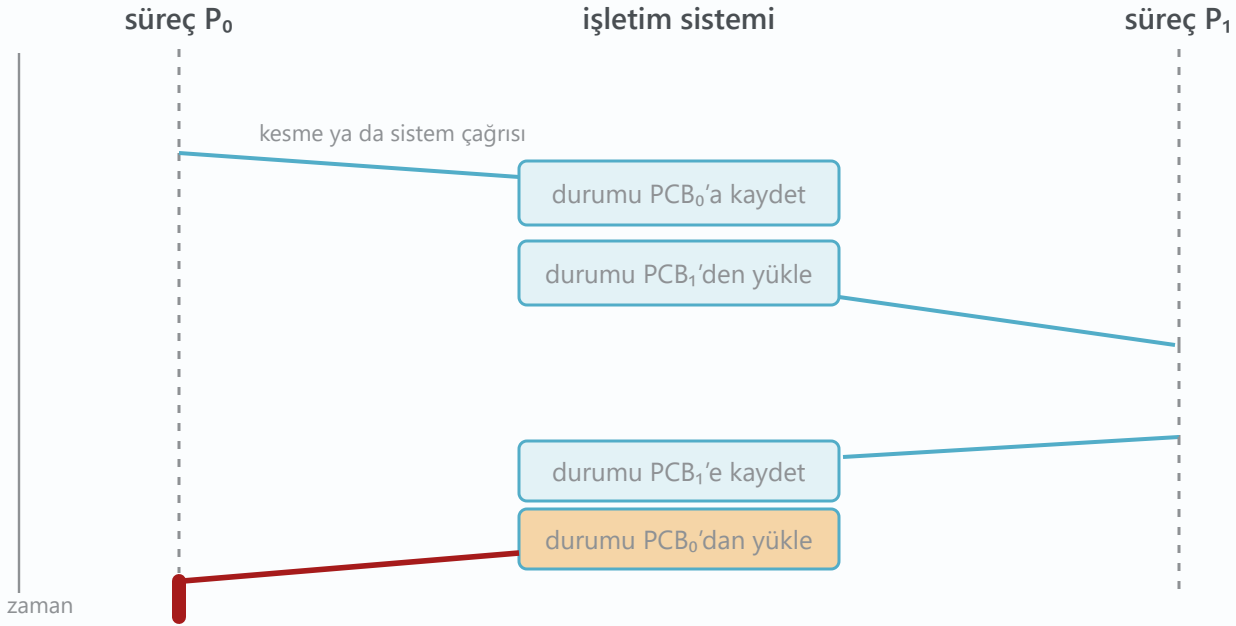
P_1 çalışıyor; P_0 boşta (hazır ya da bekliyor).

Bağlam Değiştirme (Context Switch)



Yeni kesme: P_1'in durumu PCB_1'e kaydedilir.

Bağlam Değiştirme (Context Switch)



PCB_0'dan geri yüklenir; P_0 kaldığı yerden devam eder. Aradaki gri bölümlerde hiçbir kullanıcı işi yapılmaz: **ek yük**.

Kod İzleme: Bağlam Değiştirme

```
 1 struct pcb { int pc, sp, r1; char durum; };
 2 struct pcb P[2];   /* süreç tablosu */
 3 int pc, sp, r1;    /* CPU yazmaçları */
 4 void baglam_degistir(int e, int y) {
 5   P[e].pc = pc;  P[e].sp = sp;  /* kaydet */
 6   P[e].r1 = r1;  P[e].durum = 'H';
 7   pc = P[y].pc;  sp = P[y].sp;  /* yükle */
 8   r1 = P[y].r1;  P[y].durum = 'C';
 9 }
▶ /* kesme */  baglam_degistir(0, 1);
11 /* ... P1 çalışır ... */
12 /* kesme */  baglam_degistir(1, 0);
```

	CPU (P_0)	PCB_0 çalışıyor	PCB_1 hazır
pc	104	–	340
sp	896	–	500
r1	8	–	2

P_0 çalışıyor: yazmaçlar **P_0'ın** değerlerini taşıyor. P_1'in son durumu PCB_1'de saklı. Zamanlayıcı kesmesi gelir; çekirdek P_1'e geçmeye karar verir.

Adım 1 / 11

DEĞİŞKENLER

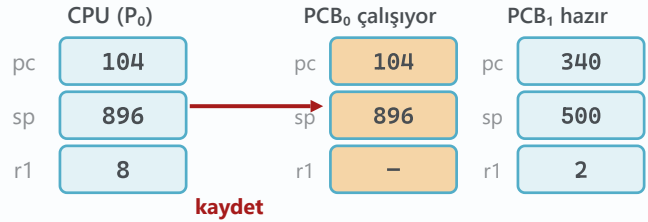
`pc = 104` `sp = 896` `r1 = 8`

ÇAĞRI YIĞINI

`kesme işleyicisi [çekirdek]`

Kod İzleme: Bağlam Değiştirme

```
1  struct pcb { int pc, sp, r1; char durum; };
2  struct pcb P[2];   /* süreç tablosu */
3  int pc, sp, r1;    /* CPU yazmaçları */
4  void baglam_degistir(int e, int y) {
▶    P[e].pc = pc;  P[e].sp = sp;  /* kaydet */
6    P[e].r1 = r1;  P[e].durum = 'H';
7    pc = P[y].pc;  sp = P[y].sp;  /* yükle */
8    r1 = P[y].r1;  P[y].durum = 'C';
9  }
10 /* kesme */  baglam_degistir(0, 1);
11 /* ... P1 çalışır ... */
12 /* kesme */  baglam_degistir(1, 0);
```



Kaydet: P_0'ın pc ve sp değerleri PCB_0'a yazılır. Bunlar kaybolursa P_0 kaldığı yere dönemez.

Adım 2 / 11

DEĞİŞKENLER

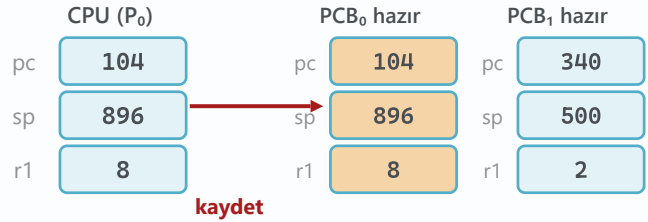
`pc = 104` `sp = 896` `r1 = 8`

ÇAĞRI YIĞINI

`baglam_degistir(0, 1)`

Kod İzleme: Bağlam Değiştirme

```
1  struct pcb { int pc, sp, r1; char durum; };
2  struct pcb P[2];   /* süreç tablosu */
3  int pc, sp, r1;    /* CPU yazmaçları */
4  void baglam_degistir(int e, int y) {
5    P[e].pc = pc;  P[e].sp = sp;  /* kaydet */
▶    P[e].r1 = r1;  P[e].durum = 'H';
7    pc = P[y].pc;  sp = P[y].sp;  /* yükle */
8    r1 = P[y].r1;  P[y].durum = 'C';
9  }
10 /* kesme */  baglam_degistir(0, 1);
11 /* ... P1 çalışır ... */
12 /* kesme */  baglam_degistir(1, 0);
```



Genel yazmaç da kaydedilir; P_0 artık **hazır** (H).

Adım 3 / 11

DEĞİŞKENLER

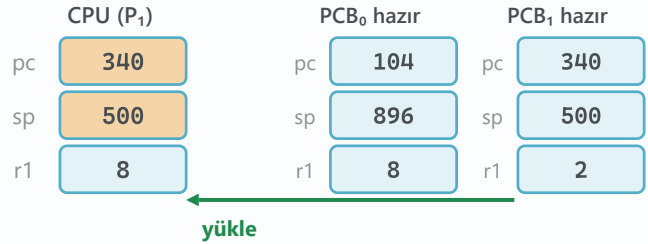
`pc = 104` `sp = 896` `r1 = 8`

ÇAĞRI YIĞINI

`baglam_degistir(0, 1)`

Kod İzleme: Bağlam Değiştirme

```
1  struct pcb { int pc, sp, r1; char durum; };
2  struct pcb P[2];   /* süreç tablosu */
3  int pc, sp, r1;    /* CPU yazmaçları */
4  void baglam_degistir(int e, int y) {
5    P[e].pc = pc;  P[e].sp = sp;  /* kaydet */
6    P[e].r1 = r1;  P[e].durum = 'H';
▶    pc = P[y].pc;  sp = P[y].sp;  /* yükle */
8    r1 = P[y].r1;  P[y].durum = 'C';
9  }
10 /* kesme */  baglam_degistir(0, 1);
11 /* ... P1 çalışır ... */
12 /* kesme */  baglam_degistir(1, 0);
```



Yükle: P_1'in pc ve sp değerleri CPU'ya. Bu andan sonra CPU P_1'in koduna ve yığınına bakar.

Adım 4 / 11

DEĞİŞKENLER

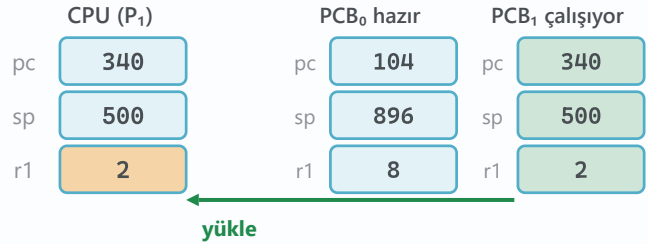
`pc = 340` `sp = 500` `r1 = 8`

ÇAĞRI YIĞINI

`baglam_degistir(0, 1)`

Kod İzleme: Bağlam Değiştirme

```
1  struct pcb { int pc, sp, r1; char durum; };
2  struct pcb P[2];   /* süreç tablosu */
3  int pc, sp, r1;    /* CPU yazmaçları */
4  void baglam_degistir(int e, int y) {
5    P[e].pc = pc;  P[e].sp = sp;  /* kaydet */
6    P[e].r1 = r1;  P[e].durum = 'H';
7    pc = P[y].pc;  sp = P[y].sp;  /* yükle */
▶    r1 = P[y].r1;  P[y].durum = 'C';
9  }
10 /* kesme */  baglam_degistir(0, 1);
11 /* ... P1 çalışır ... */
12 /* kesme */  baglam_degistir(1, 0);
```



r1 = 2 yüklenir, P_1 **çalışıyor** (C). Bu arada hiçbir kullanıcı işi yapılmadı: bağlam değiştirme saf **ek yüktür**.

Adım 5 / 11

DEĞİŞKENLER

`pc = 340` `sp = 500` `r1 = 2`

ÇAĞRI YIĞINI

`baglam_degistir(0, 1)`

Kod İzleme: Bağlam Değiştirme

```
1  struct pcb { int pc, sp, r1; char durum; };
2  struct pcb P[2];   /* süreç tablosu */
3  int pc, sp, r1;    /* CPU yazmaçları */
4  void baglam_degistir(int e, int y) {
5    P[e].pc = pc;  P[e].sp = sp;  /* kaydet */
6    P[e].r1 = r1;  P[e].durum = 'H';
7    pc = P[y].pc;  sp = P[y].sp;  /* yükle */
8    r1 = P[y].r1;  P[y].durum = 'C';
9  }
10 /* kesme */  baglam_degistir(0, 1);
▶ /* ... P1 çalışır ... */
12 /* kesme */  baglam_degistir(1, 0);
```

	CPU (P_1)	PCB_0 hazır	PCB_1 çalışıyor
pc	352	104	340
sp	492	896	500
r1	5	8	2

P_1 birkaç komut çalıştırdı: pc 340 → **352**, r1 = 5, sp = 492. PCB_1 bu sırada güncel değil; güncel değerler yalnızca CPU'da.

Adım 6 / 11

DEĞİŞKENLER

`pc = 352` `sp = 492` `r1 = 5`

ÇAĞRI YIĞINI

`P₁ kullanıcı kodu`

Kod İzleme: Bağlam Değiştirme

```
1  struct pcb { int pc, sp, r1; char durum; };
2  struct pcb P[2];   /* süreç tablosu */
3  int pc, sp, r1;    /* CPU yazmaçları */
4  void baglam_degistir(int e, int y) {
5    P[e].pc = pc;  P[e].sp = sp;  /* kaydet */
6    P[e].r1 = r1;  P[e].durum = 'H';
7    pc = P[y].pc;  sp = P[y].sp;  /* yükle */
8    r1 = P[y].r1;  P[y].durum = 'C';
9  }
10 /* kesme */  baglam_degistir(0, 1);
11 /* ... P1 çalışır ... */
▶  /* kesme */  baglam_degistir(1, 0);
```

	CPU (P_1)	PCB_0 hazır	PCB_1 çalışıyor
pc	352	104	340
sp	492	896	500
r1	5	8	2

Yeni kesme: çekirdek P_0'a dönmeye karar verir.

Adım 7 / 11

DEĞİŞKENLER

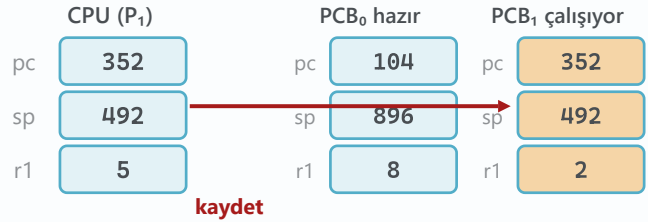
`pc = 352` `sp = 492` `r1 = 5`

ÇAĞRI YIĞINI

`kesme işleyicisi [çekirdek]`

Kod İzleme: Bağlam Değiştirme

```
1  struct pcb { int pc, sp, r1; char durum; };
2  struct pcb P[2];   /* süreç tablosu */
3  int pc, sp, r1;    /* CPU yazmaçları */
4  void baglam_degistir(int e, int y) {
▶    P[e].pc = pc;  P[e].sp = sp;  /* kaydet */
6    P[e].r1 = r1;  P[e].durum = 'H';
7    pc = P[y].pc;  sp = P[y].sp;  /* yükle */
8    r1 = P[y].r1;  P[y].durum = 'C';
9  }
10 /* kesme */  baglam_degistir(0, 1);
11 /* ... P1 çalışır ... */
12 /* kesme */  baglam_degistir(1, 0);
```



P_1'in pc ve sp değerleri PCB_1'e kaydedilir.

Adım 8 / 11

DEĞİŞKENLER

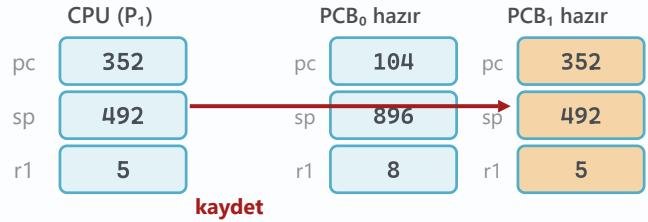
`pc = 352` `sp = 492` `r1 = 5`

ÇAĞRI YIĞINI

```
baglam_degistir(1, 0)
```

Kod İzleme: Bağlam Değiştirme

```
1  struct pcb { int pc, sp, r1; char durum; };
2  struct pcb P[2];   /* süreç tablosu */
3  int pc, sp, r1;    /* CPU yazmaçları */
4  void baglam_degistir(int e, int y) {
5    P[e].pc = pc;  P[e].sp = sp;  /* kaydet */
▶    P[e].r1 = r1;  P[e].durum = 'H';
7    pc = P[y].pc;  sp = P[y].sp;  /* yükle */
8    r1 = P[y].r1;  P[y].durum = 'C';
9  }
10 /* kesme */  baglam_degistir(0, 1);
11 /* ... P1 çalışır ... */
12 /* kesme */  baglam_degistir(1, 0);
```



r1 de kaydedilir; P_1 hazır.

Adım 9 / 11

DEĞİŞKENLER

`pc = 352` `sp = 492` `r1 = 5`

ÇAĞRI YIĞINI

```
baglam_degistir(1, 0)
```

Kod İzleme: Bağlam Değiştirme

```
1  struct pcb { int pc, sp, r1; char durum; };
2  struct pcb P[2];   /* süreç tablosu */
3  int pc, sp, r1;    /* CPU yazmaçları */
4  void baglam_degistir(int e, int y) {
5    P[e].pc = pc;  P[e].sp = sp;  /* kaydet */
6    P[e].r1 = r1;  P[e].durum = 'H';
▶    pc = P[y].pc;  sp = P[y].sp;  /* yükle */
8    r1 = P[y].r1;  P[y].durum = 'C';
9  }
10 /* kesme */  baglam_degistir(0, 1);
11 /* ... P1 çalışır ... */
12 /* kesme */  baglam_degistir(1, 0);
```

	CPU (P_0)	PCB_0 hazır	PCB_1 hazır
pc	104	104	352
sp	896	896	492
r1	5	8	5

PCB_0'daki değerler geri yüklenir: pc = 104, sp = 896.

Adım 10 / 11

DEĞİŞKENLER

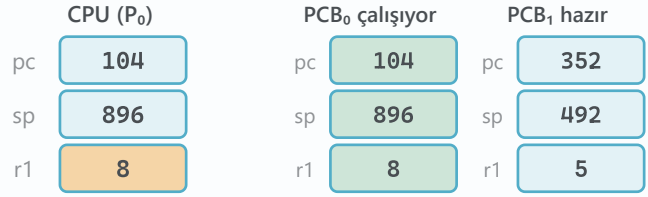
`pc = 104` `sp = 896` `r1 = 5`

ÇAĞRI YIĞINI

`baglam_degistir(1, 0)`

Kod İzleme: Bağlam Değiştirme

```
1  struct pcb { int pc, sp, r1; char durum; };
2  struct pcb P[2];   /* süreç tablosu */
3  int pc, sp, r1;    /* CPU yazmaçları */
4  void baglam_degistir(int e, int y) {
5    P[e].pc = pc;  P[e].sp = sp;  /* kaydet */
6    P[e].r1 = r1;  P[e].durum = 'H';
7    pc = P[y].pc;  sp = P[y].sp;  /* yükle */
▶    r1 = P[y].r1;  P[y].durum = 'C';
9  }
10 /* kesme */  baglam_degistir(0, 1);
11 /* ... P1 çalışır ... */
12 /* kesme */  baglam_degistir(1, 0);
```



r1 = 8: P_0 tam olarak kesildiği durumdan, **pc = 104**'ten sürer; kesildiğini fark etmez. Gerçekte yazmaç sayısı onlarcadır ve adres alanı da değişir.

Adım 11 / 11

DEĞİŞKENLER

`pc = 104` `sp = 896` `r1 = 8`

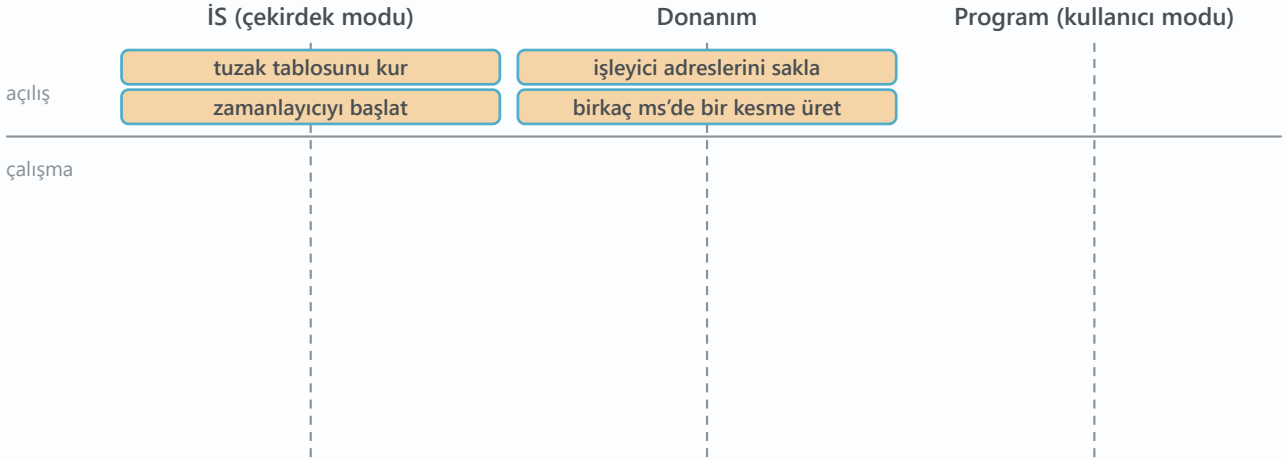
ÇAĞRI YIĞINI

`baglam_degistir(1, 0)`

ÇİZELGELEME

Sınırlı Doğrudan Yürütme

Program CPU'da doğrudan çalışır; ayrıcalıklı işler çekirdekte, CPU'yu İS **zorla** geri alır.



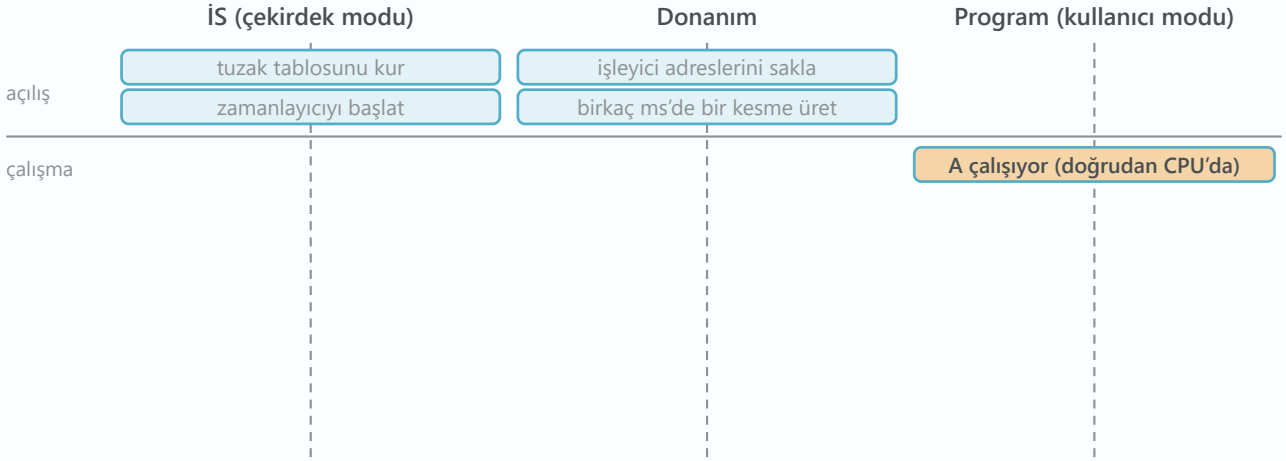
Açılışta, çekirdek modunda: İS hangi olayda hangi kodun çalışacağını (tuzak/kesme tablosu) donanıma bildirir ve zamanlayıcıyı kurar. İkisi de **ayrıcalıklı** komuttur.

Adım 1 / 5

Kesme olmasaydı (**işbirlikçi** çok görevlilik, Windows 3.x) sonsuz döngü makineyi kilitlerdi.

Sınırlı Doğrudan Yürütme

Program CPU'da doğrudan çalışır; ayrıcalıklı işler çekirdekte, CPU'yu İS **zorla** geri alır.

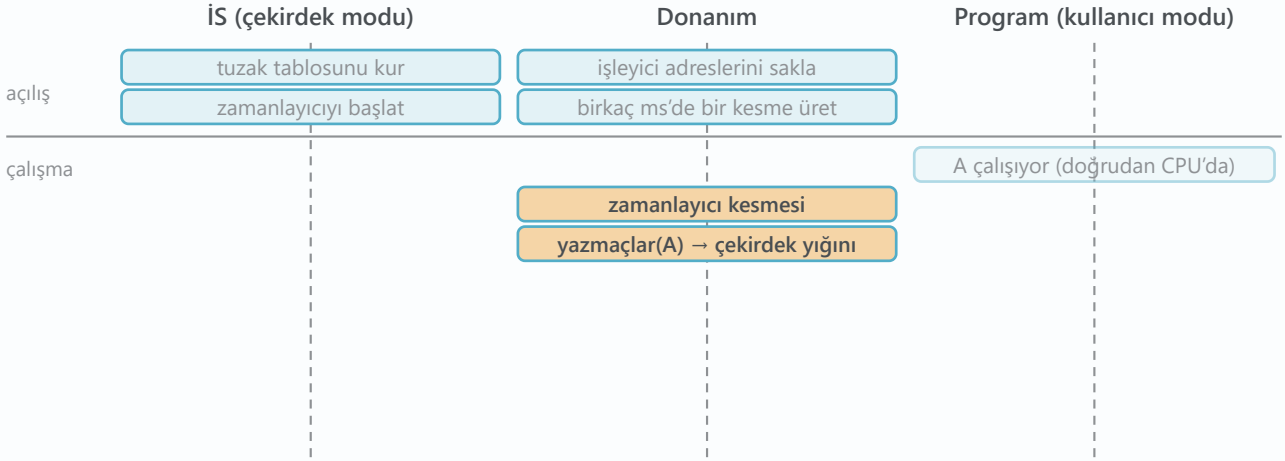


A kullanıcı modunda **doğrudan** CPU'da çalışır: arada yorumlayıcı yok, tam hız. Ayrıcalıklı bir iş için sistem çağrısıyla tuzağa düşmek zorundadır.

Kesme olmasaydı (**işbirlikçi** çok görevlilik, Windows 3.x) sonsuz döngü makineyi kilitlerdi.

Sınırlı Doğrudan Yürütme

Program CPU'da doğrudan çalışır; ayrıcalıklı işler çekirdekte, CPU'yu İS **zorla** geri alır.

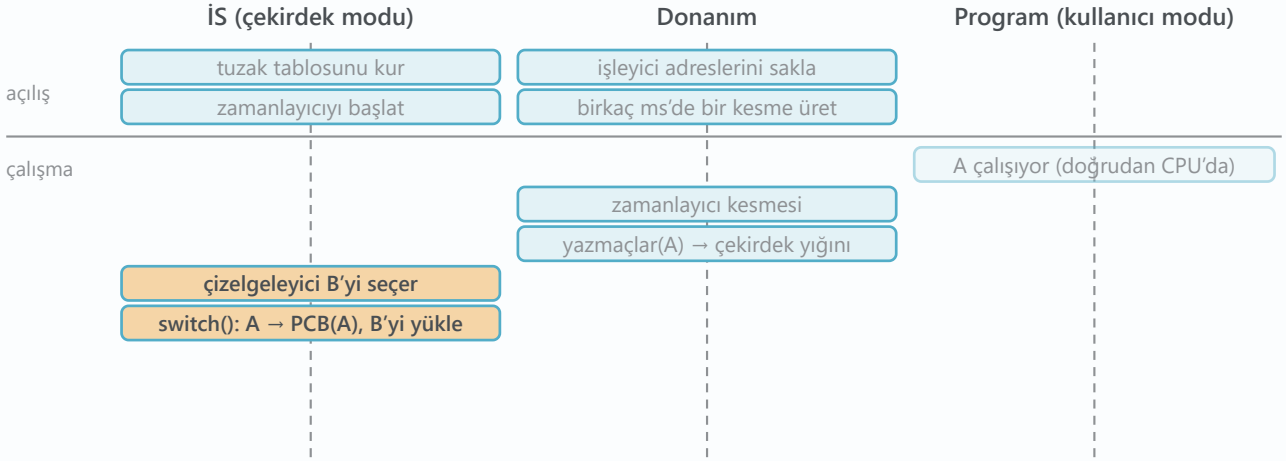


A sonsuz döngüde olsa bile kesme gelir. Donanım A'nın yazmaçlarını kaydeder, çekirdek moduna geçer ve İS'nin işleyicisine atlar: **denetim İS'de**.

Kesme olmasaydı (**işbirlikçi** çok görevlilik, Windows 3.x) sonsuz döngü makineyi kilitlerdi.

Sınırlı Doğrudan Yürütme

Program CPU'da doğrudan çalışır; ayrıcalıklı işler çekirdekte, CPU'yu İS **zorla** geri alır.

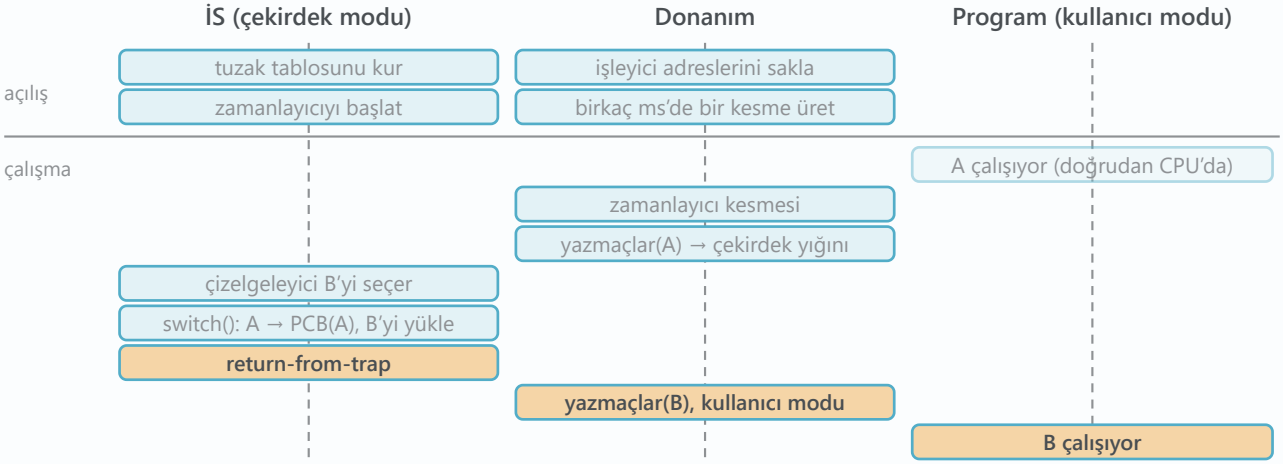


İS, A'nın dilimi dolduğu için B'ye geçmeye karar verir. `switch()` A'nın bağlamını PCB(A)'ya yazar, B'ninkini yükler ve B'nin çekirdek yığınına geçer.

Kesme olmasaydı (**işbirlikçi** çok görevlilik, Windows 3.x) sonsuz döngü makineyi kilitlerdi.

Sınırlı Doğrudan Yürütme

Program CPU'da doğrudan çalışır; ayrıcalıklı işler çekirdekte, CPU'yu İS **zorla** geri alır.



Tuzaktan dönüş, B sanki kesildiği yerden dönüyormuş gibi yazmaçlarını yükler ve kullanıcı moduna geçer: artık **B** çalışır.

Kesme olmasaydı (**işbirlikçi** çok görevlilik, Windows 3.x) sonsuz döngü makineyi kilitlerdi.

Bağlam Değiştirmenin Maliyeti

- CPU başka sürece geçerken eski sürecin durumunu **kaydetmeli**, yenisininkini **yüklemelidir**.
- Bu süre **saf ek yüktür** (overhead): bu sırada sistem yararlı iş yapmaz.
- PCB ve işletim sistemi ne kadar karmaşıksa bağlam değiştirme o kadar uzun sürer.

Doğrudan maliyet

Yazmaçların kaydı/yüklenmesi, çekirdek kodu: güncel Linux'ta tipik olarak **birkaç mikrosaniye**.

Dolaylı maliyet

Önbellek ve TLB'nin "soğuması": yeni süreç ilk anlarda daha çok önbellek ıskası yaşar; çoğu zaman doğrudan maliyetten büyüktür.

Donanım desteği

Birden çok yazmaç kümesi, TLB'de adres alanı etiketi (ASID / PCID) ile boşaltmadan kaçınma.

ÇİZELGELEME

Kesmenin Ele Alınması

Bir G/Ç kesmesi geldiğinde işletim sisteminin en alt katmanında olanlar (Tanenbaum):

1. Donanım program sayacı, PSW vb. değerleri yığına kaydeder.
   Kesme isteği komut sınırında kabul edilir; CPU çekirdek moduna geçer.

ÇİZELGELEME

Kesmenin Ele Alınması

Bir G/Ç kesmesi geldiğinde işletim sisteminin en alt katmanında olanlar (Tanenbaum):

1. Donanım program sayacı, PSW vb. değerleri yığına kaydeder.

2. Donanım kesme vektöründen yeni program sayacını yükler.
   Vektör, her kesme türü için işleyicinin adresini tutan tablodur.

ÇİZELGELEME

Kesmenin Ele Alınması

Bir G/Ç kesmesi geldiğinde işletim sisteminin en alt katmanında olanlar (Tanenbaum):

1. Donanım program sayacı, PSW vb. değerleri yığına kaydeder.

2. Donanım kesme vektöründen yeni program sayacını yükler.

3. Assembly yordamı yazmaçları kaydeder.
   Yazmaçlar o anki sürecin PCB'sine (ya da çekirdek yığınına) yazılır.

ÇİZELGELEME

Kesmenin Ele Alınması

Bir G/Ç kesmesi geldiğinde işletim sisteminin en alt katmanında olanlar (Tanenbaum):

1. Donanım program sayacı, PSW vb. değerleri yığına kaydeder.
2. Donanım kesme vektöründen yeni program sayacını yükler.
3. Assembly yordamı yazmaçları kaydeder.
4. Assembly yordamı yeni bir yığın hazırlar.
   Kesme işleyicisi çekirdek yığınında çalışır.

ÇİZELGELEME

Kesmenin Ele Alınması

Bir G/Ç kesmesi geldiğinde işletim sisteminin en alt katmanında olanlar (Tanenbaum):

1. Donanım program sayacı, PSW vb. değerleri yığına kaydeder.
2. Donanım kesme vektöründen yeni program sayacını yükler.
3. Assembly yordamı yazmaçları kaydeder.
4. Assembly yordamı yeni bir yığın hazırlar.
5. C kesme hizmet yordamı çalışır.

   Genelde girdiyi okur/tamponlar; bekleyen süreci hazır yapar.

Kesmenin Ele Alınması

Bir G/Ç kesmesi geldiğinde işletim sisteminin en alt katmanında olanlar (Tanenbaum):

1. Donanım program sayacı, PSW vb. değerleri yığına kaydeder.
2. Donanım kesme vektöründen yeni program sayacını yükler.
3. Assembly yordamı yazmaçları kaydeder.
4. Assembly yordamı yeni bir yığın hazırlar.
5. C kesme hizmet yordamı çalışır.
6. Çizelgeleyici sıradaki süreci seçer.
   Uyanan süreç daha öncelikliyse kesilen süreç yerine o seçilebilir.

Kesmenin Ele Alınması

Bir G/Ç kesmesi geldiğinde işletim sisteminin en alt katmanında olanlar (Tanenbaum):

1. Donanım program sayacı, PSW vb. değerleri yığına kaydeder.
2. Donanım kesme vektöründen yeni program sayacını yükler.
3. Assembly yordamı yazmaçları kaydeder.
4. Assembly yordamı yeni bir yığın hazırlar.
5. C kesme hizmet yordamı çalışır.
6. Çizelgeleyici sıradaki süreci seçer.
7. C yordamı assembly koduna döner.

   Kalan iş, seçilen sürecin bağlamının yüklenmesidir.

ÇİZELGELEME

Kesmenin Ele Alınması

Bir G/Ç kesmesi geldiğinde işletim sisteminin en alt katmanında olanlar (Tanenbaum):

1. Donanım program sayacı, PSW vb. değerleri yığına kaydeder.
2. Donanım kesme vektöründen yeni program sayacını yükler.
3. Assembly yordamı yazmaçları kaydeder.
4. Assembly yordamı yeni bir yığın hazırlar.
5. C kesme hizmet yordamı çalışır.
6. Çizelgeleyici sıradaki süreci seçer.
7. C yordamı assembly koduna döner.
8. Assembly yordamı yeni süreci başlatır.

   Yazmaçlar ve bellek eşlemesi yüklenir, kullanıcı moduna dönülür.

Süreçler Arası İletişim

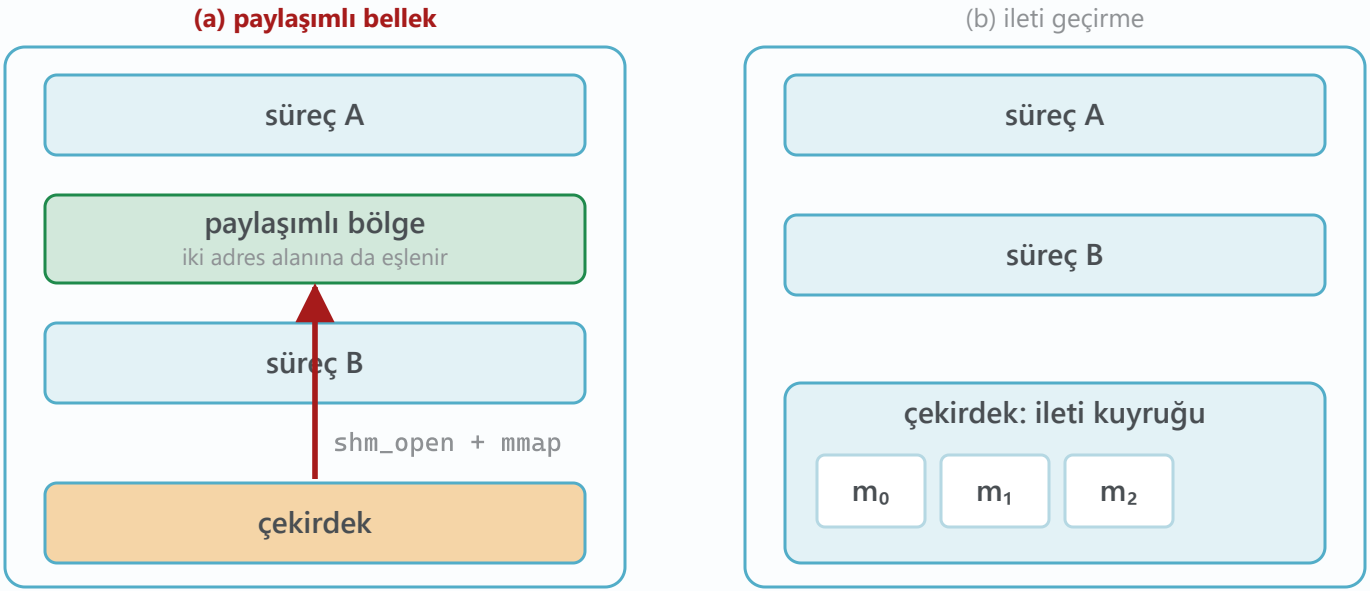
- Süreçler **bağımsız** (başkasını etkilemez) ya da **işbirlikçi** (veri paylaşır) olabilir.
- İşbirliği nedenleri: bilgi paylaşımı, hızlanma (işi parçalamak), modülerlik, güvenlik ve yalıtım (Chrome).
- İşbirlikçi süreçler bir **IPC** (interprocess communication) mekanizmasına ihtiyaç duyar.

Üç temel soru

1. Bir süreç diğerine bilgiyi **nasıl** iletir?
2. Süreçler birbirinin yoluna çıkmasın: **aynı koltuğa iki rezervasyon** nasıl önlenir?
3. Bağımlılıklarda **doğru sıra** nasıl sağlanır? (Nişan almadan ateş edilmemeli; üretilmeden tüketilmemeli.)

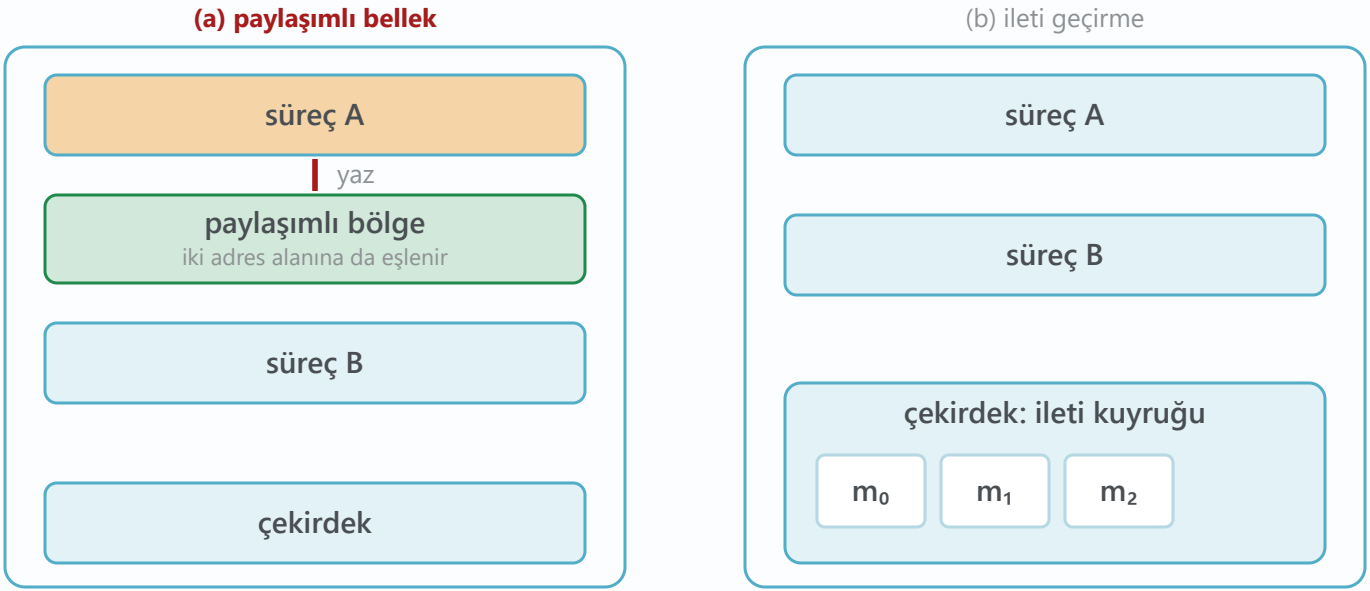
İki temel model: **paylaşımlı bellek** ve **ileti geçirme**. Son iki soru **eşgüdüm** (senkronizasyon) gerektirir: paylaşılan veriye denetimsiz erişim, sonucu yürütme sırasına bağlı kılar (yarış durumu). Aynı sorular iş parçacıkları için de geçerlidir.

Paylaşımlı Bellek ve İleti Geçirme



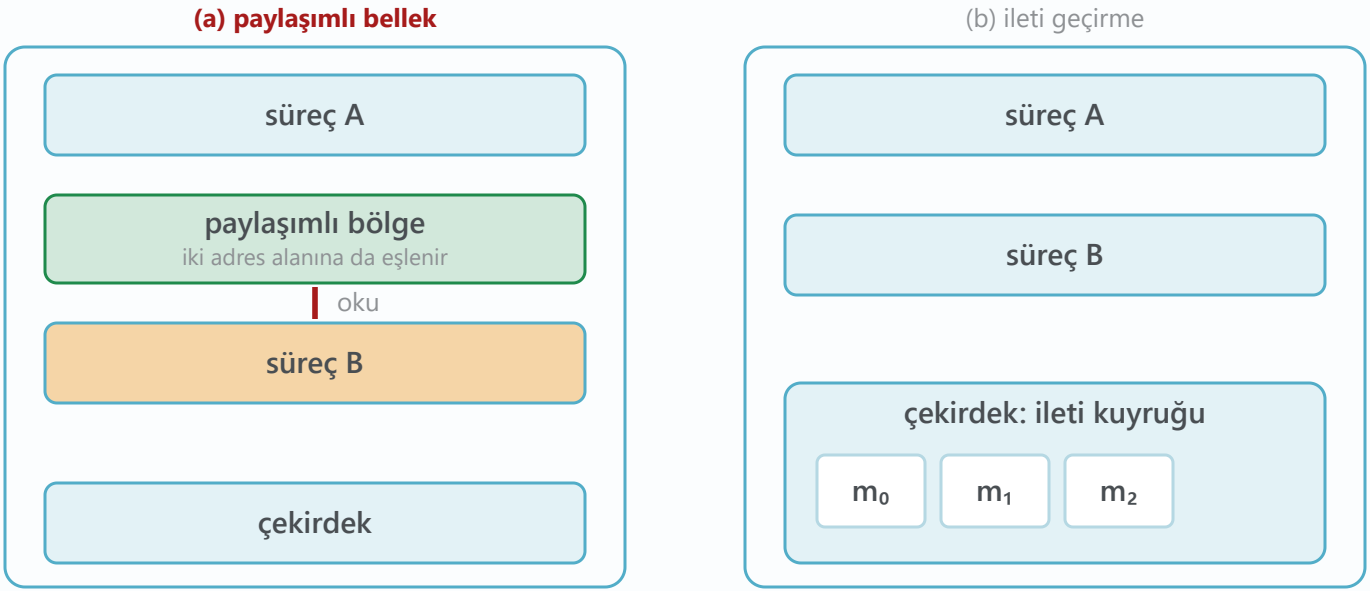
Paylaşımlı bellek: çekirdek bir bölge oluşturur ve iki sürecin adres alanına eşler (yalnızca kurulumda sistem çağrısı).

Paylaşımlı Bellek ve İleti Geçirme



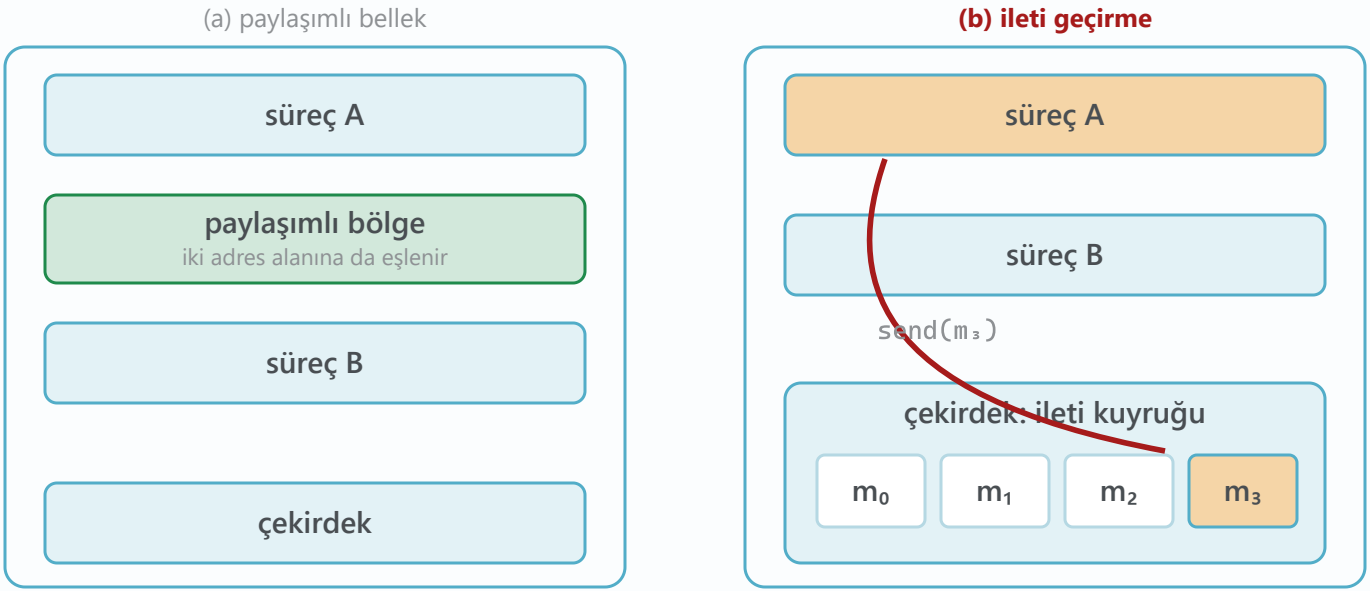
A bölgeye doğrudan yazar: çekirdek devrede **değildir**, bu yüzden çok hızlıdır.

Paylaşımlı Bellek ve İleti Geçirme



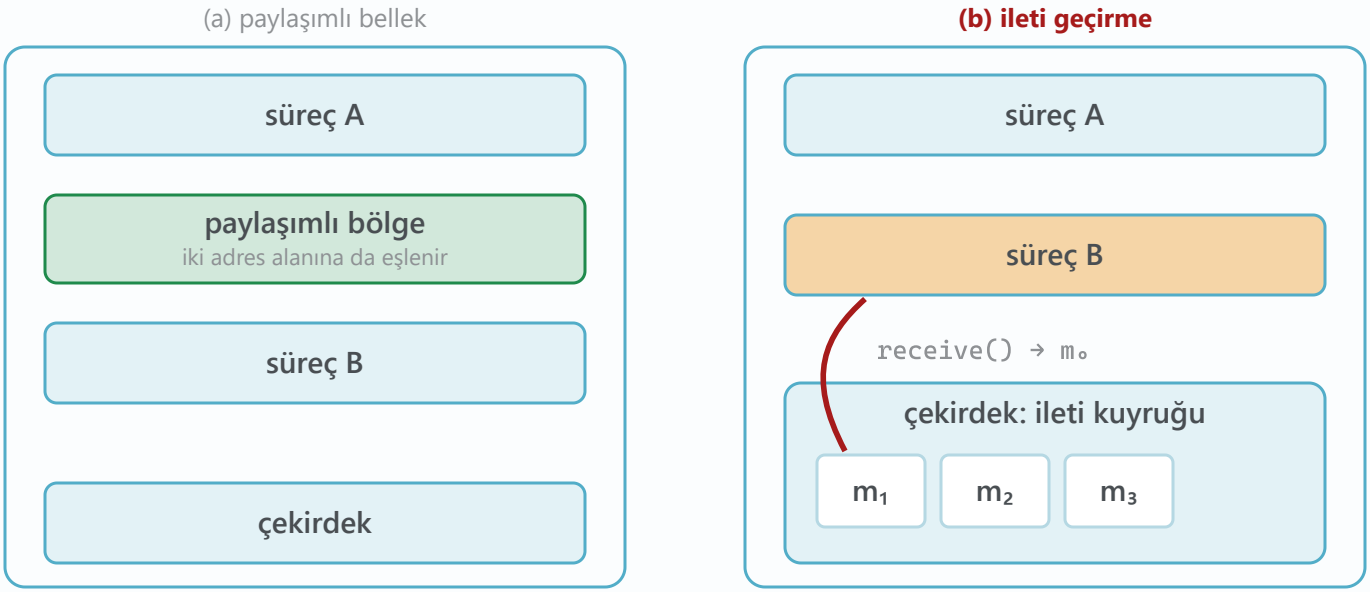
B aynı veriyi okur. Ancak senkronizasyonu (kim ne zaman yazar?) süreçler kendileri sağlamalıdır.

Paylaşımlı Bellek ve İleti Geçirme



İleti geçirme: A `send(m₃)` der; çekirdek iletiyi kendi kuyruğuna kopyalar.

Paylaşımlı Bellek ve İleti Geçirme



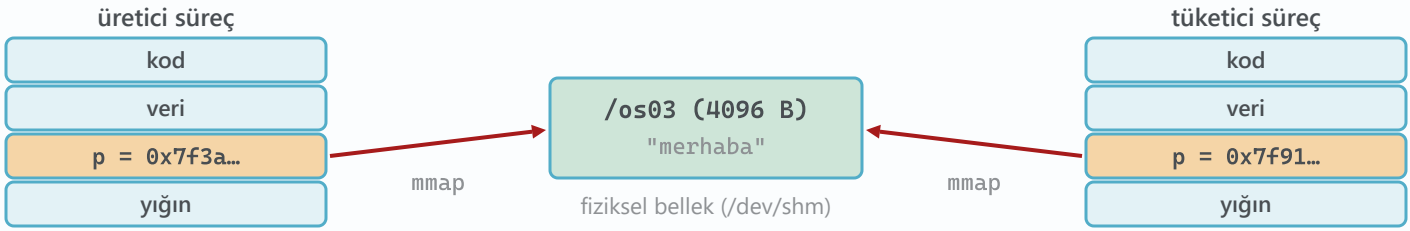
B `receive()` ile sıradaki iletiyi alır. Her iletide sistem çağrısı ve kopyalama vardır; ama senkronizasyon çekirdekçe sağlanır, ağ üzerinden de çalışır.

POSIX Paylaşımlı Bellek

```c
/* üretici */
int fd = shm_open("/os03", O_CREAT|O_RDWR, 0600);
ftruncate(fd, 4096);
char *p = mmap(NULL, 4096, PROT_READ|PROT_WRITE,
               MAP_SHARED, fd, 0);
strcpy(p, "merhaba");   /* doğrudan yaz */

/* tüketici (başka süreç) */
int fd = shm_open("/os03", O_RDONLY, 0);
char *p = mmap(NULL, 4096, PROT_READ,
               MAP_SHARED, fd, 0);
printf("%s\n", p);
shm_unlink("/os03");    /* sil */
```

- `shm_open` adlı bir nesne oluşturur (Linux'ta `/dev/shm` altında).
- `ftruncate` boyutu belirler, `mmap` adres alanına eşler.
- Sonrası sıradan bellek erişimidir.
- Tüketici üreticiden önce okursa boş görür: gerçek kodda semafor gibi bir **senkronizasyon** gerekir.
- Eski System V API: `shmget` / `shmat`.



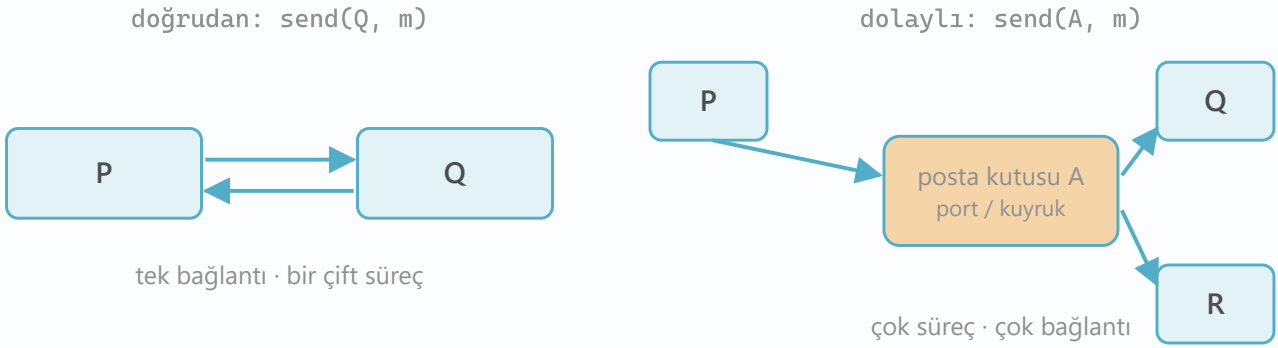
İleti Geçirme Tasarımı

İki temel işlem: `send(mesaj)` ve `receive(mesaj)`. Önce bir **iletişim bağı** kurulmalıdır.

Boyut	Seçenekler
Fiziksel bağ	paylaşımlı bellek, donanım veri yolu, ağ
Adlandırma	**doğrudan** (sürecin adı) · **dolaylı** (posta kutusu / port)
Eş zamanlı / eş zamansız	**bloklayan (senkron)**: gönderen alınana kadar, alıcı ileti gelene kadar bekler · **bloklamayan (eş zamansız)**: hemen döner
Tamponlama	**sıfır kapasite** (gönderen bekler: buluşma/rendezvous) · **sınırlı** (kuyruk doluysa bekler) · **sınırsız** (hiç beklemez)

Bloklayan send + bloklayan receive = **buluşma (rendezvous)**: üretici–tüketici problemi kendiliğinden çözülür. Örnekler: POSIX `mq_send` / `mq_receive`, Mach portları, Windows ALPC.

Doğrudan ve Dolaylı İletişim



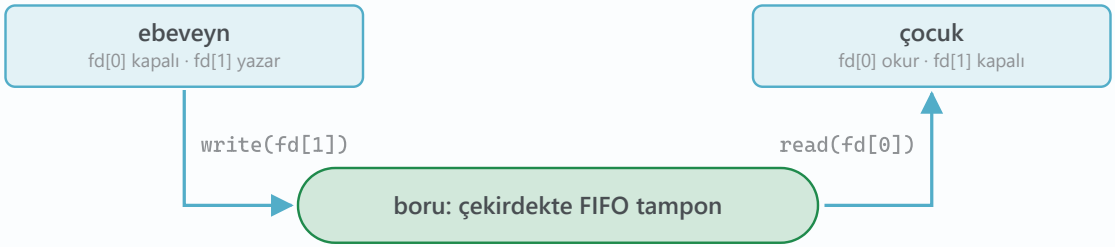
Doğrudan

- `send(P, m)` : P'ye gönder; `receive(Q, m)` : Q'dan al.
- Bağlantı otomatik kurulur, tam olarak bir çift sürece aittir.
- Her çift arasında tek bağlantı; genelde çift yönlü.

Dolaylı (posta kutusu)

- `send(A, m)` / `receive(A, m)` : A kutusu üzerinden.
- Her posta kutusunun tekil kimliği vardır; aynı kutuyu paylaşanlar haberleşir.
- Bir bağlantı çok sürece, bir çift çok bağlantıya sahip olabilir; tek ya da çift yönlü.

Borular (Pipes)



yazma ucu → okuma ucu (tek yönlü)

Tasarım soruları

- Tek yönlü mü (simplex), çift yönlü mü?
- Çift yönlüyse yarı mı (half), tam mı (full duplex)?
- Süreçler arasında ebeveyn-çocuk ilişkisi gerekli mi?
- Ağ üzerinden kullanılabilir mi?

	Sıradan (adsız)	Adlandırılmış (FIFO)
Erişim	yalnız oluşturan ve çocukları	dosya adı bilen her süreç
Yön	tek yönlü	UNIX: yarı çift yönlü; Windows: tam çift yönlü
Ömür	süreçler kapatınca biter	`mkfifo`; dosya sisteminde kalır
Ağ	hayır	UNIX hayır; Windows evet

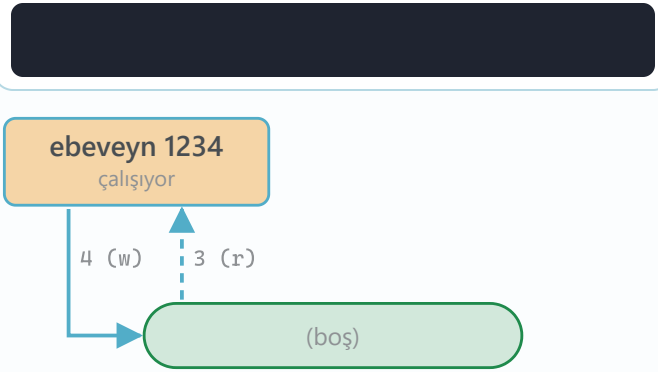
Kod İzleme: pipe() + fork()

```
1  int main(void) {
2    int fd[2];
3    char buf[32];
▶4   pipe(fd);
5    pid_t pid = fork();
6    if (pid == 0) {   /* çocuk: okur */
7      close(fd[1]);
8      ssize_t n = read(fd[0], buf, sizeof buf -
   1);
9      buf[n] = '\0';
10     printf("çocuk aldı: %s\n", buf);
11     close(fd[0]);
12     return 0;
13   }
14   close(fd[0]);     /* ebeveyn: yazar */
15   write(fd[1], "merhaba", 7);
16   close(fd[1]);
17   wait(NULL);
18   return 0;
19 }
```

DEĞİŞKENLER

`fd[0] = 3` `fd[1] = 4`

ÇIKTI



`pipe(fd)`: çekirdek boş bir tampon açar; okuma ucu `fd[0]=3`, yazma ucu `fd[1]=4` (0–2 stdin/stdout/stderr).

Kod İzleme: pipe() + fork()

```
1  int main(void) {
2    int fd[2];
3    char buf[32];
4    pipe(fd);
▶    pid_t pid = fork();
6    if (pid == 0) {    /* çocuk: okur */
7      close(fd[1]);
8      ssize_t n = read(fd[0], buf, sizeof buf -
   1);
9      buf[n] = '\0';
10     printf("çocuk aldı: %s\n", buf);
11     close(fd[0]);
12     return 0;
13   }
14   close(fd[0]);      /* ebeveyn: yazar */
15   write(fd[1], "merhaba", 7);
16   close(fd[1]);
17   wait(NULL);
18   return 0;
19 }
```

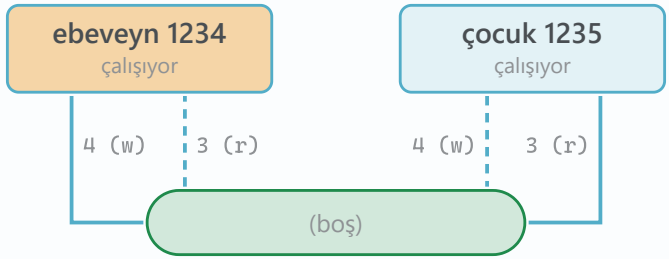
DEĞİŞKENLER

`fd[0] = 3` `fd[1] = 4`

`ebeveyn.pid = 1235` `çocuk.pid = 0`

ÇIKTI





`fork()` : çocuk tanımlayıcıları miras alır; şimdi her iki uç da iki süreçte açık.

Kod İzleme: pipe() + fork()

```
1  int main(void) {
2    int fd[2];
3    char buf[32];
4    pipe(fd);
5    pid_t pid = fork();
6    if (pid == 0) {   /* çocuk: okur */
▶      close(fd[1]);
8      ssize_t n = read(fd[0], buf, sizeof buf -
   1);
9      buf[n] = '\0';
10     printf("çocuk aldı: %s\n", buf);
11     close(fd[0]);
12     return 0;
13   }
14   close(fd[0]);     /* ebeveyn: yazar */
15   write(fd[1], "merhaba", 7);
16   close(fd[1]);
17   wait(NULL);
18   return 0;
19 }
```

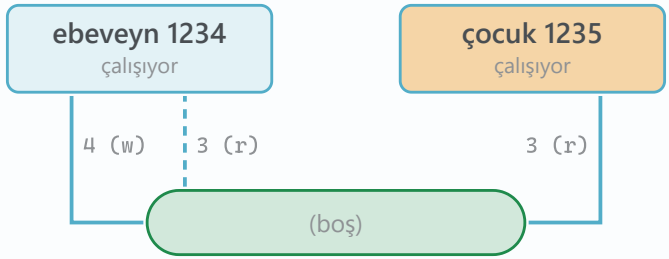
DEĞİŞKENLER

`fd[0] = 3` `fd[1] = 4`

`ebeveyn.pid = 1235` `çocuk.pid = 0`

ÇIKTI





Çocuk kullanmayacağı yazma ucunu kapatır.

Kod İzleme: pipe() + fork()

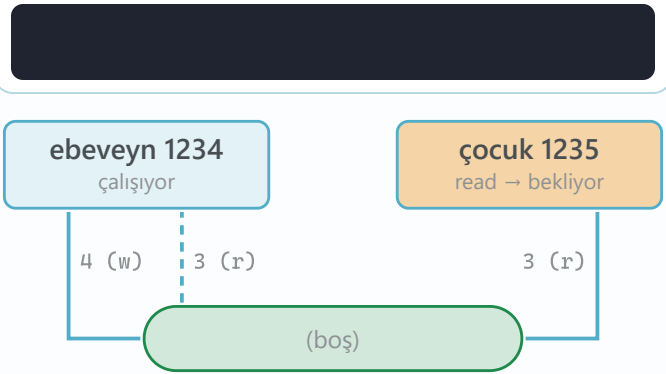
```
1  int main(void) {
2    int fd[2];
3    char buf[32];
4    pipe(fd);
5    pid_t pid = fork();
6    if (pid == 0) {   /* çocuk: okur */
7      close(fd[1]);
▶      ssize_t n = read(fd[0], buf, sizeof buf -
   1);
9      buf[n] = '\0';
10     printf("çocuk aldı: %s\n", buf);
11     close(fd[0]);
12     return 0;
13   }
14   close(fd[0]);     /* ebeveyn: yazar */
15   write(fd[1], "merhaba", 7);
16   close(fd[1]);
17   wait(NULL);
18   return 0;
19 }
```

DEĞİŞKENLER

`fd[0] = 3` `fd[1] = 4`

`ebeveyn.pid = 1235` `çocuk.pid = 0`

ÇIKTI



Boru boş: `read` çocuğu **bloke** eder.

Kod İzleme: pipe() + fork()

```
1  int main(void) {
2    int fd[2];
3    char buf[32];
4    pipe(fd);
5    pid_t pid = fork();
6    if (pid == 0) {   /* çocuk: okur */
7      close(fd[1]);
8      ssize_t n = read(fd[0], buf, sizeof buf -
   1);
9      buf[n] = '\0';
10     printf("çocuk aldı: %s\n", buf);
11     close(fd[0]);
12     return 0;
13   }
▶  close(fd[0]);     /* ebeveyn: yazar */
15   write(fd[1], "merhaba", 7);
16   close(fd[1]);
17   wait(NULL);
18   return 0;
19 }
```

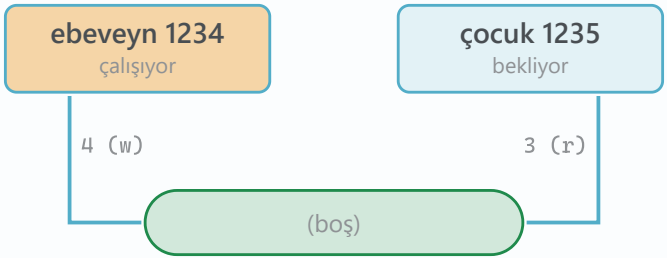
DEĞİŞKENLER

`fd[0] = 3` `fd[1] = 4`

`ebeveyn.pid = 1235` `çocuk.pid = 0`

ÇIKTI





Ebeveyn okuma ucunu kapatır.

Kod İzleme: pipe() + fork()

```
1  int main(void) {
2    int fd[2];
3    char buf[32];
4    pipe(fd);
5    pid_t pid = fork();
6    if (pid == 0) {   /* çocuk: okur */
7      close(fd[1]);
8      ssize_t n = read(fd[0], buf, sizeof buf -
   1);
9      buf[n] = '\0';
10     printf("çocuk aldı: %s\n", buf);
11     close(fd[0]);
12     return 0;
13   }
14   close(fd[0]);     /* ebeveyn: yazar */
▶    write(fd[1], "merhaba", 7);
16   close(fd[1]);
17   wait(NULL);
18   return 0;
19 }
```

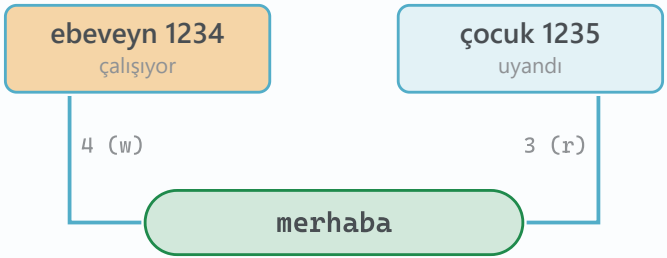
DEĞİŞKENLER

`fd[0] = 3` `fd[1] = 4`

`ebeveyn.pid = 1235` `çocuk.pid = 0`

ÇIKTI





Ebeveyn 7 bayt yazar; ≤ PIPE_BUF yazımlar bölünmez (atomik). Bekleyen okuyucu uyanır.

Kod İzleme: pipe() + fork()

```
1  int main(void) {
2    int fd[2];
3    char buf[32];
4    pipe(fd);
5    pid_t pid = fork();
6    if (pid == 0) {   /* çocuk: okur */
7      close(fd[1]);
8      ssize_t n = read(fd[0], buf, sizeof buf -
   1);
9      buf[n] = '\0';
10     printf("çocuk aldı: %s\n", buf);
11     close(fd[0]);
12     return 0;
13   }
14   close(fd[0]);     /* ebeveyn: yazar */
15   write(fd[1], "merhaba", 7);
▶    close(fd[1]);
▶    wait(NULL);
18   return 0;
19 }
```

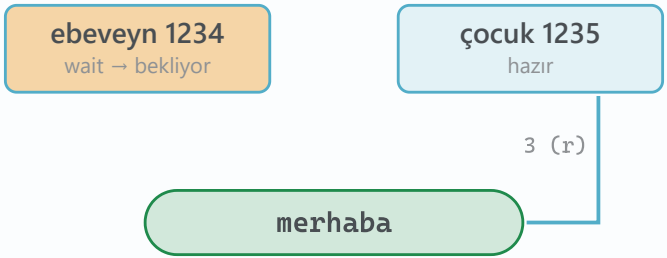
DEĞİŞKENLER

`fd[0] = 3` `fd[1] = 4`

`ebeveyn.pid = 1235` `çocuk.pid = 0`

ÇIKTI





Ebeveyn yazma ucunu kapatır (okuyucu ileride EOF görebilsin) ve `wait` ile bekler.

Kod İzleme: pipe() + fork()

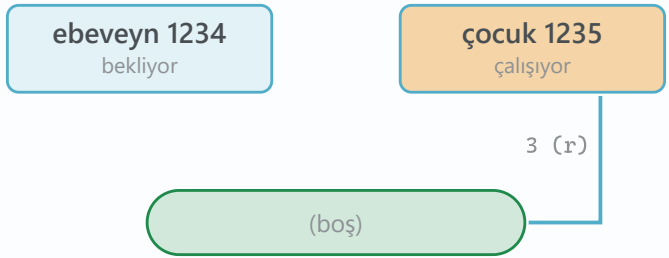
```
1  int main(void) {
2    int fd[2];
3    char buf[32];
4    pipe(fd);
5    pid_t pid = fork();
6    if (pid == 0) {   /* çocuk: okur */
7      close(fd[1]);
▶      ssize_t n = read(fd[0], buf, sizeof buf - 1);
9      buf[n] = '\0';
10     printf("çocuk aldı: %s\n", buf);
11     close(fd[0]);
12     return 0;
13   }
14   close(fd[0]);     /* ebeveyn: yazar */
15   write(fd[1], "merhaba", 7);
16   close(fd[1]);
17   wait(NULL);
18   return 0;
19 }
```

DEĞİŞKENLER

`n = 7` `buf = "merhaba"` `çocuk.pid = 0`

ÇIKTI





`read` 7 döndürür; veri tampondan çocuğun `buf` dizisine kopyalanır.

Kod İzleme: pipe() + fork()

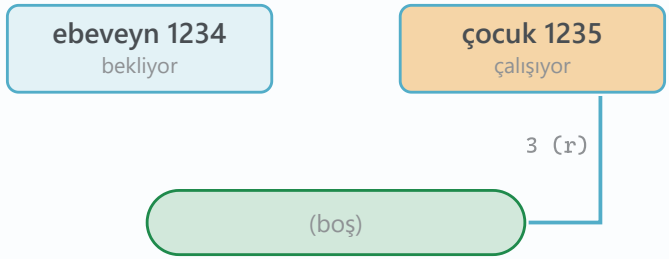
```
1  int main(void) {
2    int fd[2];
3    char buf[32];
4    pipe(fd);
5    pid_t pid = fork();
6    if (pid == 0) {   /* çocuk: okur */
7      close(fd[1]);
8      ssize_t n = read(fd[0], buf, sizeof buf -
   1);
▶      buf[n] = '\0';
▶      printf("çocuk aldı: %s\n", buf);
11     close(fd[0]);
12     return 0;
13   }
14   close(fd[0]);     /* ebeveyn: yazar */
15   write(fd[1], "merhaba", 7);
16   close(fd[1]);
17   wait(NULL);
18   return 0;
19 }
```

DEĞİŞKENLER

`n = 7` `buf = "merhaba\0"` `çocuk.pid = 0`

ÇIKTI

```
çocuk aldı: merhaba
```



Sonlandırıcı `'\0'` eklenir ve yazdırılır.

Kod İzleme: pipe() + fork()

```
1  int main(void) {
2    int fd[2];
3    char buf[32];
4    pipe(fd);
5    pid_t pid = fork();
6    if (pid == 0) {   /* çocuk: okur */
7      close(fd[1]);
8      ssize_t n = read(fd[0], buf, sizeof buf -
   1);
9      buf[n] = '\0';
10     printf("çocuk aldı: %s\n", buf);
11     close(fd[0]);
12     return 0;
13   }
14   close(fd[0]);     /* ebeveyn: yazar */
15   write(fd[1], "merhaba", 7);
16   close(fd[1]);
▶    wait(NULL);
▶    return 0;
19 }
```

DEĞİŞKENLER

`ebeveyn.pid = 1235`

ÇIKTI

```
çocuk aldı: merhaba
```

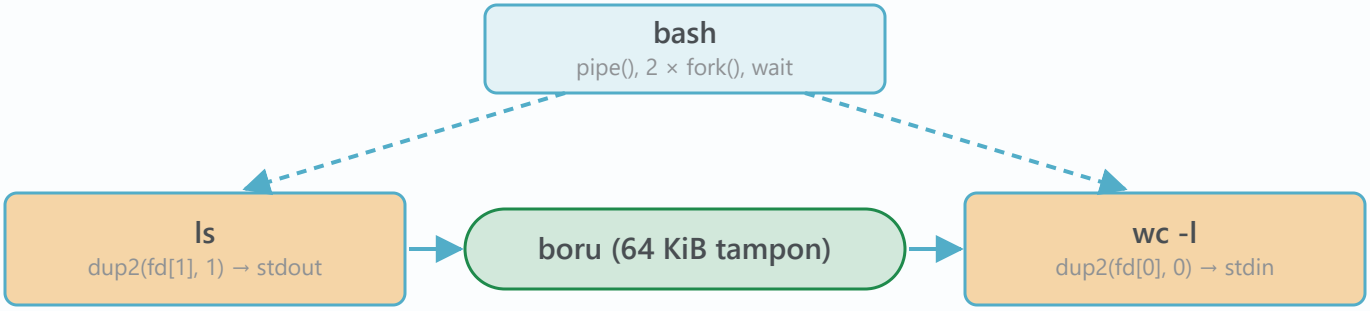
ebeveyn 1234
çalışıyor

çocuk 1235
bitti

(boş)

Çocuk okuma ucunu kapatıp çıkar; ebeveyn `wait`'ten döner ve biter. Yalnız çocuk yazdığı için çıktı belirlenimlidir.

Kabukta Boru Hattı: ls | wc -l

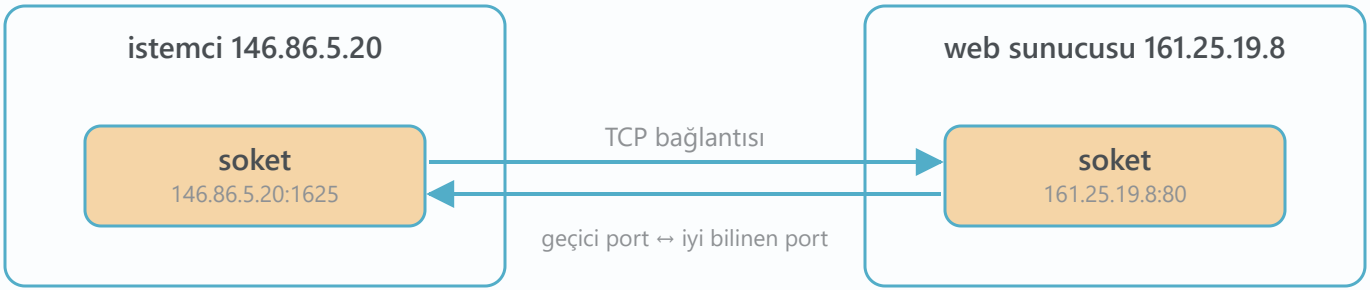


her iki süreç de kullanmadığı uçları kapatmalı; aksi hâlde wc EOF göremez

```
int fd[2]; pipe(fd);
if (fork() == 0) { dup2(fd[1], 1); close(fd[0]); close(fd[1]);
                   execlp("ls", "ls", NULL); }
if (fork() == 0) { dup2(fd[0], 0); close(fd[0]); close(fd[1]);
                   execlp("wc", "wc", "-l", NULL); }
close(fd[0]); close(fd[1]); wait(NULL); wait(NULL);
```

Hata denetimleri kısalık için çıkarıldı. `dup2(eski, yeni)` yeni tanımlayıcıyı eskinin kopyası yapar; programlar stdin/stdout ile çalıştığı için boruyu fark etmez.

Soketler



- **Soket:** iletişimin uç noktası; IP adresi + port ile tanımlanır (`161.25.19.8:80`).
- İletişim **bir çift soket** arasında olur; port, makinedeki hizmeti ayırt eder.
- 1024'ün altındaki portlar **iyi bilinen** hizmetlere ayrılmıştır (SSH 22, HTTP 80, HTTPS 443); sunucu olarak bu portları açmak (bind) yetki gerektirir.
- `127.0.0.1` (IPv6: `::1`) geri döngü adresidir: aynı makinedeki süreçler.
- **TCP** bağlantılı, sıralı, güvenilir bayt akışı; **UDP** bağlantısız datagram.
- Aynı makinede **UNIX alan soketleri** (`AF_UNIX`) daha hızlıdır; Docker, systemd, X/Wayland bunları kullanır.

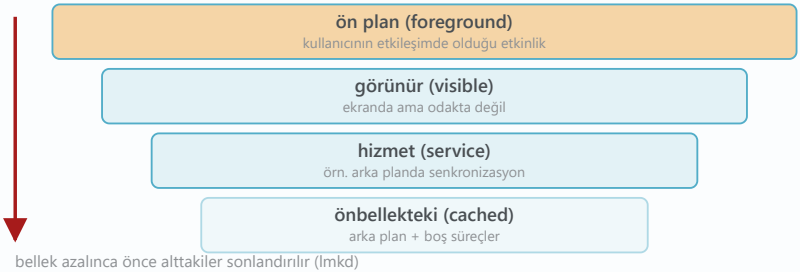
Mobil Sistemlerde Çoklu Görev

iOS / iPadOS

- İlk sürümler (2007) yalnız tek kullanıcı uygulamasına izin veriyordu; iOS 4 (2010) sınırlı çoklu görev getirdi.
- Arka planda yalnız izin verilen işler: ses çalma, konum, VoIP, `BGTaskScheduler` görevleri; diğerleri askıya alınır.

Android

- Arka plan süreçlerine izin verir; uzun işler **hizmet (service)** bileşeniyle, arayüz askıdayken bile sürer.
- Android 8'den beri arka plan kısıtları: uzun iş için bildirimli **ön plan hizmeti** ya da `WorkManager`.



Kısıtların nedeni sınırlı bellek ve pil. Eski belgelerdeki "arka plan" ve "boş" süreçler bugün **önbellekteki (cached)** grupta toplanır; bellek azalınca `lmkd` alttan başlayarak sonlandırır.

ÖZET

Özet

Süreç
Yürütülen program: metin, veri, heap, yığın + CPU bağlamı + kaynaklar. Çekirdek onu PCB ile izler.

Yaşam döngüsü
Yeni → Hazır ⇄ Çalışıyor → Bekliyor → Hazır … → Sonlandı. Çıkış durumu `wait` ile alınır.

UNIX API
`fork` kopyalar (COW), `exec` görüntüyü değiştirir, `wait` bekler ve biçer, `exit` bitirir.

Çizelgeleme altyapısı
Hazır/bekleme kuyrukları; bağlam değiştirme saf ek yüktür.

IPC
Paylaşımlı bellek hızlı ama senkronizasyon ister; ileti geçirme, boru ve soketler çekirdek aracılığıyla çalışır.

Mobil çoklu görev
Sınırlı bellek ve pil yüzünden arka plandaki süreçler askıya alınır, gerekirse sonlandırılır.

ÖZET

Kontrol Soruları

1. `fork(); fork();` çalıştıran program toplam kaç süreçle biter? Ya `for (i = 0; i < 3; i++) fork();` ?
2. `printf("A"); fork(); printf("B\n");` stdout bir *dosyaya* yönlendirildiğinde çıktı nedir? Neden?
3. Bir süreç Bekliyor durumundan doğrudan Çalışıyor durumuna geçebilir mi?
4. Zombi ile yetim süreç arasındaki fark nedir? Hangisi sistem için daha çok sorun çıkarır?
5. `exec` başarılı olursa ardından gelen satır ne zaman çalışır?
6. p = 0,5 ve n = 3 için CPU kullanımı nedir?
7. Boru örneğinde ebeveyn `close(fd[1])` yapmasaydı ve çocuk EOF'a kadar okusaydı ne olurdu?

▶ **Cevapları göster**

Kaynaklar

- A. Silberschatz, P. B. Galvin, G. Gagne, *Operating System Concepts*, 10. baskı, Wiley, 2018 — Bölüm 3: Processes.
- A. S. Tanenbaum, H. Bos, *Modern Operating Systems*, 4. baskı (5. baskı 2023), Pearson — Bölüm 2: Processes and Threads.
- R. H. Arpaci-Dusseau, A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces* (OSTEP), pages.cs.wisc.edu/~remzi/OSTEP — Processes, Process API.
- Linux kılavuz sayfaları: `fork(2)`, `execve(2)`, `wait(2)`, `pipe(7)`, `signal(7)`, `shm_overview(7)`.
- Chromium Projects, “Multi-process Architecture” ve “Site Isolation” belgeleri; Android Developers, “Processes and app lifecycle”.

Son

Bölüm 3: Süreçler

Sonraki bölüm: İş Parçacıkları · Terim sözlüğü