

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

Yarış durumu ve klasik problemler

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

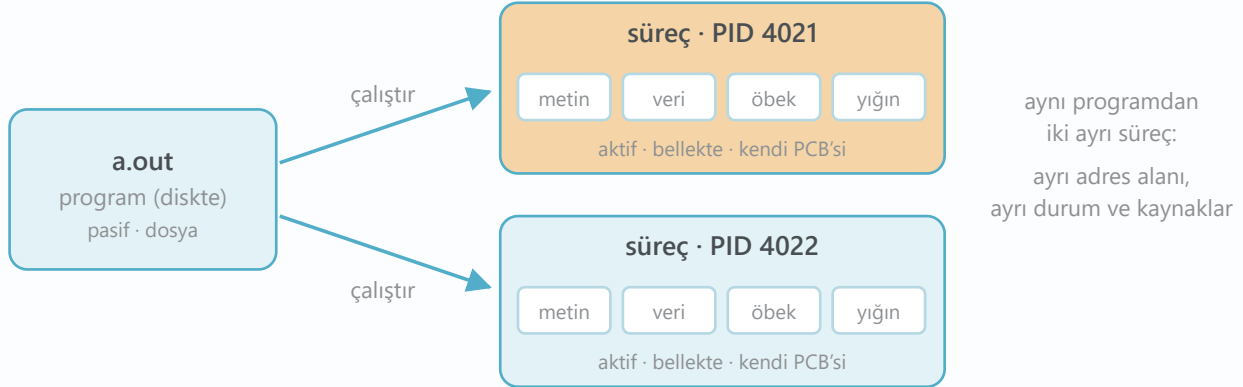
Paylaşımlı bellek, ileti geçirme, boru ve soketleri karşılaştırmak.

Eş zamanlılık sorunları

Yarış durumu, kritik bölge ve klasik problemleri tanımak (ayrıntısı Bölüm 6'da).

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 (Bölüm 4).



Bellekte Süreç Yerleşimi



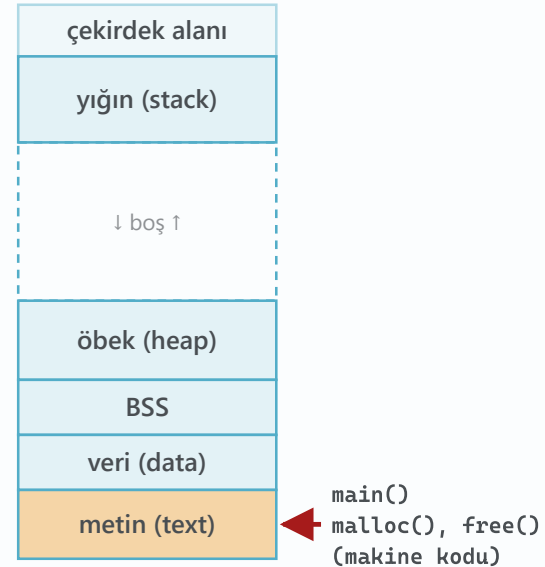
Yığın (stack) yüksek adreslerden aşağı, öbek (heap) yukarı doğru büyür. Program sayacı bellekte değil, **işlemcide bir yazmaçtır**; sürecin "şu anki konumu"nu gösterir. Gerçek düzende ASLR adresleri her çalıştırmada rastgele kaydırır.

Değişkenler Nereye Yerleşir?

```
#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 paylaşılabılır.

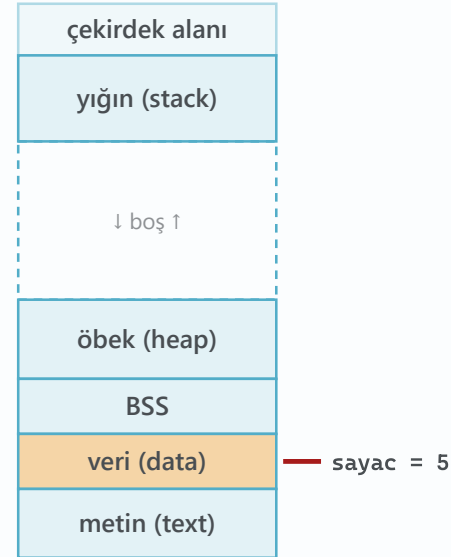
SÜREÇ KAVRAMI

Değişkenler Nereye Yerleşir?

```
#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.



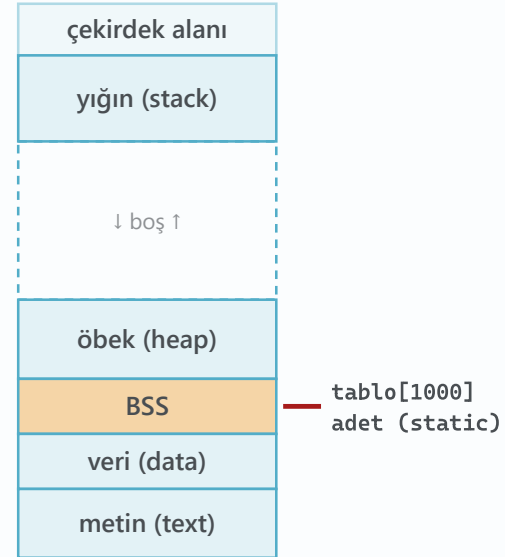
Veri: ilk değeri verilmiş global `sayac` ; değeri çalıştırılabilir dosyada saklanır.

Değişkenler Nereye Yerleşir?

```
#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.



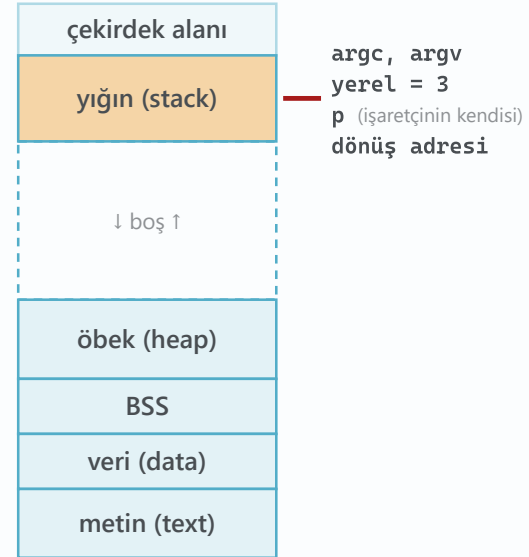
BSS: ilk değeri verilmemiş global ve `static` değişkenler; yükleyici sıfırla doldurur, dosyada yer kaplamaz.

Değişkenler Nereye Yerleşir?

```
#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.



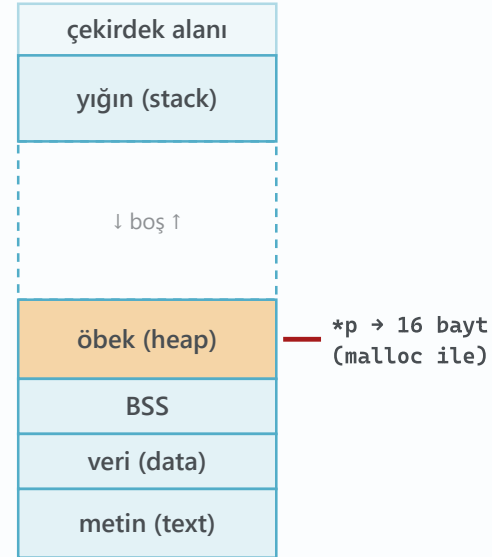
Yığın: `main` çağrı çerçevesi: parametreler, yerel değişkenler ve işaretçi `p` 'nin kendisi.

Değişkenler Nereye Yerleşir?

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

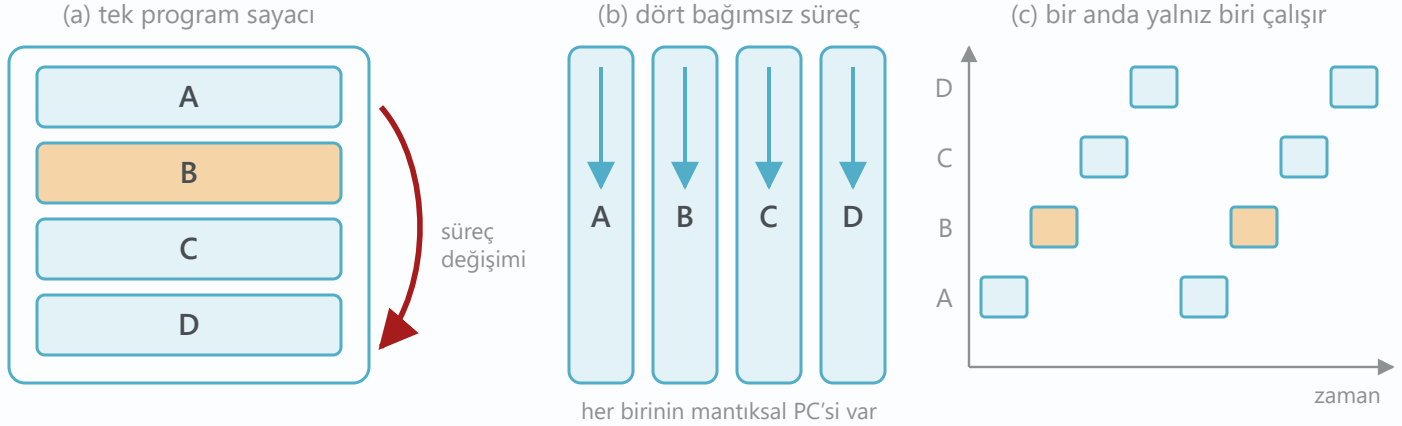
int main(int argc, char *argv[]) {
    int yerel = 3;
    static int adet;
    int *p = malloc(4 * sizeof *p);
    p[0] = yerel + sayac;
    free(p);
    return 0;
}
```

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



Öbek: `malloc` ile ayrılan 16 bayt; `p` yığında durur ama gösterdiği alan öbektir. `free` ile geri verilir.

Sözde Paralellik ve Çoklu Programlama

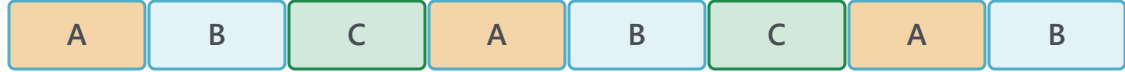


- Tek çekirdekte herhangi bir anda yalnızca **bir** süreç yürütülür; çizelgeleyici süreçler arasında hızla geçiş yaparak **sözde paralellik** yaratır.
- Geçişler tipik olarak birkaç milisaniyelik zaman dilimleriyle olur (Linux EEVDF'de taban dilim ~3 ms; eski CFS'de hedef gecikme 6–24 ms); kullanıcıya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

Eş Zamanlılık ve Paralellik

Eş zamanlılık

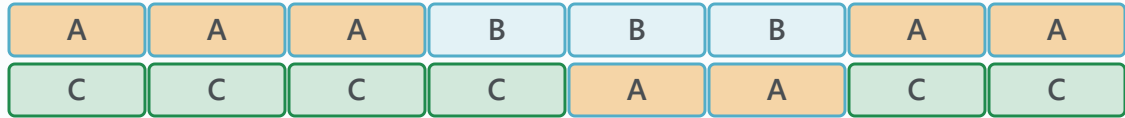
(1 çekirdek)



Paralellik çekirdek 0

(2 çekirdek)

çekirdek 1



zaman →

Eş zamanlılık (concurrency)

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

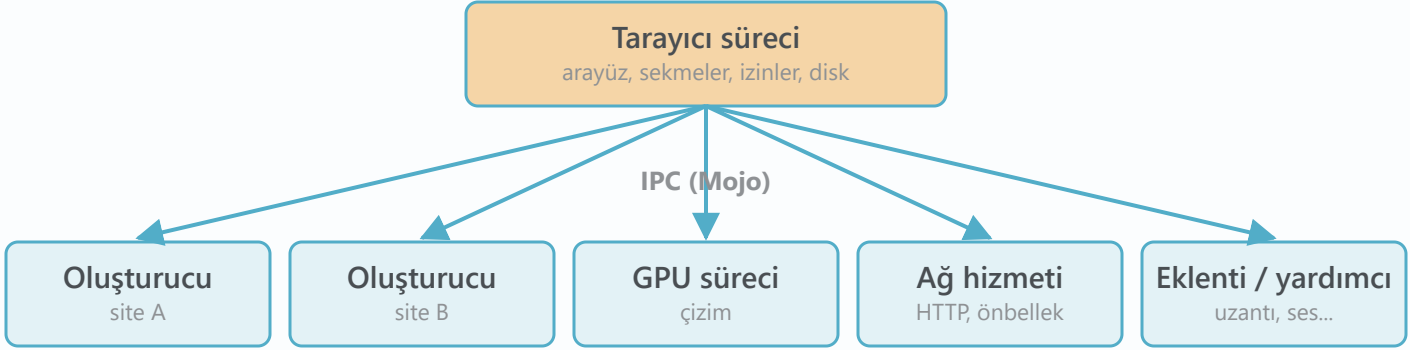
Paralellik (parallelism)

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.

SÜREÇ KAVRAMI

Örnek: Chrome'un Çok Süreçli Mimarisi



oluşturucular kum havuzunda (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.

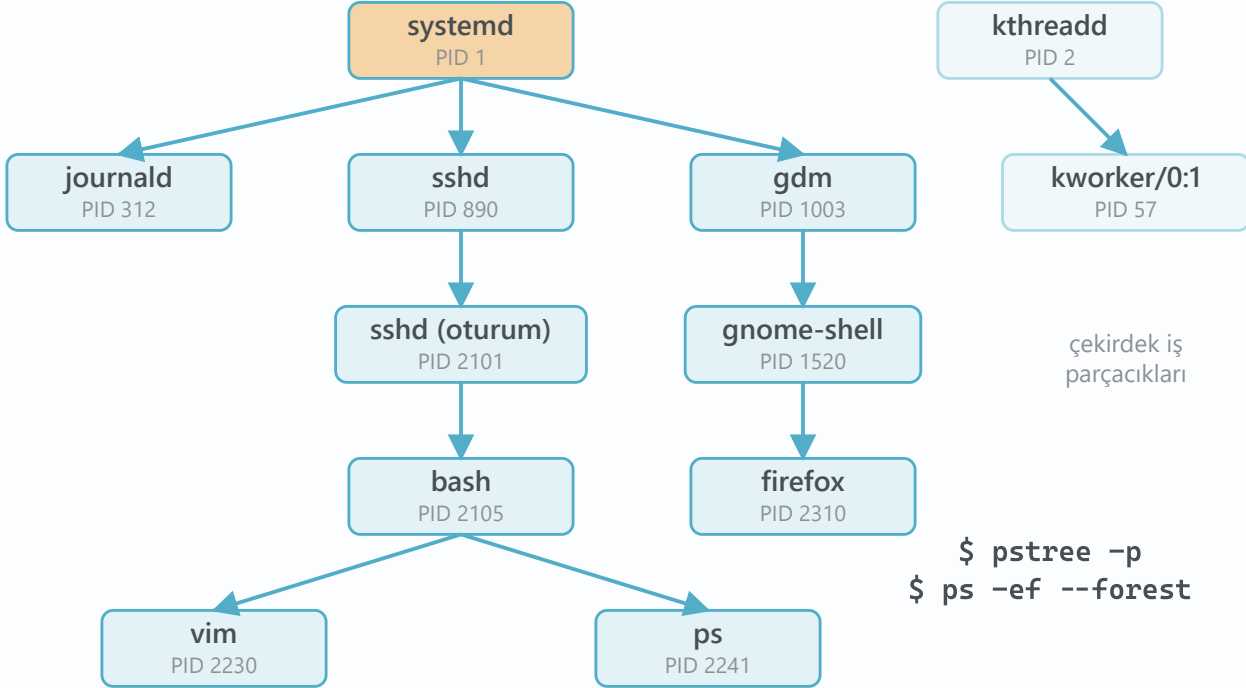
Her durumda teknik olarak aynı şey olur: var olan bir süreç, yeni süreç oluşturan bir **sistem çağrısı** yürütür.

Sistem çağrıları

Sistem	Çağrı
UNIX / Linux	<code>fork()</code> + <code>exec()</code> ; ayrıca <code>posix_spawn()</code> , Linux'ta <code>clone()</code>
Windows	<code>CreateProcess()</code> : 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 açılıшта PID 1 olarak **systemd** (ya da başka bir init) çalıştırır; PID 2 **kthreadd** çekirdek iş parçacıklarının ebeveynidir. PID değerleri örnektir.

Ebeveyn ve Çocuk İlişkisi

Kaynak paylaşımı

Çocuk ebeveyn kaynaklarının tümünü, bir kısmını 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

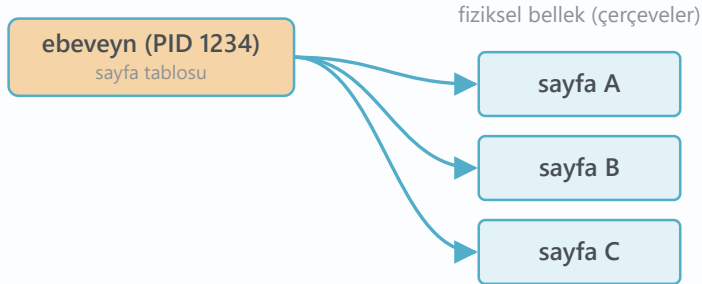
`getpid()` kendi PID'ini, `getppid()` ebeveynin PID'ini döndürür.

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

SÜREÇ OLUŞTURMA

fork(): Bir Çağrı, İki Dönüş

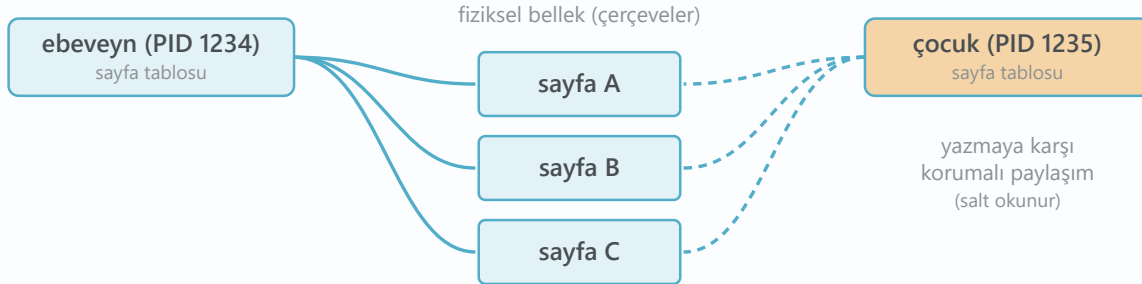
```
pid_t pid = fork();  
if (pid < 0)      { perror("fork"); exit(1); }  
else if (pid == 0) { /* çocuk: fork 0 döndürdü */ }  
else             { /* ebeveyn: pid = çocuğun PID'i */ }
```



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

fork(): Bir Çağrı, İki Dönüş

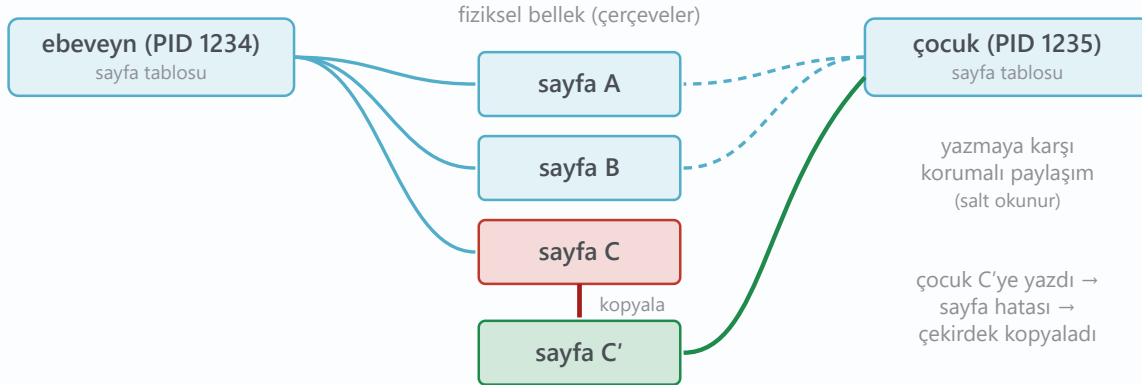
```
pid_t pid = fork();  
if (pid < 0)      { perror("fork"); exit(1); }  
else if (pid == 0) { /* çocuk: fork 0 döndürdü */ }  
else             { /* ebeveyn: pid = çocuğun PID'i */ }
```



fork() : çekirdek PCB'yi ve sayfa tablosunu kopyalar; sayfalar **kopyalanmaz**, ikisine de salt okunur eşlenir. Ebeveyn çocuğun PID'ini, çocuk 0 alır.

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) {
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);
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()

```
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);
15    return 0;
16 }
```

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    }
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

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();
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

ebeveyn.pid = 1235

çocuk.x = 10

çocuk.pid = 0

ÇIKTI

ebeveyn · 1234
bekliyorçocuk · 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) {
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 = 15

çocuk.pid = 0

ÇIKTI

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;
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

çocuk: x=15

ebeveyn · 1234
bekliyor

çocuk · 1235
çalışıyor

Çocuk yazdırır.

Adım 6 / 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);
15    return 0;
16 }
```

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    }
13    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
çalışıyorçocuk · 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);
14    ▶ printf("ebeveyn: x=%d\n", x);
15    return 0;
16 }
```

DEĞİŞKENLER

ebeveyn.x = 10

ebeveyn.pid = 1235

ÇIKTI

```
çocuk: x=15
ata: x=10
```

ebeveyn · 1234
çalışıyorçocuk · 1235
biçildiEbeveyn yazdırır: kendi 'i hâlâ 10.

Adım 9 / 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);
15    return 0;
16 }
```

DEĞİŞKENLER

ebeveyn.x = 10

ebeveyn.pid = 1235

ÇIKTI

```
çocuk: x=15
ata: x=10
```

ebeveyn · 1234
sonlandıçocuk · 1235
biçildi

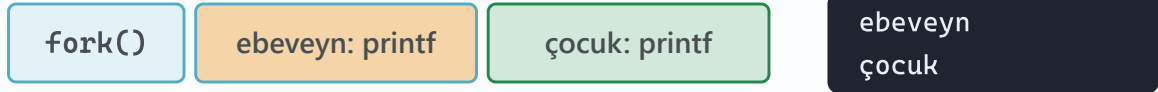
Ebeveyn da sonlanır. `wait()` sayesinde çıktı sırası her çalıştırmada aynıdır.

SÜREÇ OLUŞTURMA

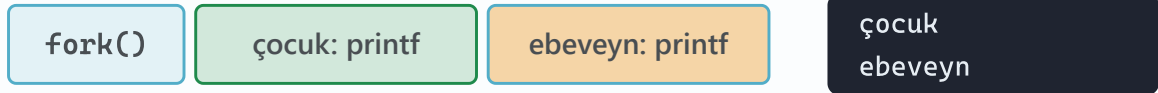
wait() Olmadan Çıktı Sırası Belirsizdir

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

Çizelge 1



Çizelge 2

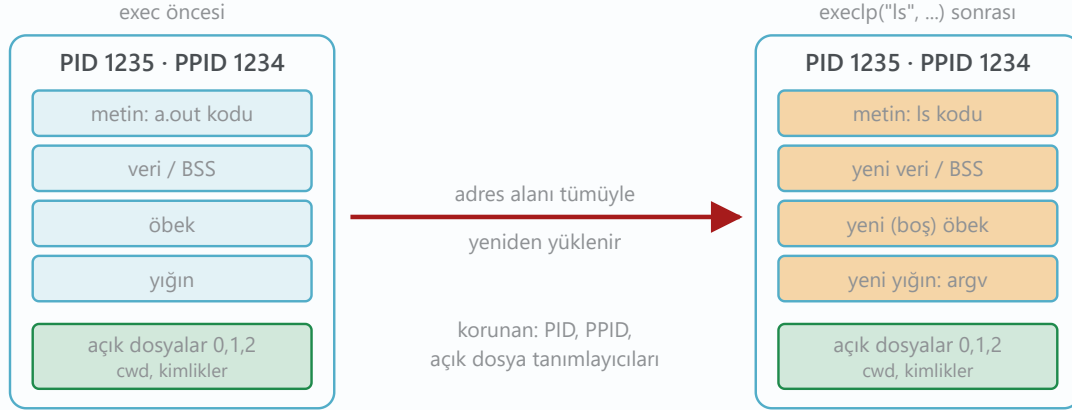


zaman (tek çekirdek) →

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.

SÜREÇ OLUŞTURMA

exec() Ailesi



Çağrı	Argümanlar	PATH araması	Ortam
<code>execl</code> / <code>execv</code>	liste / dizi	hayır	devralınır
<code>execlp</code> / <code>execvp</code>	liste / dizi	evet	devralınır
<code>execl_e</code> / <code>execve</code>	liste / dizi	hayır	açıkça verilir

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

SÜREÇ OLUŞTURMA

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

eski kod, veri, öbek,
yığın: silinir

korunur:

PID ve ebeveyn
açık fd'ler (cloexec hariç)
çalışma dizini, kullanıcı
sinyal maskesi



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.

SÜREÇ OLUŞTURMA

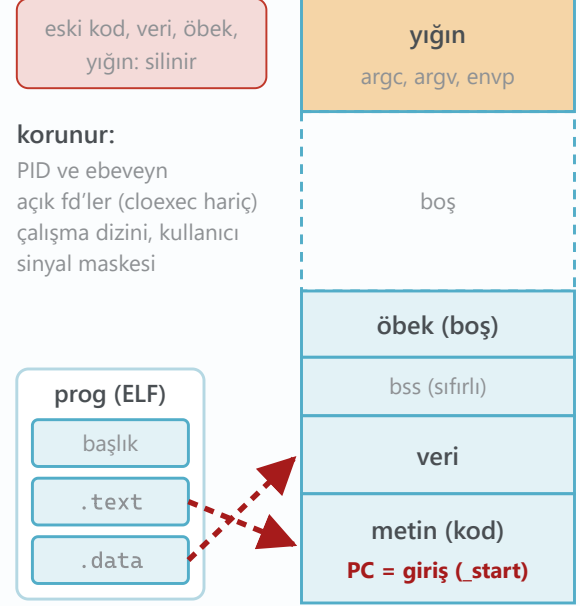
exec Çekirdekte Ne Yapar?

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

2. Eski adres alanını bırak.

Kod, veri, öbek ve yığın eşlemeleri silinir; süreçteki diğer iş parçacıkları sonlandırılır. Bu noktadan sonra geri dönecek program yoktur.

Adım 2 / 6



eşleme tembeldir: sayfa ilk erişimde okunur

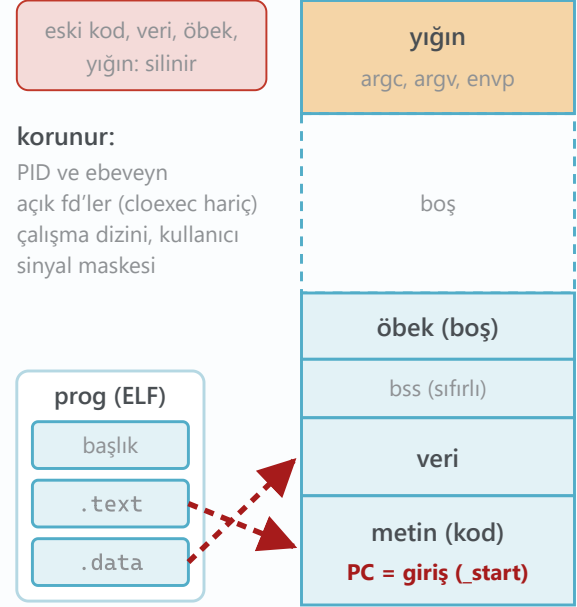
Diskteki program pasif bir dosyadır; `exec` onu çalışan bir sürecin adres alanına dönüştürür. Linux'ta tüm `exec` ailesi sonunda `execve()` 'ye iner.

SÜREÇ OLUŞTURMA

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

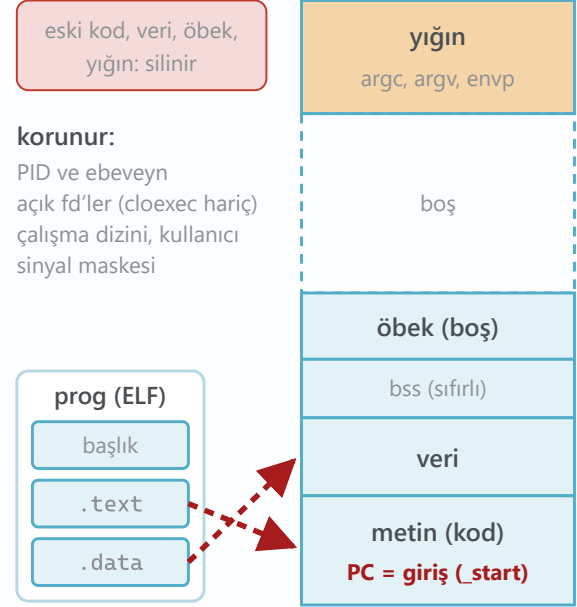
Diskteki program pasif bir dosyadır; exec onu çalışan bir sürecin adres alanına dönüştürür. Linux'ta tüm exec ailesi sonunda `execve()` 'ye iner.

SÜREÇ OLUŞTURMA

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

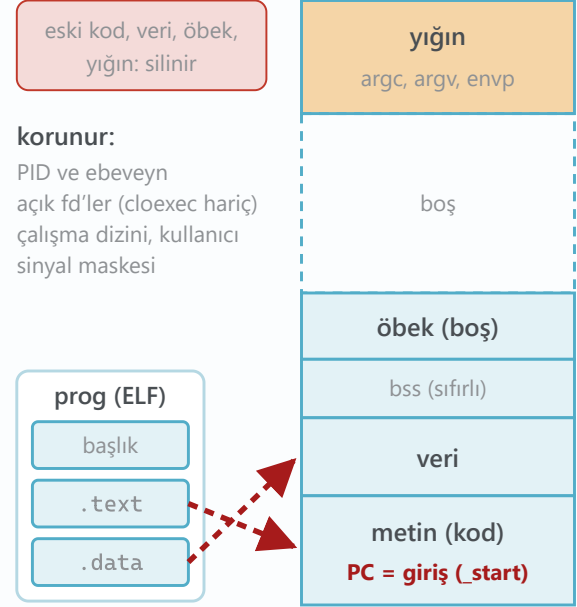
Diskteki program pasif bir dosyadır; `exec` onu çalışan bir sürecin adres alanına dönüştürür. Linux'ta tüm `exec` ailesi sonunda `execve()`'ye iner.

SÜREÇ OLUŞTURMA

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, öbek 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

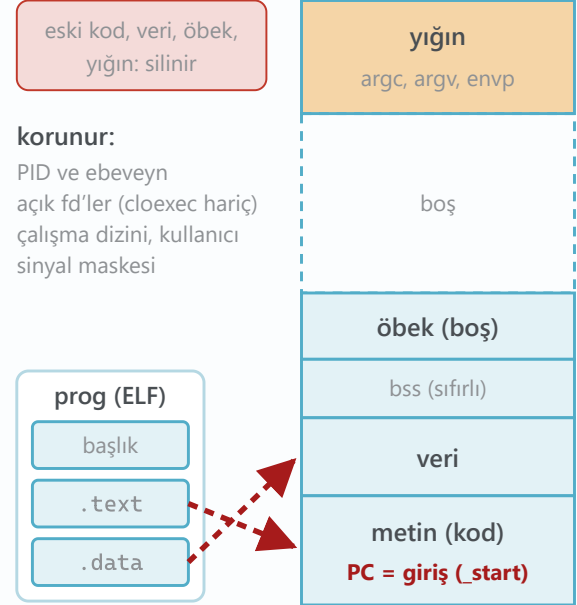
Diskteki program pasif bir dosyadır; exec onu çalışan bir sürecin adres alanına dönüştürür. Linux'ta tüm exec ailesi sonunda `execve()` 'ye iner.

SÜREÇ OLUŞTURMA

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, öbek 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 temeldir: 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) {
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);
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.

Adım 1 / 6

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);
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.

Adım 2 / 6

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);
15     printf("bitti: %d\n", WEXITSTATUS(st));
16     return 0;
17 }
```

DEĞİŞKENLER

ebeveyn.pid = 1235

çocuk = ls -l

ÇIKTI

ebeveyn · 1234
bekliyorçocuk · 1235
exec: ls

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

Adım 3 / 6

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);
15     printf("bitti: %d\n", WEXITSTATUS(st));
16     return 0;
17 }
```

DEĞİŞKENLER

ebeveyn.pid = 1235

çocuk = ls -l

ÇIKTI

(ls -l çıktısı)

ebeveyn · 1234
bekliyorçocuk · 1235
ls çalışıyor

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

Adım 4 / 6

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);
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.

Adım 5 / 6

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);
15    ▶ 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

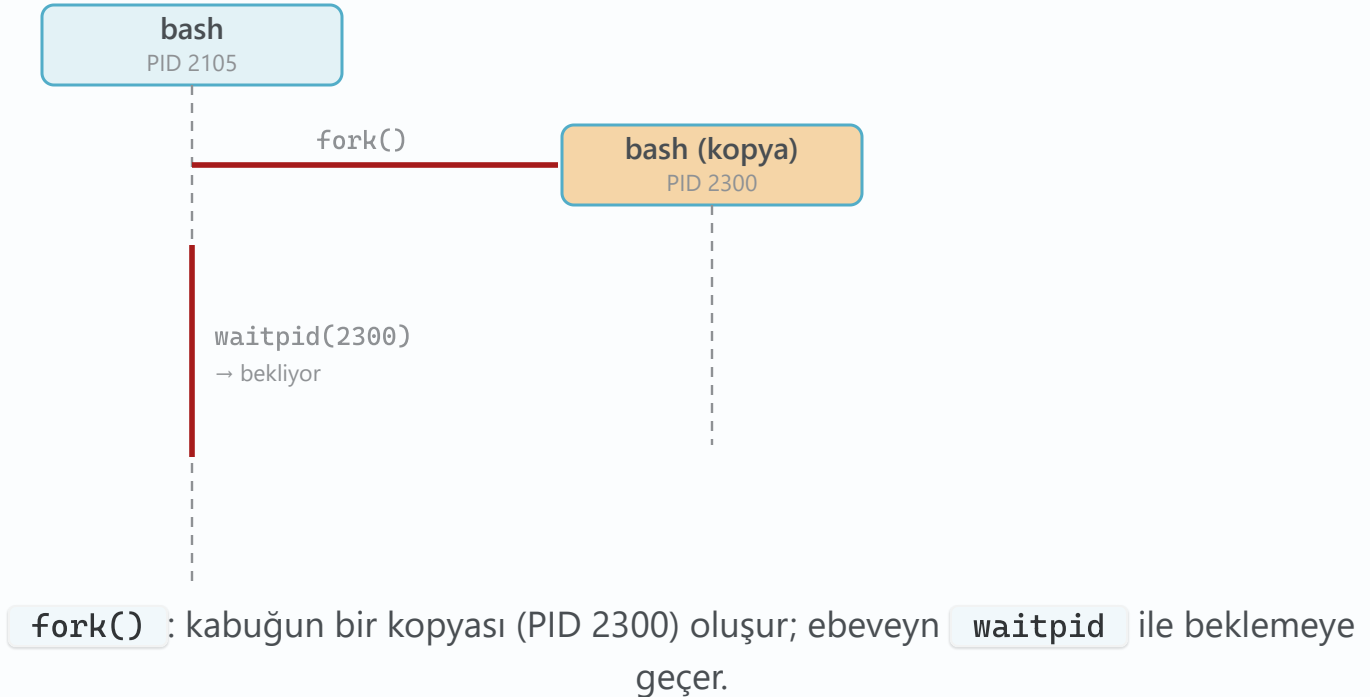
WEXITSTATUS(st) = 0 yazdırılır.

Adım 6 / 6

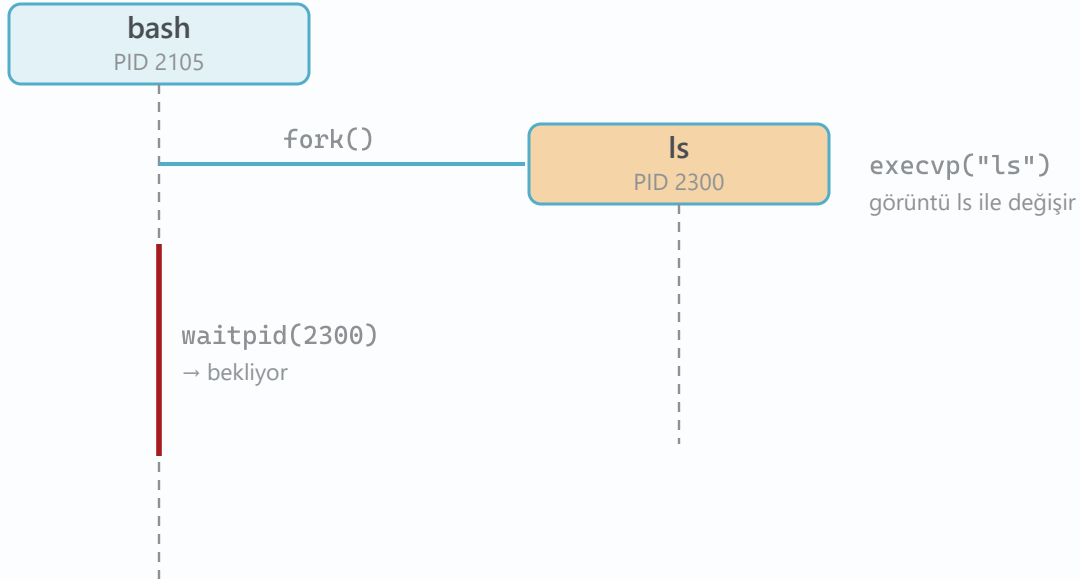
Kabuk Döngüsü: **fork** → **exec** → **wait**



Kabuk Döngüsü: **fork** → **exec** → **wait**

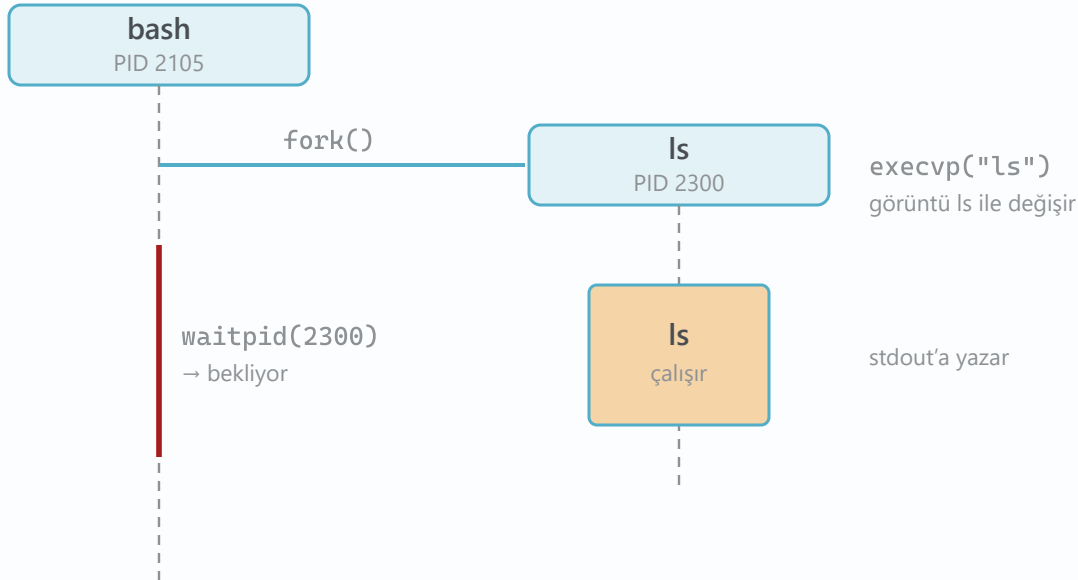


Kabuk Döngüsü: fork → exec → wait



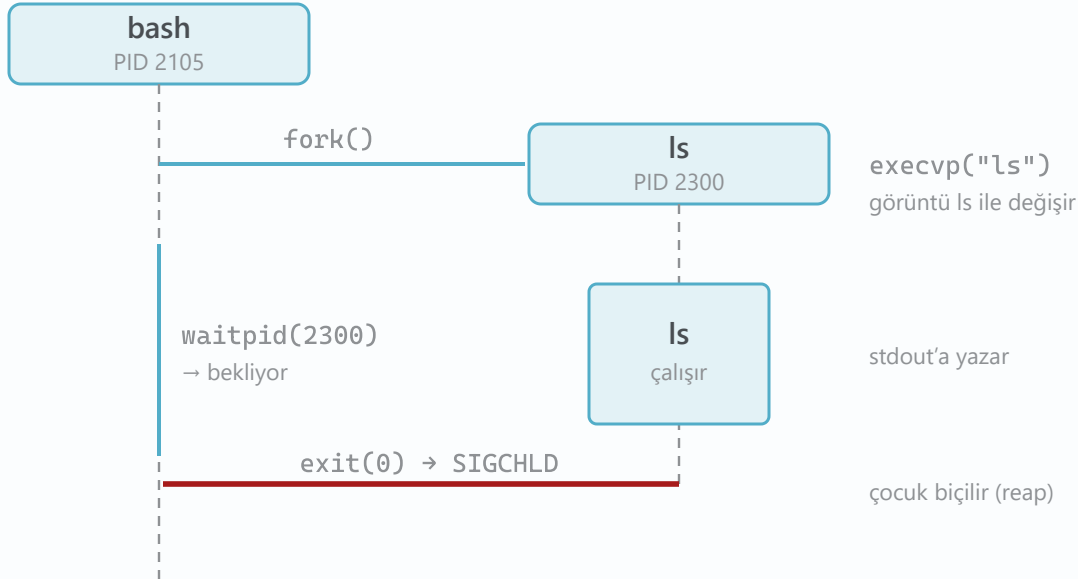
Çocuk `execvp("ls", argv)` çağırır; görüntüsü `ls` ile değişir.

Kabuk Döngüsü: **fork** → **exec** → **wait**



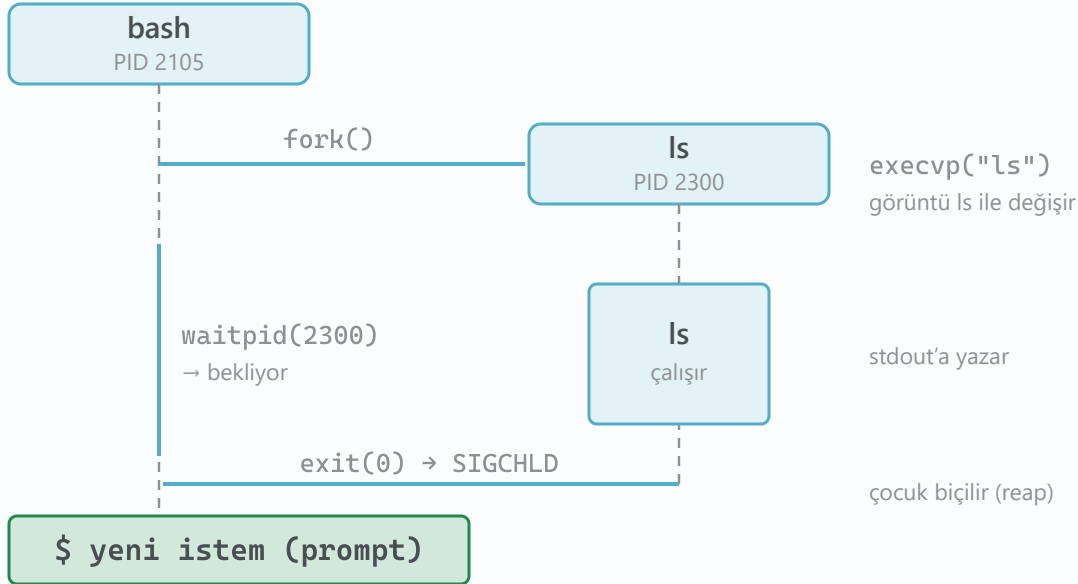
`ls` çalışır, miras aldığı terminal stdout'una yazar.

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 Ayır? 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

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, izin, öncelik) hazırlar. Windows'un `CreateProcess` 'i bunları parametre ister.

Neden fork ve exec Ayır? 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

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.

Adım 2 / 6

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

Neden fork ve exec Ayır? 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

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.

Adım 3 / 6

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

Neden fork ve exec Ayır? 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

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.

Adım 4 / 6

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

Neden fork ve exec Ayır? 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         ▶ execvp("wc", "wc", "-l", "notlar.txt", NULL);
9         perror("execvp");
10        exit(1);
11    }
12    waitpid(pid, NULL, 0);
13    printf("bitti\n");
14    return 0;
15 }
```

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.

Adım 5 / 6

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

Neden fork ve exec Ayır? 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, izin, öncelik) hazırlar.

Windows'un `CreateProcess` 'i bunları parametre ister.

SÜREÇ SONLANDIRMA

Süreç Sonlandırma

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

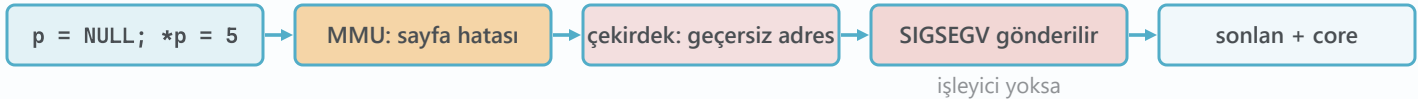
- `exit(durum)` : C kütüphanesi tamponları boşaltır, `atexit` işleyicilerini çalıştırır, sonra `_exit` sistem çağrısı.
- `abort()` : çağıran süreç kendine SIGABRT gönderir, anormal biter (çekirdek dökümü oluşabilir). Başka süreci sonlandırmak için `kill()` kullanılır.
- Çekirdek sürecin belleğini, açık 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
Tamsayı 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.



Sinyallerle Sonlandırma: kill, pkill, killall

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

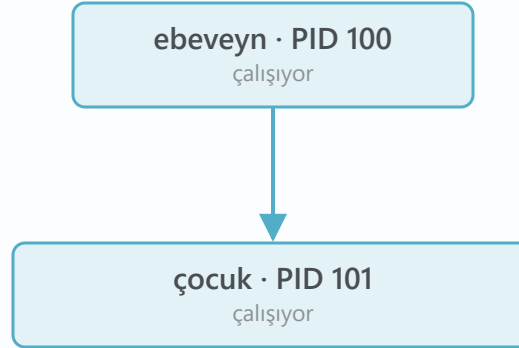
```
$ ps -e | grep yaz
1234 ?  00:00:03 yaz
$ kill 1234          # SIGTERM
$ kill -9 1234       # SIGKILL
$ pkill yaz          # ada göre
$ killall yaz
```

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

Numaralar x86/ARM Linux içindir; taşınabilir kodda adları kullanın.

Zombi ve Yetim Süreçler

Senaryo 1: çocuk önce biter



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.

Zombi ve Yetim Süreçler

Senaryo 1: çocuk önce biter



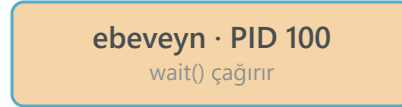
Çocuk `exit()` ile bitti ama ebeveyn henüz `wait()` çağırmadı: çocuk **zombi**dir. Belleği serbest; süreç tablosunda PID ve çıkış durumu bekler (`ps` 'te `Z` / `<defunct>`).

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

Senaryo 1: çocuk önce biter



durum=0 alındı

PID 101 süreç tablosundan silindi

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

Adım 3 / 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

Senaryo 2: ebeveyn önce biter

systemd · PID 1
init / subreaper

ebeveyn (100) sonlandı

çocuk · PID 101

YETİM · PPID artık yok

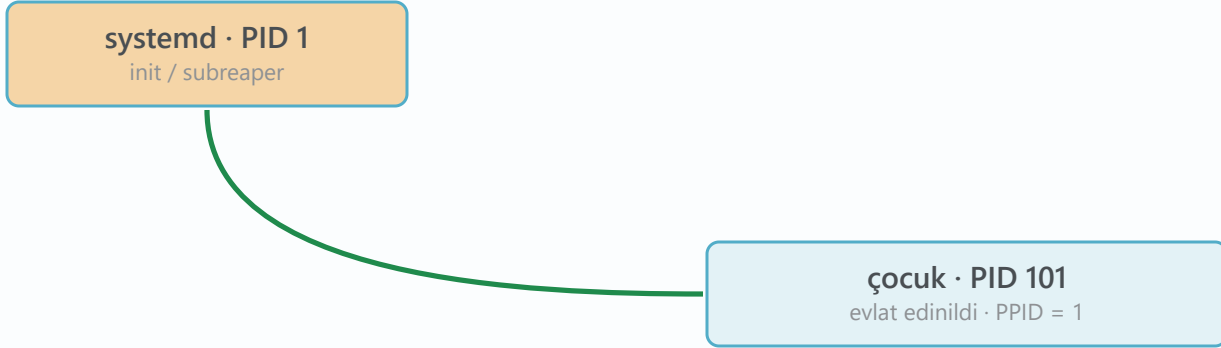
Senaryo 2: ebeveyn (100) çocuktan önce sonlandı. Çocuk (101) hâlâ çalışıyor: **yetim** süreç.

Adım 4 / 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

Senaryo 2: ebeveyn önce biter



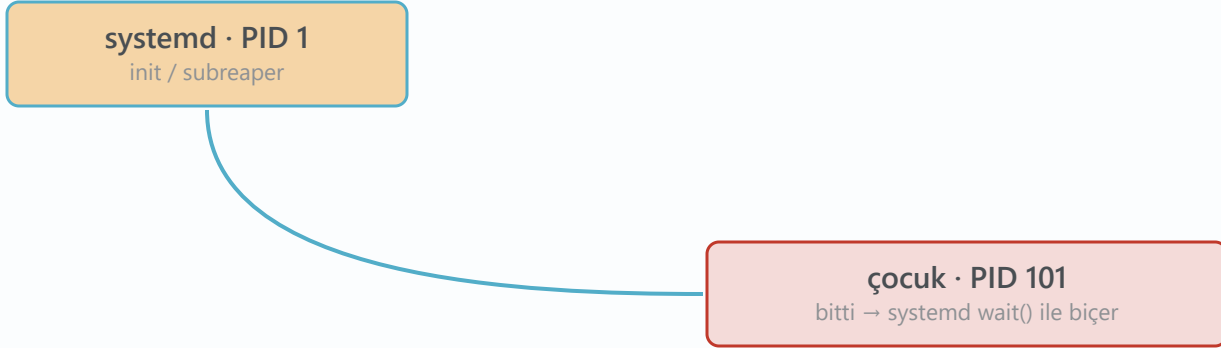
Çekirdek yetimi `init` / `systemd` 'ye (PID 1) ya da tanımlı bir alt-biçici (subreaper) sürece bağlar; `getppid()` artık 1 döndürür.

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

Senaryo 2: ebeveyn önce biter

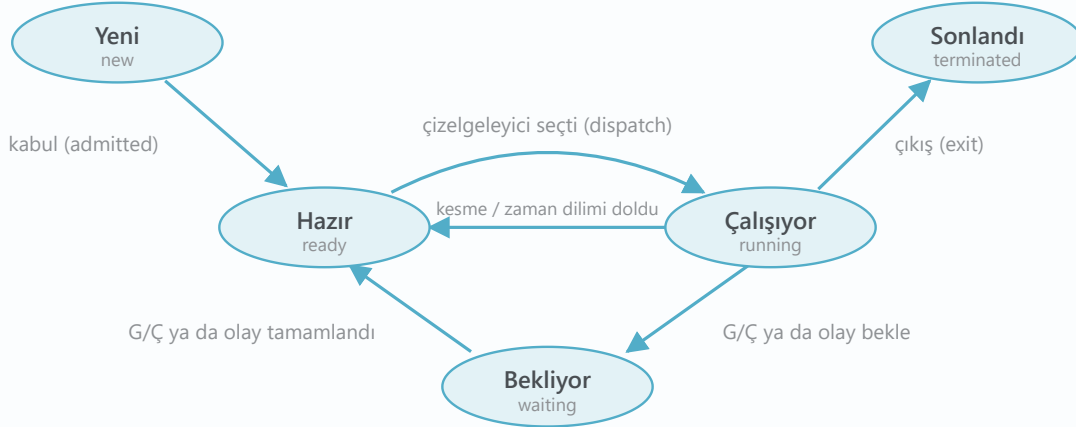
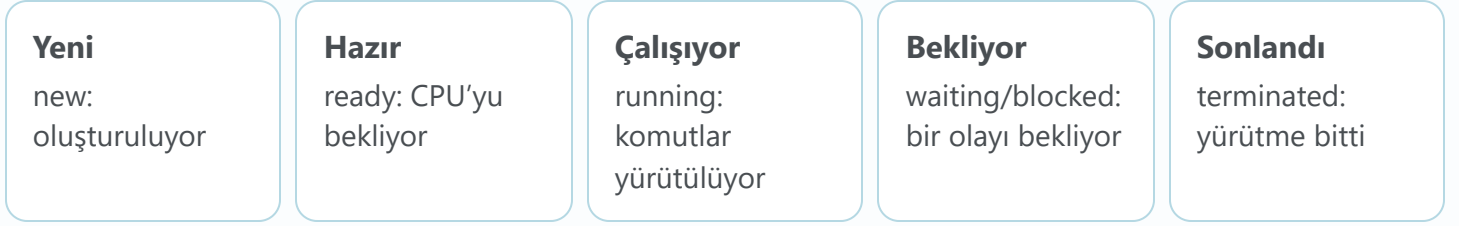


Yetim bitince yeni ebeveyni onu `wait()` ile biçer; böylece zombi birikmez.

Adım 6 / 6

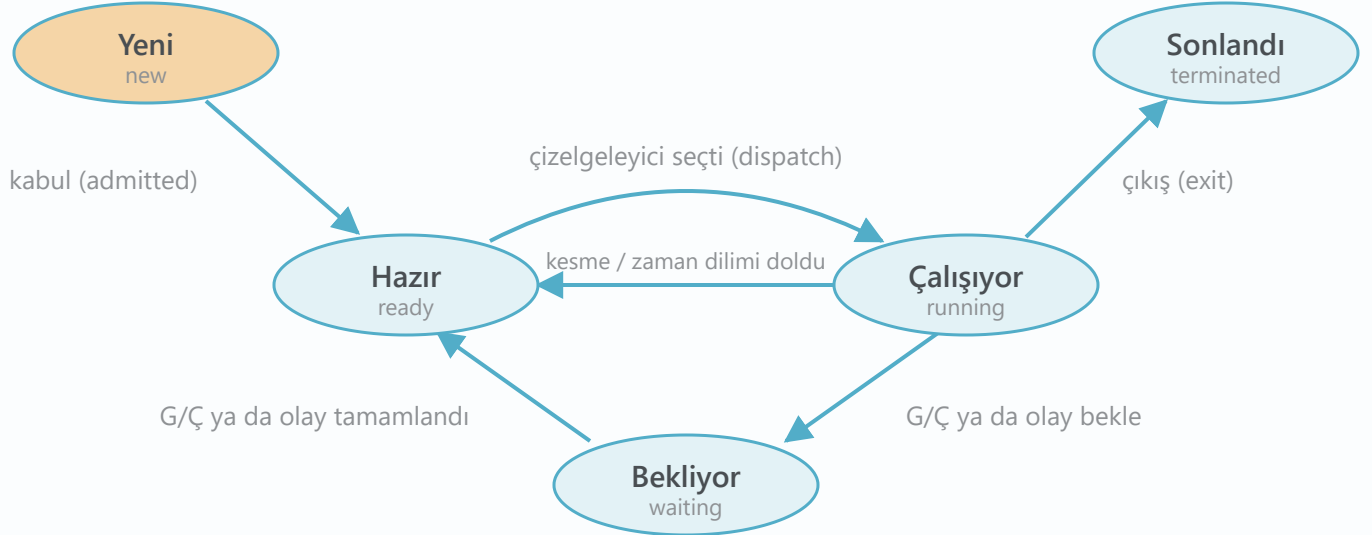
Zombi kaynak harcamaz ama PID tüketir; `wait` çağırmayan uzun ömürlü bir sunucu binlerce zombi biriktirip PID'leri tüketebilir. Çözüm: `wait` / `waitpid` ya da SIGCHLD işleyicisi.

Süreç Durumları (Beş Durum Modeli)



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

Durum Geçişleri Adım Adım

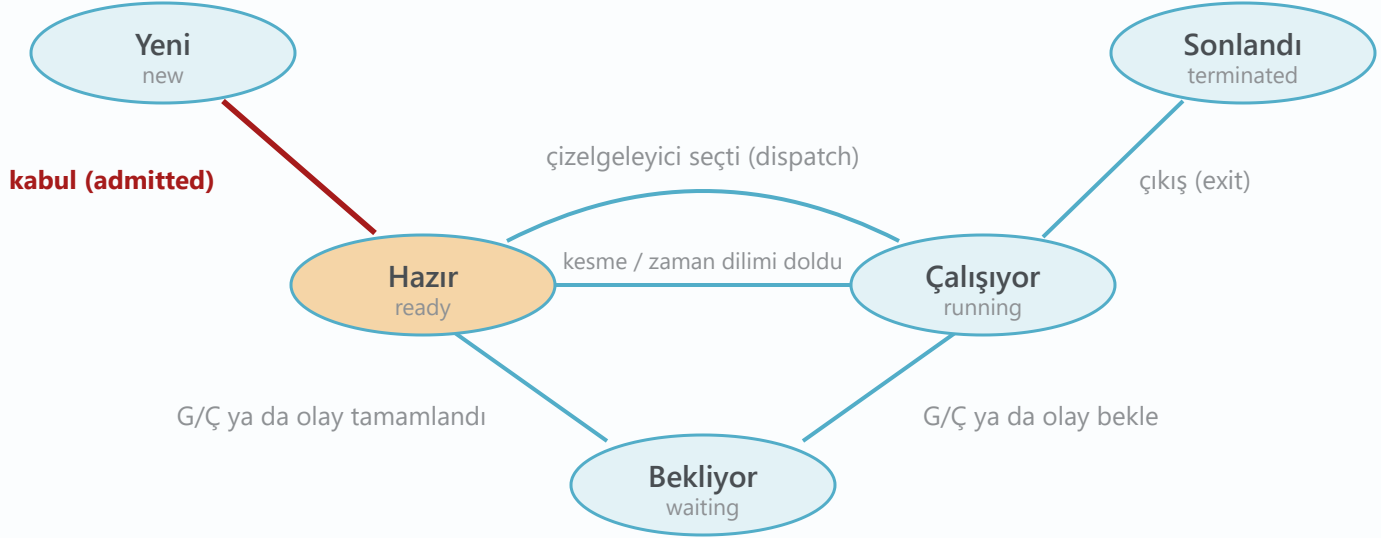


Kullanıcı programı başlattı: PCB oluşturuluyor, bellek ayrılıyor → **Yeni**.

Adım 1 / 9

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

Durum Geçişleri Adım Adım

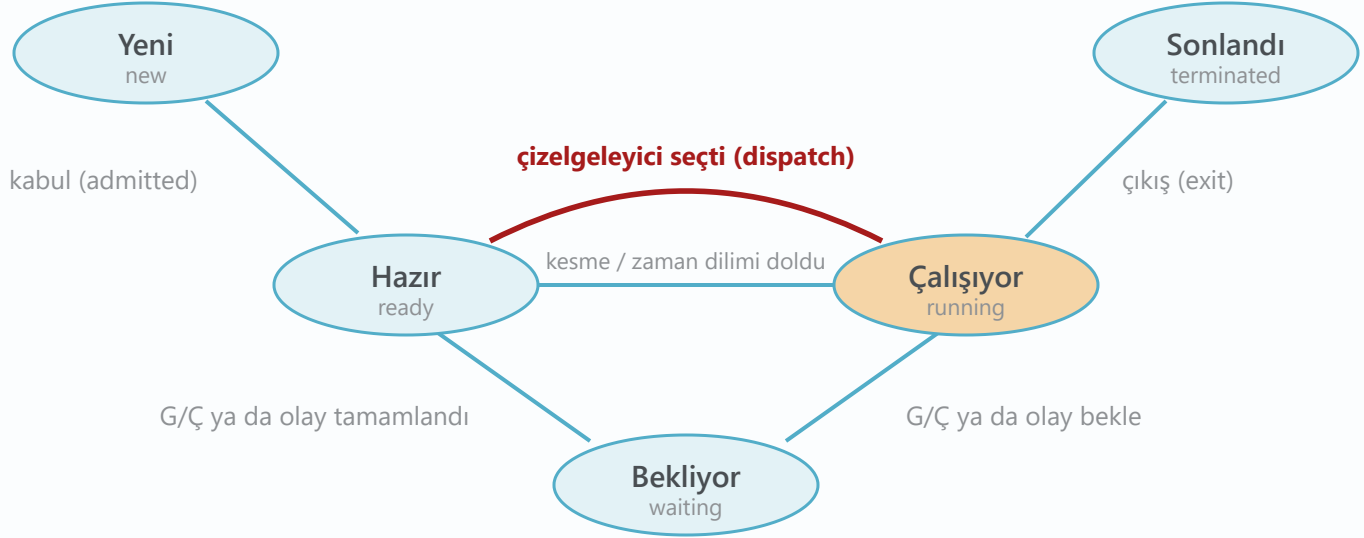


Çekirdek süreci kabul etti ve hazır kuyruğuna koydu → **Hazır**.

Adım 2 / 9

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

Durum Geçişleri Adım Adım

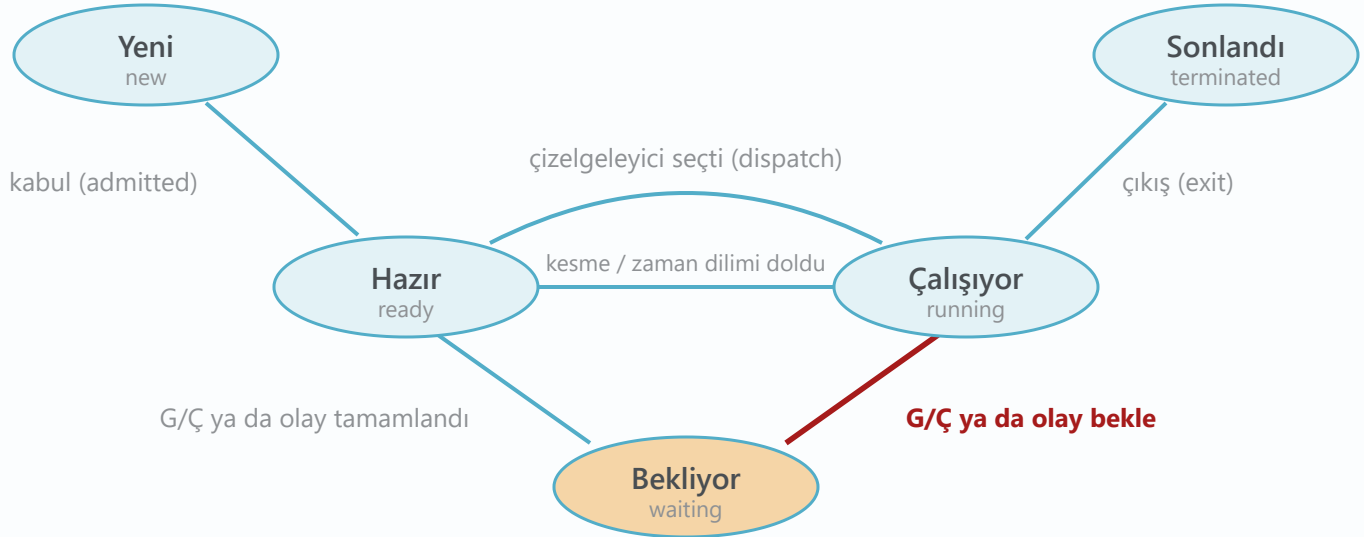


Çizelgeleyici bu süreci seçti, bağlamı yükledi → **Çalışıyor**.

Adım 3 / 9

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

Durum Geçişleri Adım Adım

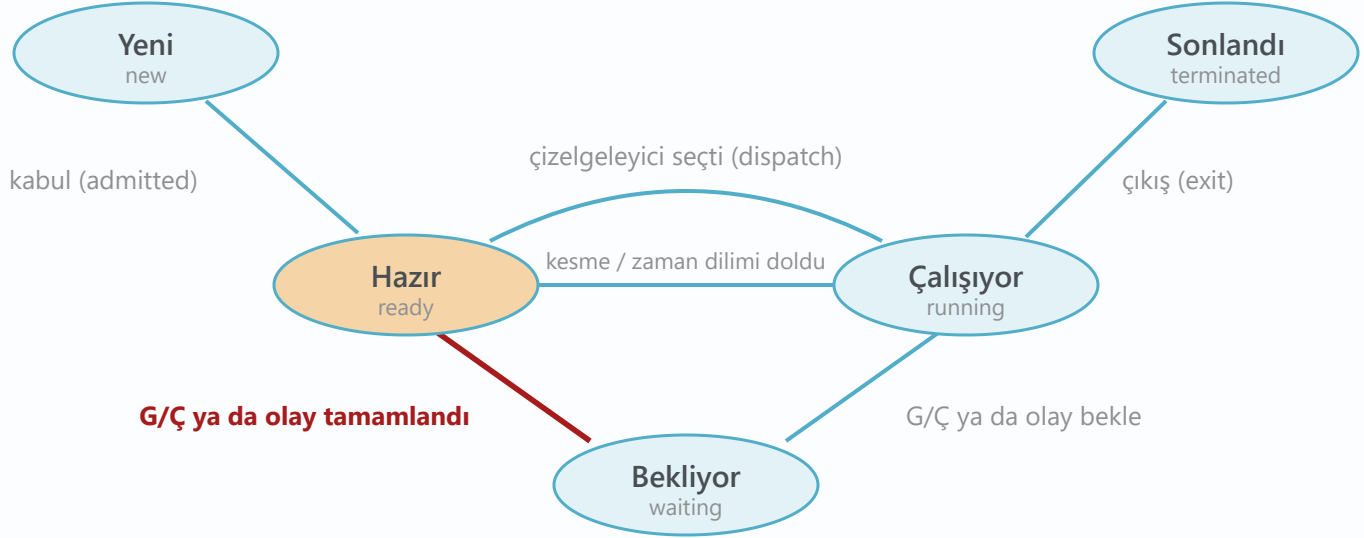


Süreç diskten `read()` istedi; veri gelene kadar CPU'yu bırakır → **Bekliyor**.

Adım 4 / 9

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

Durum Geçişleri Adım Adım

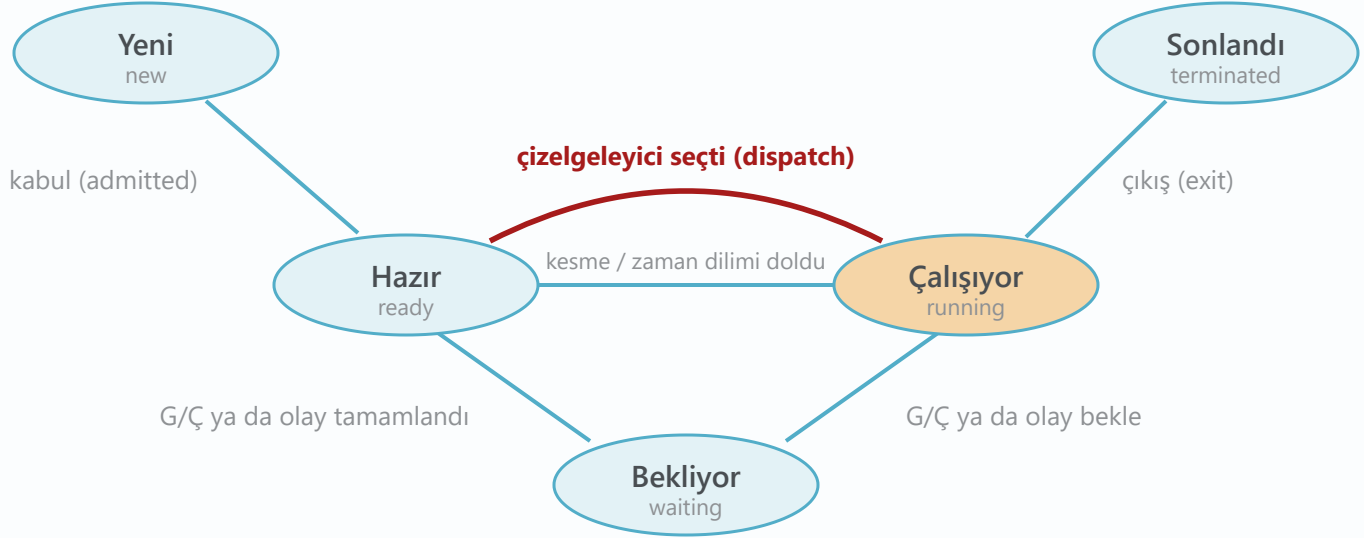


Disk kesmesi: veri hazır. Süreç doğrudan çalışmaz, hazır kuyruğuna döner → **Hazır**.

Adım 5 / 9

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

Durum Geçişleri Adım Adım

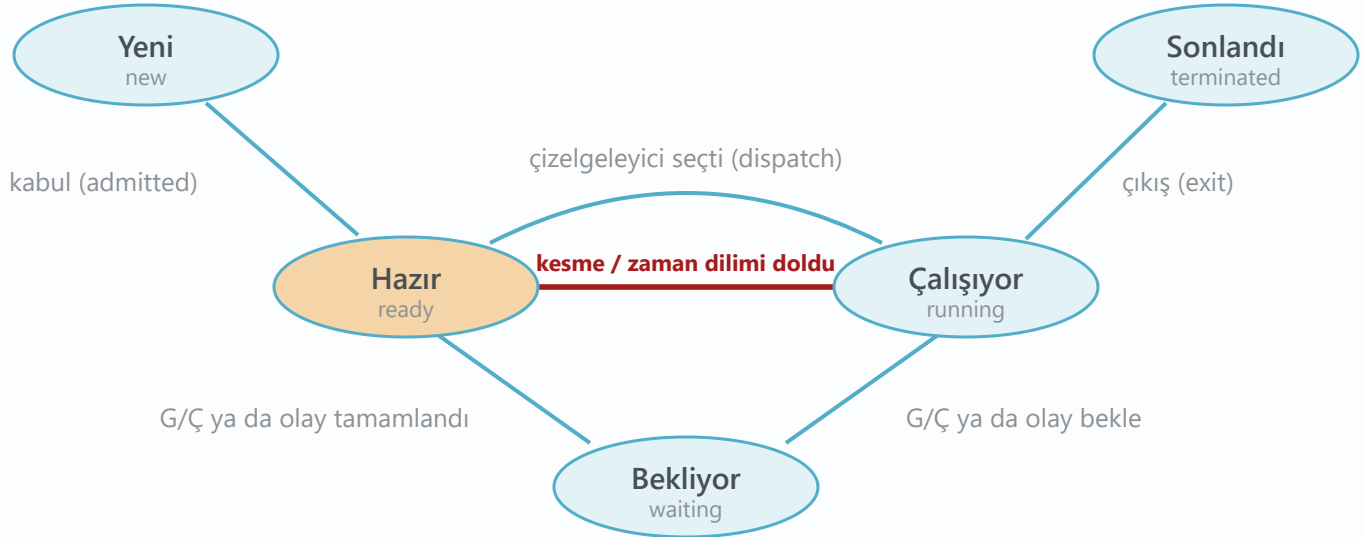


Yeniden seçildi → **Çalışıyor.**

Adım 6 / 9

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

Durum Geçişleri Adım Adım



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

Adım 7 / 9

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

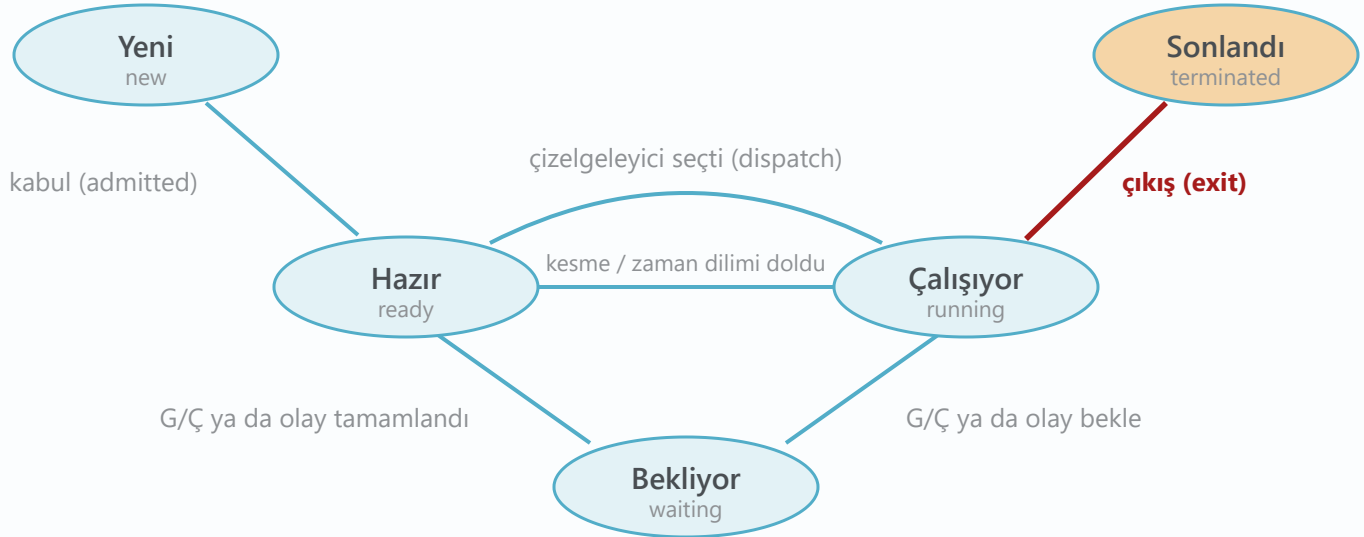
Durum Geçişleri Adım Adım



Adım 8 / 9

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

Durum Geçişleri Adım Adım



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

Adım 9 / 9

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

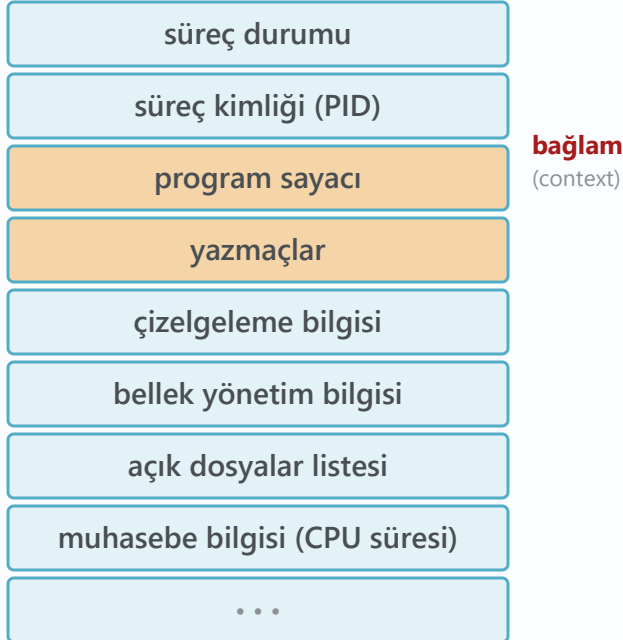
Linux'ta Süreç Durumları

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.

DURUMLAR VE PCB

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 izin · 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

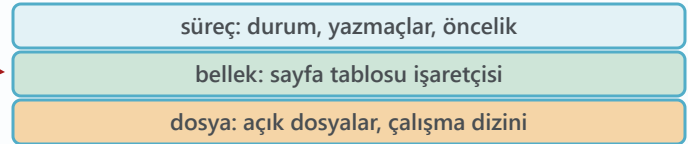
`alarm()` ile istenen SIGALRM'nin teslim zamanı.

süreç tablosu



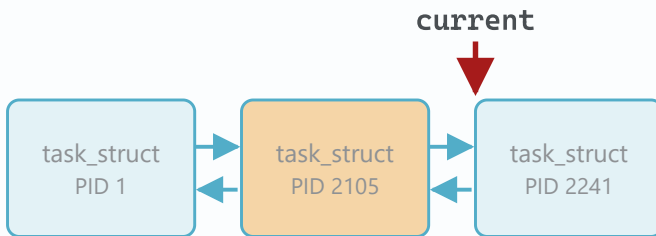
PID ile indislenen dizi ya da liste

PCB (PID 1033)



Linux'ta PCB: task_struct

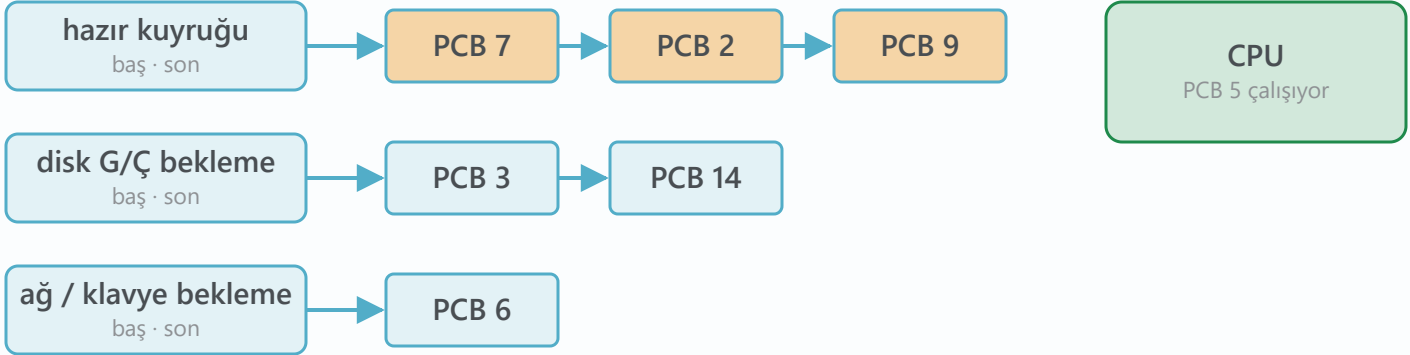
```
struct task_struct {           /* <linux/sched.h> */
    pid_t pid;                 /* süreç kimliği */
    unsigned int __state;      /* süreç durumu */
    int prio;                  /* öncelik */
    struct sched_entity se;     /* CFS/EEVDF bilgisi */
    struct task_struct *parent; /* ebeveyn 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.

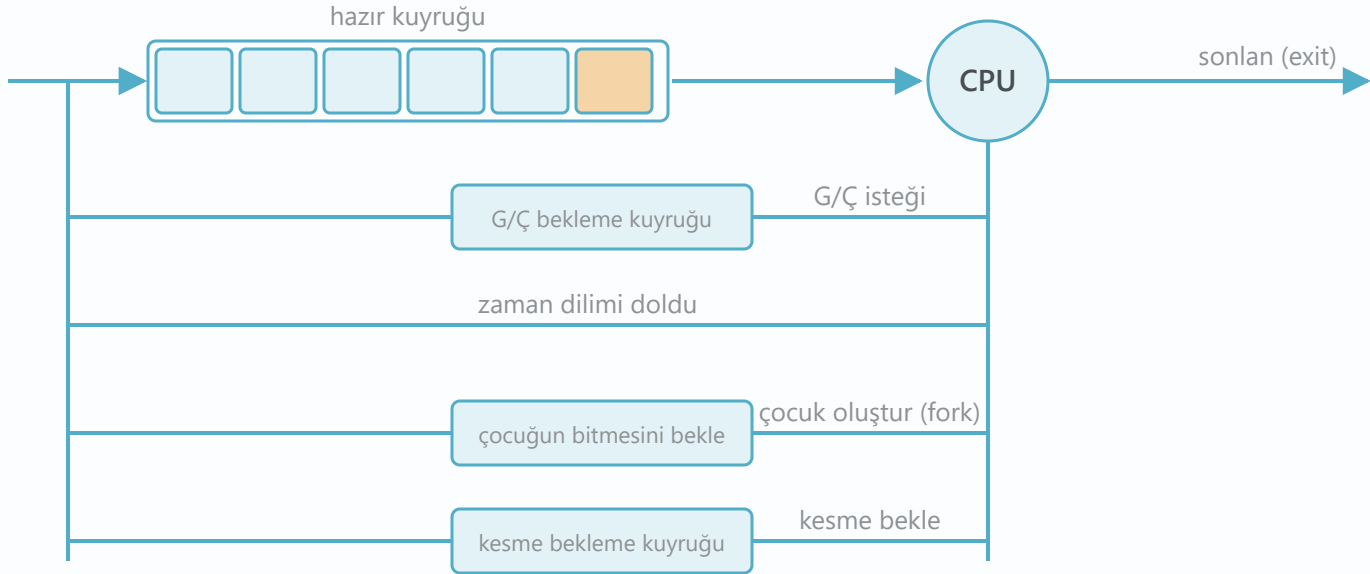
Hazır ve Bekleme Kuyrukları



kuyruklar PCB'lere bağlı listeler olarak tutulur

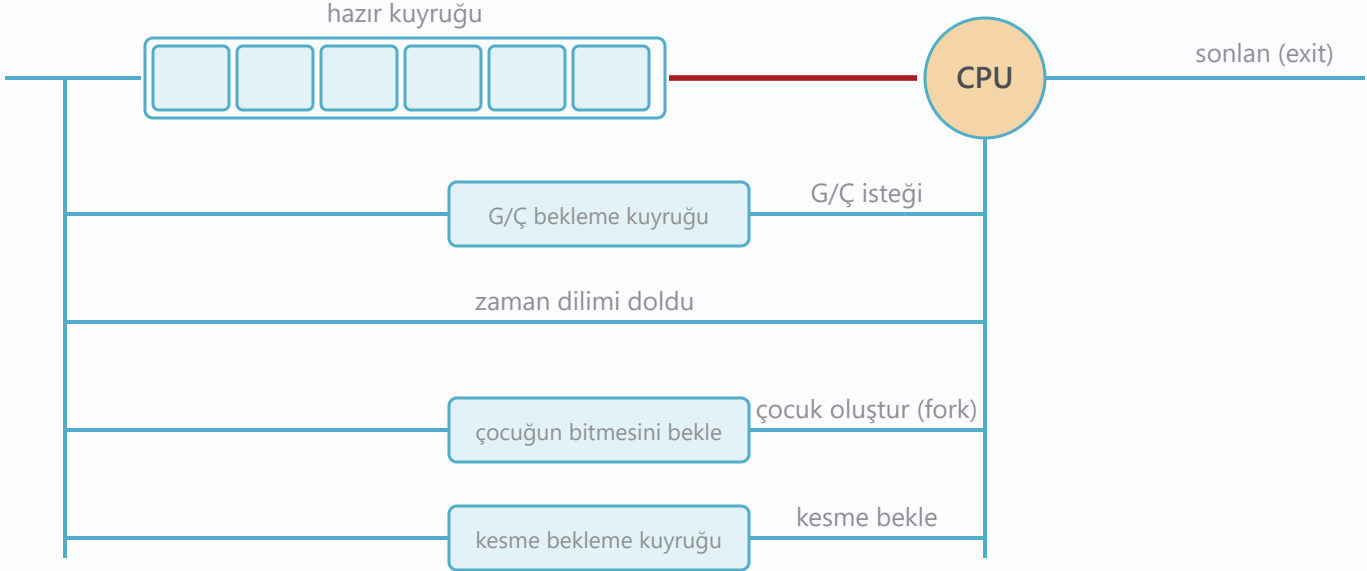
- **Çizelgeleyici** (scheduler) hazır süreçler arasından CPU'ya geçecek olanı seçer; amaç CPU'yu meşgul tutmak ve kullanıcıya hızlı yanıt vermektir.
- **Hazır kuyruğu**: bellekte olup yürütülmeyi bekleyen süreçler. **Bekleme kuyrukları**: belirli bir olayı (G/Ç bitişi, çocuğun sonlanması...) bekleyenler; her olay türü için ayrı kuyruk vardır.
- Linux her çekirdeğe ayrı bir çalışma kuyruğu (runqueue) atar; CFS/EEVDF bunu kırmızı-siyah ağaçla tutar. Algoritmalar Bölüm 5'te.

Kuyruk Diyagramı: Bir Sürecin Yolculuğu



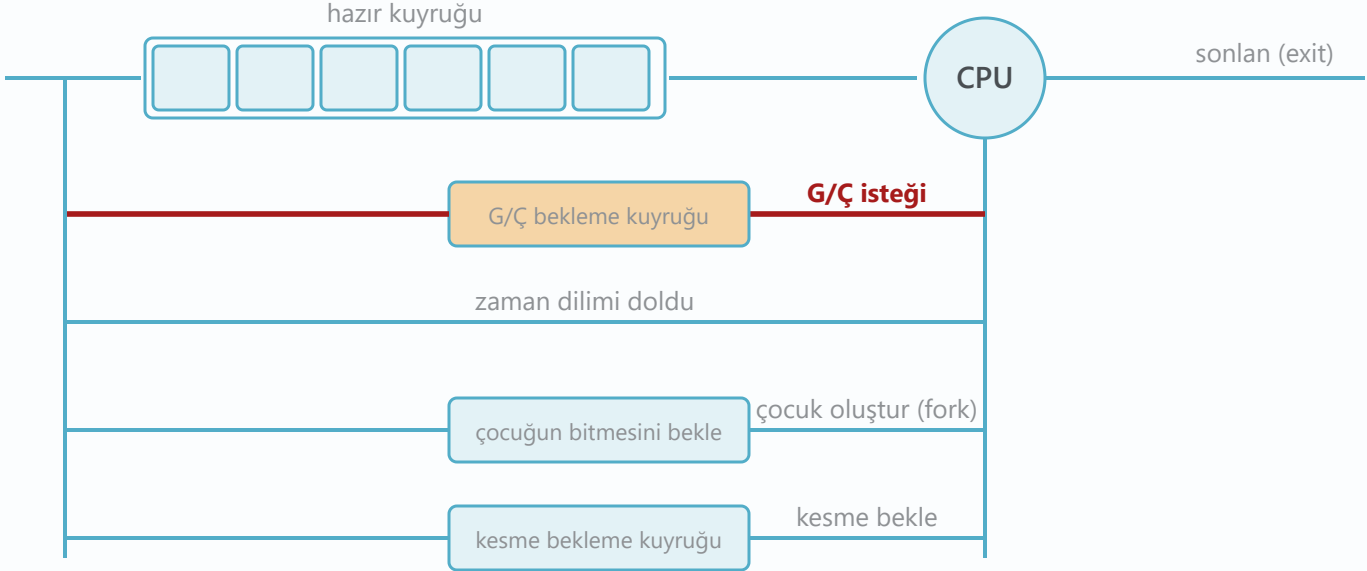
Yeni süreç hazır kuyruğunun sonuna eklenir ve sırasını bekler.

Kuyruk Diyagramı: Bir Sürecin Yolculuğu



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

Kuyruk Diyagramı: Bir Sürecin Yolculuğu



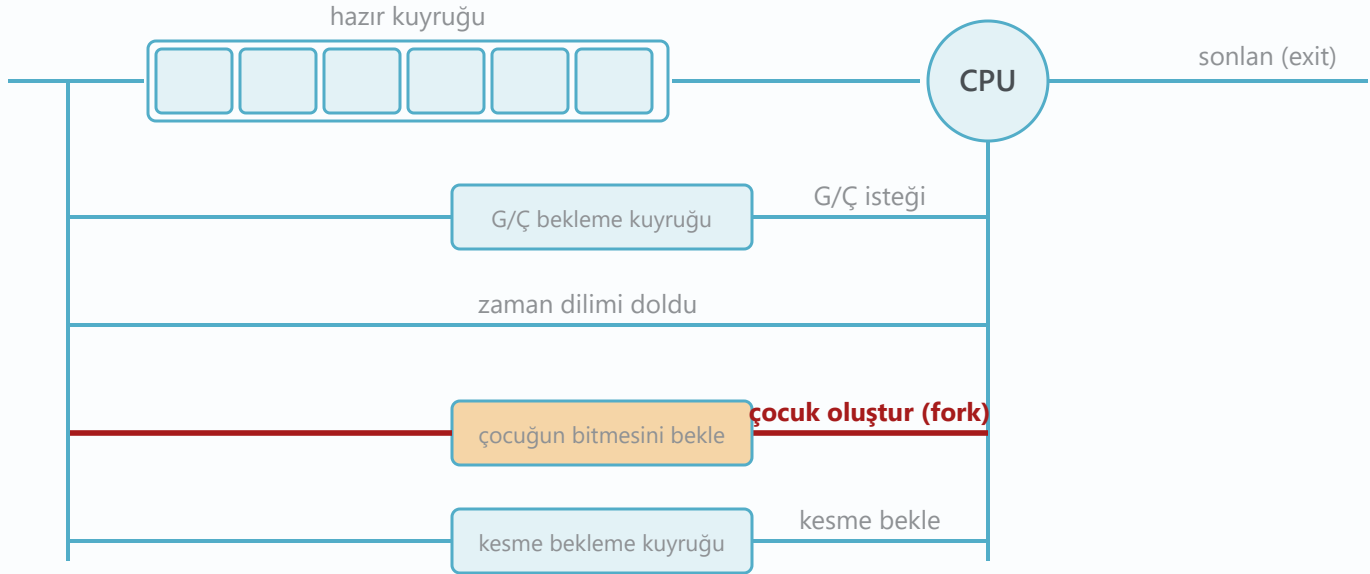
G/Ç isteği yaptı: G/Ç bekleme kuyruğına geçer; G/Ç bitince hazır kuyruğına döner.

Kuyruk Diyagramı: Bir Sürecin Yolculuğu



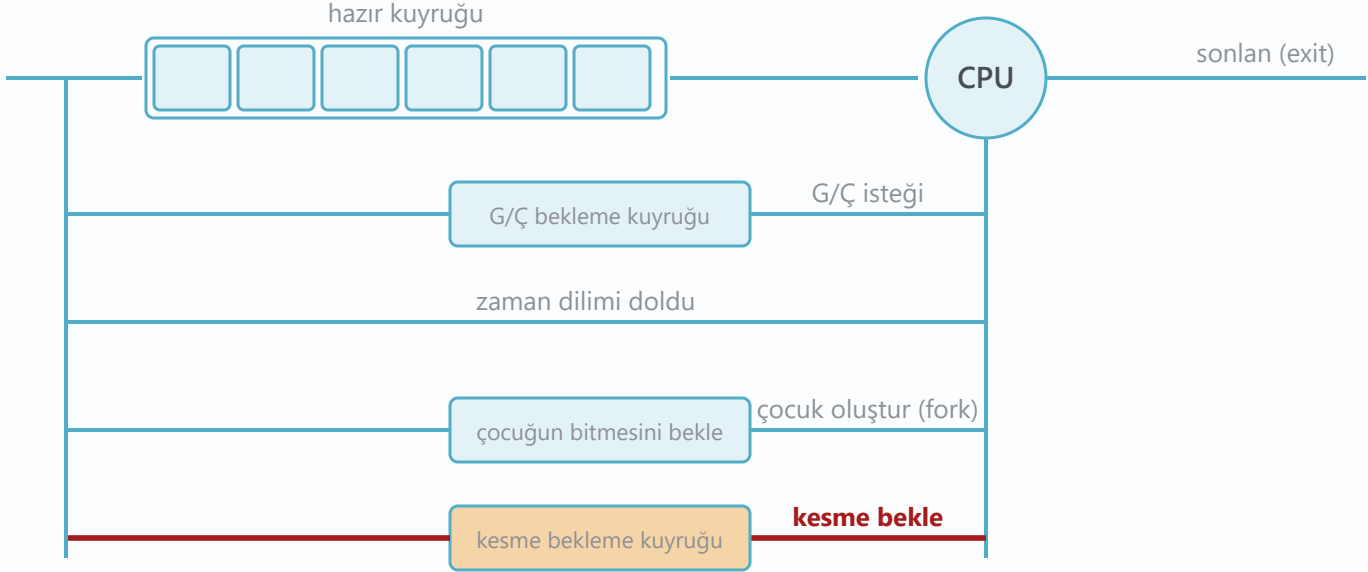
Zaman dilimi doldu: zamanlayıcı kesmesiyle CPU'dan alınır, doğrudan hazır kuyruğuna döner.

Kuyruk Diyagramı: Bir Sürecin Yolculuğu



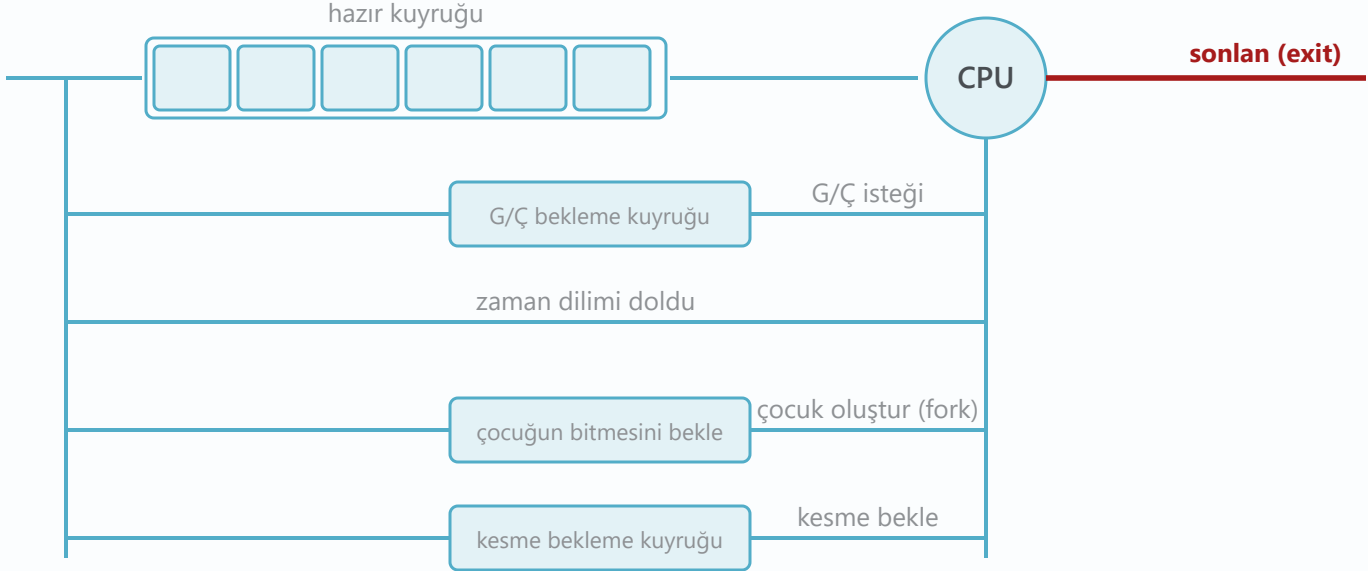
Çocuk oluşturup `wait()` çağırdı: çocuk bitene kadar bekler.

Kuyruk Diyagramı: Bir Sürecin Yolculuğu



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

Kuyruk Diyagramı: Bir Sürecin Yolculuğu



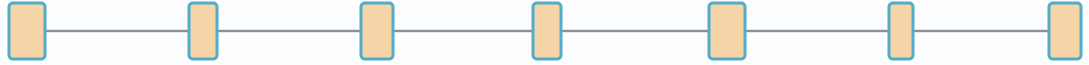
İşi bitti: `exit()` ile sistemden çıkar; PCB'si ebeveyn `wait` edince silinir.

CPU'ya Bağlı ve G/Ç'ye Bağlı Süreçler

(a) CPU'ya bağlı



(b) G/Ç'ye bağlı



 CPU patlaması (burst)

 G/Ç bekleme

zaman →

CPU'ya bağlı (CPU-bound)

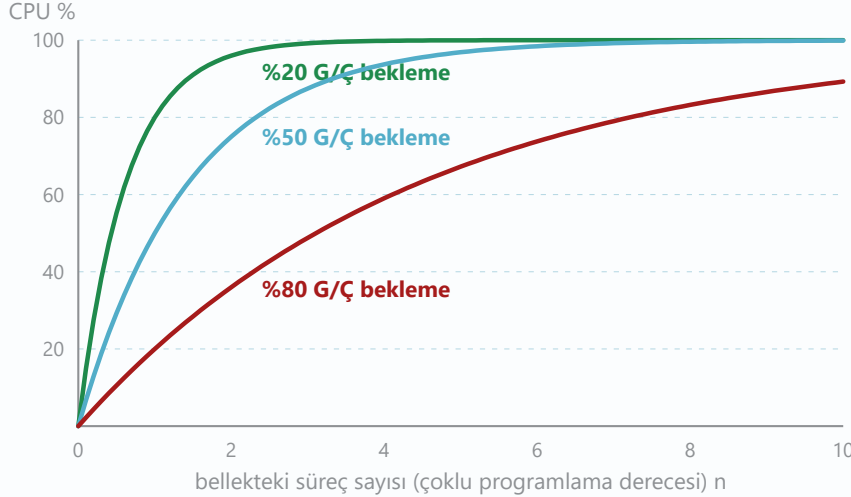
Uzun hesaplama patlamaları, seyrek G/Ç:
derleme, video kodlama, bilimsel hesap.

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 p^n 'dir:

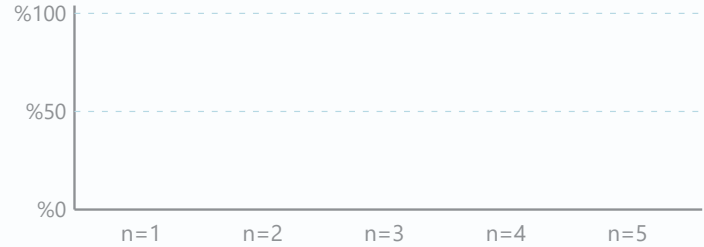
$$\text{CPU kullanımı} = 1 - p^n$$

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

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 }
```



boş olasılığı $p^n = \text{—}$

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

Adım 1 / 11

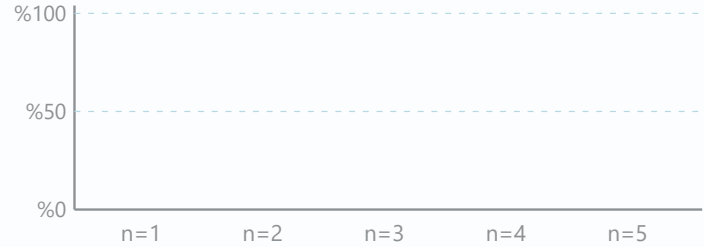
DEĞİŞKENLER

$p = 0.8$

$pn = 1.0$

Kod İzleme: CPU Kullanımı $1 - p^n$

```
1 double p = 0.8;      /* G/Ç oranı */
2 double pn = 1.0;
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 }
```



boş olasılığı $p^n = 0,8$

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

Adım 2 / 11

DEĞİŞKENLER

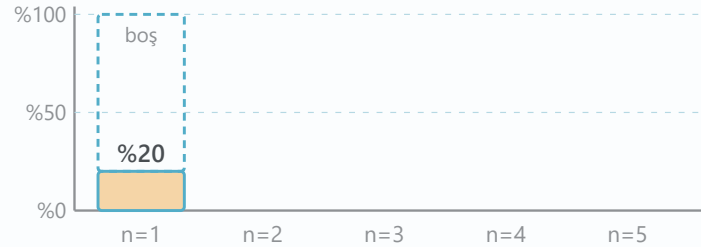
$p = 0.8$

$pn = 0.8$

$n = 1$

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 */
5     ▶ double kul = 1 - pn;
6     ▶ printf("n=%d %%.0f\n", n, kul * 100);
7 }
```



boş olasılığı $p^n = 0,8$

$kul = 1 - 0,8 = \text{\%20}$: tek programlı sistemde CPU zamanının beşte dördünde boş.

Adım 3 / 11

DEĞİŞKENLER

$p = 0.8$

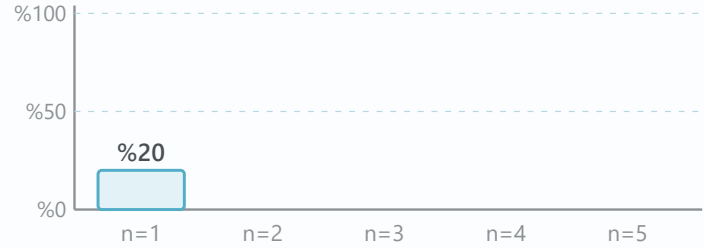
$pn = 0.8$

$n = 1$

$kul = 0.2$

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 }
```



boş olasılığı $p^n = 0,64$

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

Adım 4 / 11

DEĞİŞKENLER

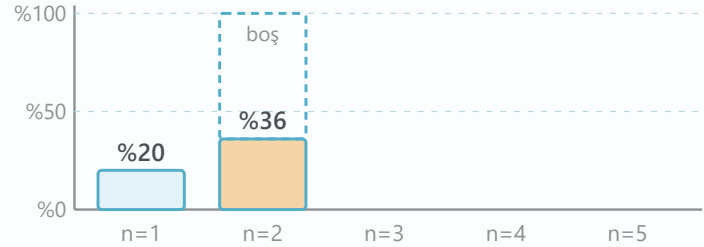
$p = 0.8$

$pn = 0.64$

$n = 2$

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 */
5     ▶ double kul = 1 - pn;
6     ▶ printf("n=%d %%.0f\n", n, kul * 100);
7 }
```



boş olasılığı $p^n = 0,64$

kul = %36.

Adım 5 / 11

DEĞİŞKENLER

p = 0.8

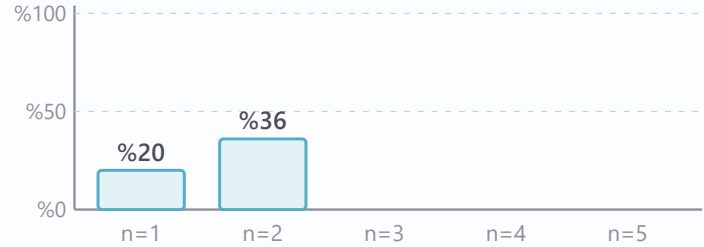
pn = 0.64

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 $0,8^3 = \mathbf{0,512}$.

Adım 6 / 11

DEĞİŞKENLER

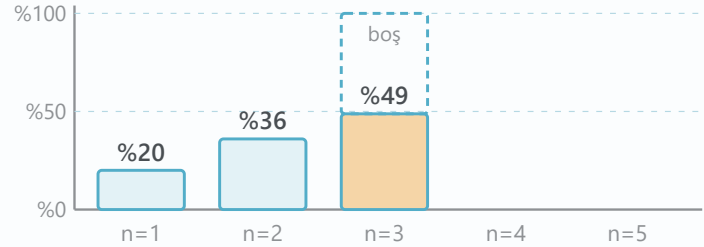
$p = 0.8$

$pn = 0.512$

$n = 3$

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 */
5     ▶ double kul = 1 - pn;
6     ▶ 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

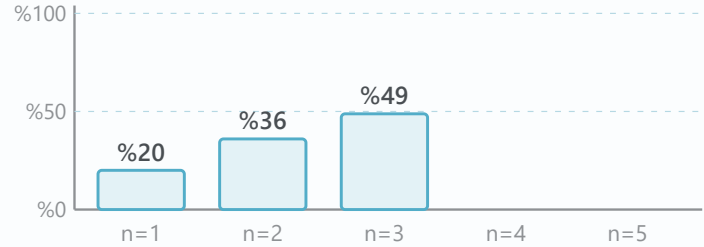
pn = 0.512

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;
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 }
```



boş olasılığı $p^n = 0,4096$

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

Adım 8 / 11

DEĞİŞKENLER

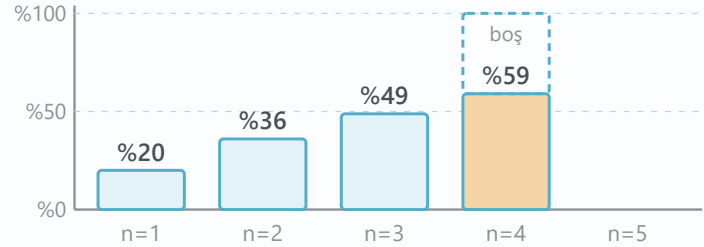
$p = 0.8$

$pn = 0.4096$

$n = 4$

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 */
5     ▶ double kul = 1 - pn;
6     ▶ 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

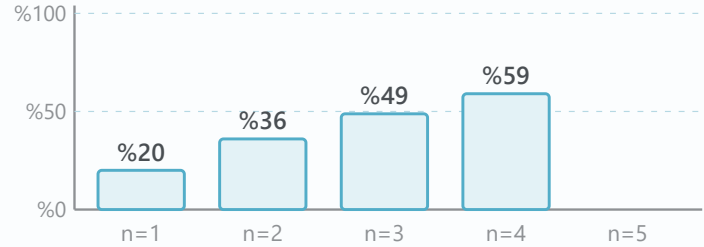
pn = 0.4096

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;
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 }
```



boş olasılığı $p^n = 0,3277$

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

Adım 10 / 11

DEĞİŞKENLER

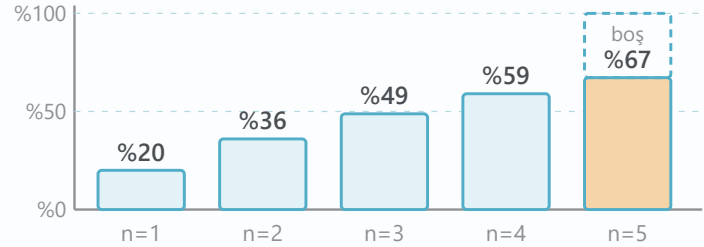
$p = 0.8$

$pn = 0.32768$

$n = 5$

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 */
5     ▶ double kul = 1 - pn;
6     ▶ 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)



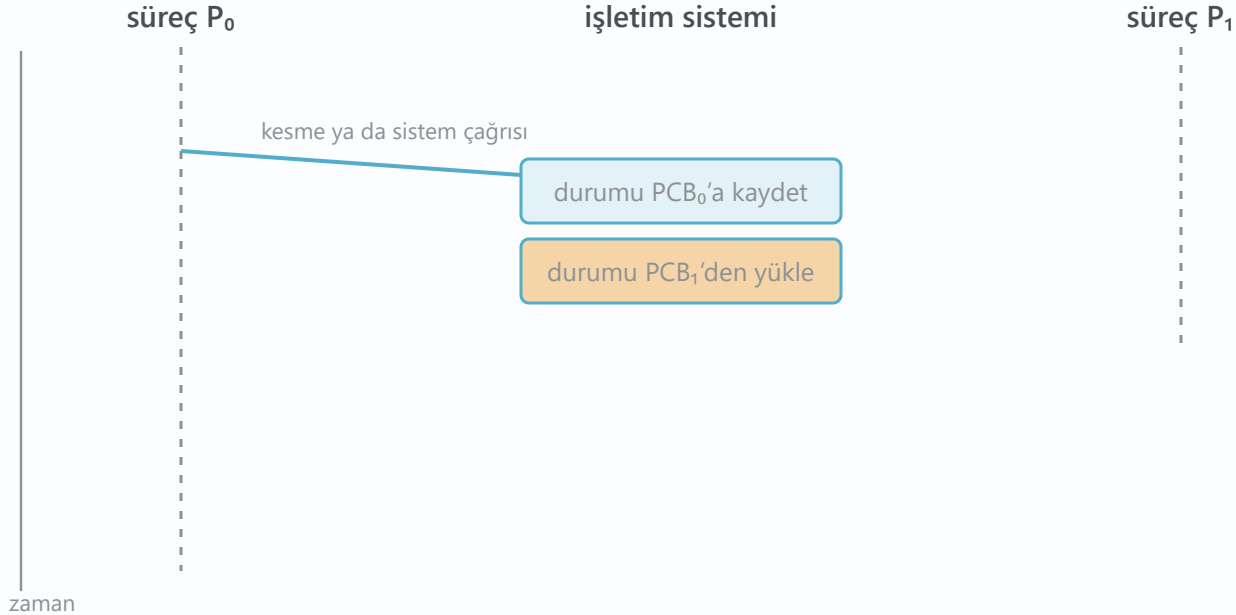
P₀ çalışıyor (kalın çizgi); P₁ hazır kuyruğunda bekliyor.

Bağlam Değiştirme (Context Switch)



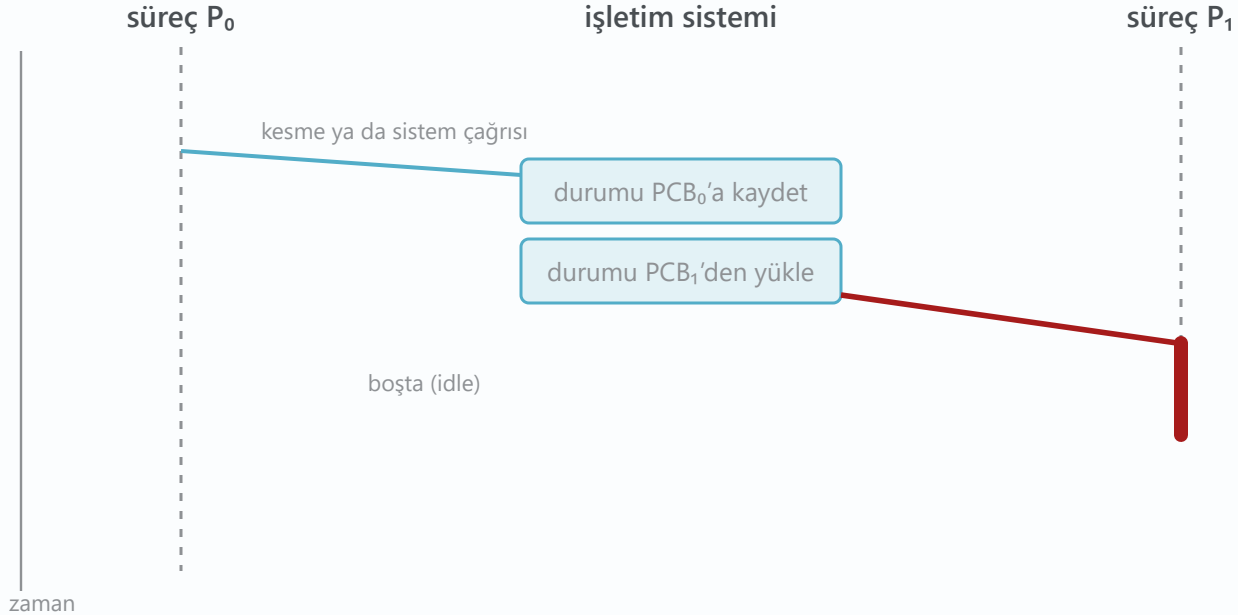
Kesme ya da sistem çağırısı: CPU çekirdek moduna geçer, P_0 'ın program sayacı ve yazmaçları **PCB₀**'a kaydedilir.

Bağlam Değiştirme (Context Switch)



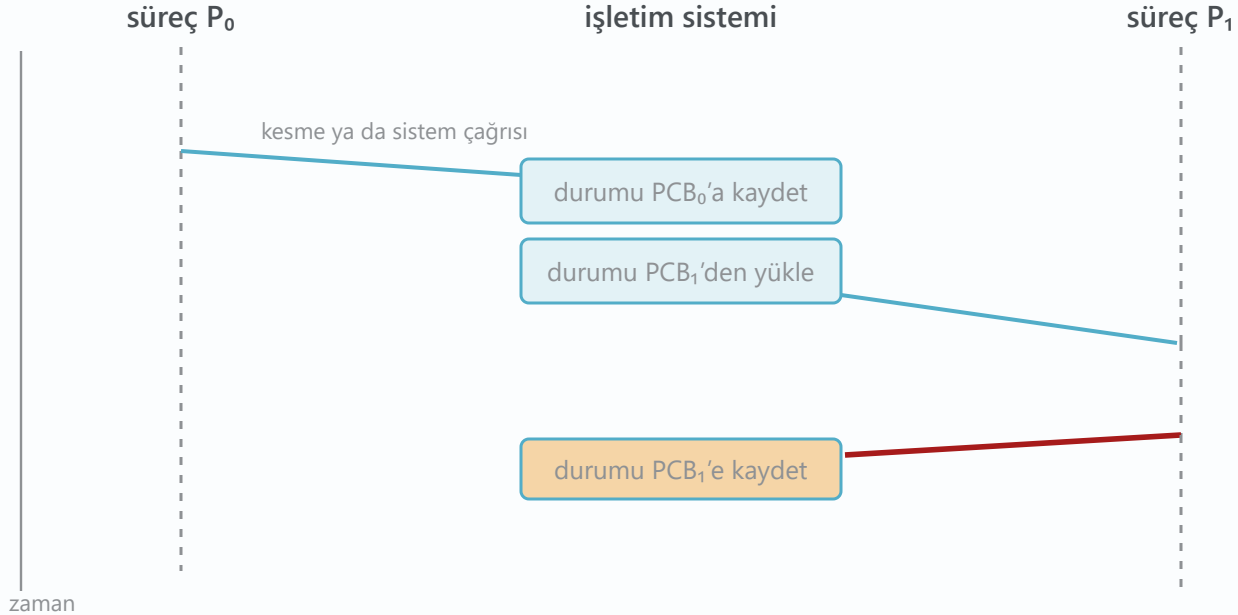
Çizelgeleyici P_1 'i seçer; P_1 'in bağlamı **PCB₁**'den yüklenir (adres alanı da değişir).

Bağlam Değiştirme (Context Switch)



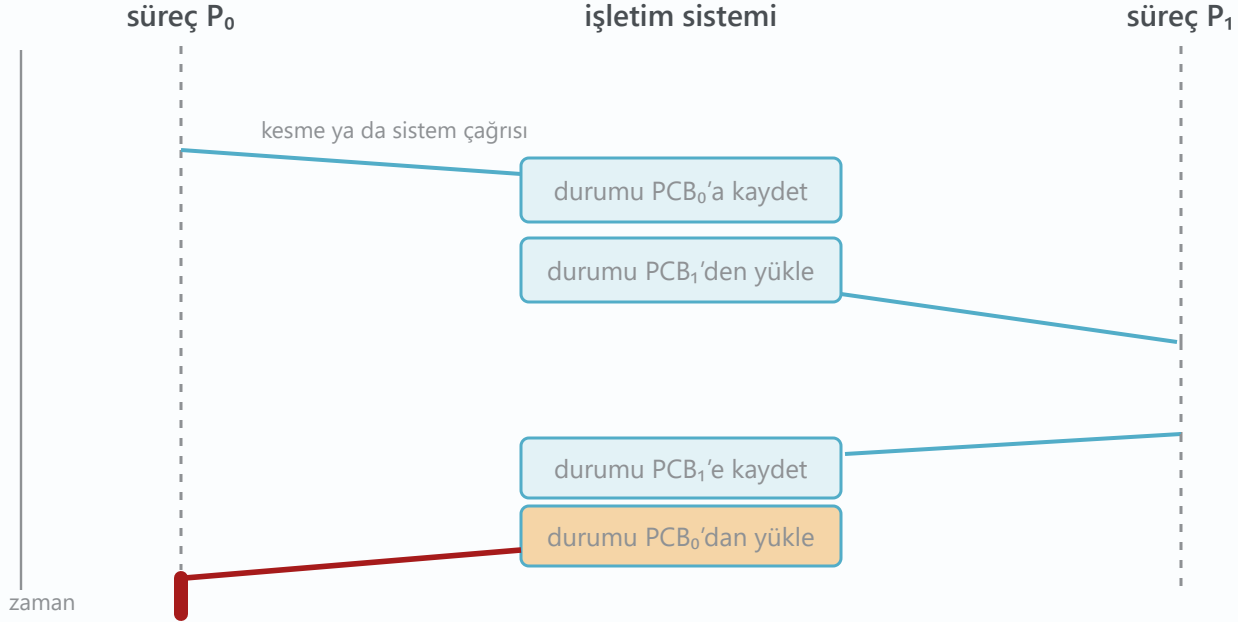
P_1 çalışıyor; P_0 boşta (hazır ya da bekliyor).

Bağlam Değiştirme (Context Switch)



Yeni kesme: P_1 'in durumu PCB₁'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		pc	—		pc	340	
sp	896		sp	—		sp	500	
r1	8		r1	—		r1	2	

P₀ çalışıyor: yazmaçlar **P₀'ın** değerlerini taşıyor. P₁'in son durumu PCB₁'de saklı. Zamanlayıcı kesmesi gelir; çekirdek P₁'e geçmeye karar verir.

Adım 1 / 11

DEĞİŞKENLER

pc = 104

sp = 896

r1 = 8

ÇAĞRI YİĞİNİ

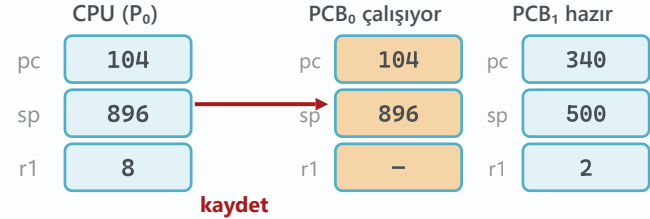
kesme işleyicisi [çekirdek]

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) {
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 ... */
12 /* kesme */ baglam_degistir(1, 0);

```



Kaydet: P₀'ın pc ve sp değerleri PCB₀'a yazılır. Bunlar kaybolursa P₀ kaldığı yere dönemez.

Adım 2 / 11

DEĞİŞKENLER

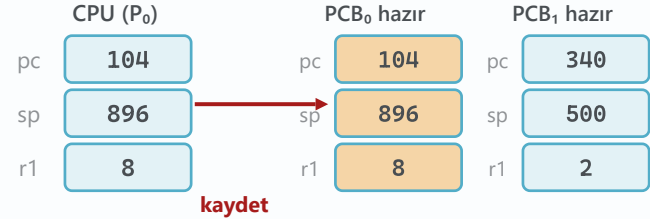
pc = 104 sp = 896 r1 = 8

ÇAĞRI YİĞİNİ

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);
11 /* ... P1 çalışır ... */
12 /* kesme */ baglam_degistir(1, 0);
```



Genel yazmaç da kaydedilir; P₀ artık **hazır** (H).

Adım 3 / 11

DEĞİŞKENLER

pc = 104

sp = 896

r1 = 8

ÇAĞRI YİĞİNİ

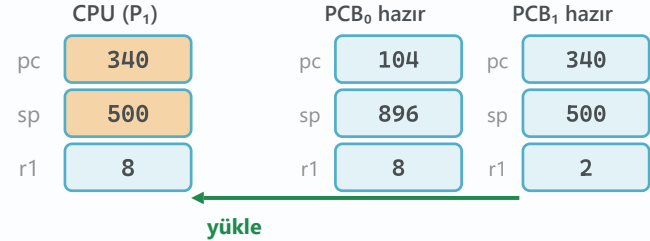
baglam_degistir(0, 1)

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);
11 /* ... P1 çalışır ... */
12 /* kesme */ baglam_degistir(1, 0);

```



Yükle: P₁'in pc ve sp değerleri CPU'ya. Bu andan sonra CPU P₁'in koduna ve yığına bakar.

Adım 4 / 11

DEĞİŞKENLER

pc = 340 sp = 500 r1 = 8

ÇAĞRI YIĞINI

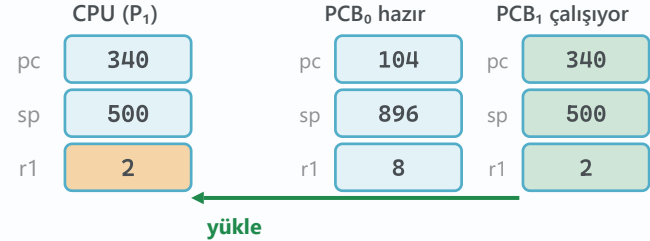
baglam_degistir(0, 1)

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 */
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 = 2 yüklenir, P₁ **çalışıyor** (C). Bu arada hiçbir kullanıcı işi yapılmadı: bağlam değiştirme saf **ek yük**tür.

Adım 5 / 11

DEĞİŞKENLER

pc = 340

sp = 500

r1 = 2

ÇAĞRI YİĞİNİ

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);
11 ► /* ... P1 çalışır ... */
12 /* kesme */ baglam_degistir(1, 0);
```

CPU (P ₁)			PCB ₀ hazır			PCB ₁ çalışıyor		
pc	352		pc	104		pc	340	
sp	492		sp	896		sp	500	
r1	5		r1	8		r1	2	

P₁ birkaç komut çalıştırdı: pc 340 → **352**, r1 = 5, sp = 492. PCB₁ bu sırada güncel değil; güncel değerler yalnızca CPU'da.

Adım 6 / 11

DEĞİŞKENLER

pc = 352

sp = 492

r1 = 5

ÇAĞRI YİĞİNİ

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		pc	104		pc	340	
sp	492		sp	896		sp	500	
r1	5		r1	8		r1	2	

Yeni kesme: çekirdek P₀'a dönmeye karar verir.

Adım 7 / 11

DEĞİŞKENLER

pc = 352

sp = 492

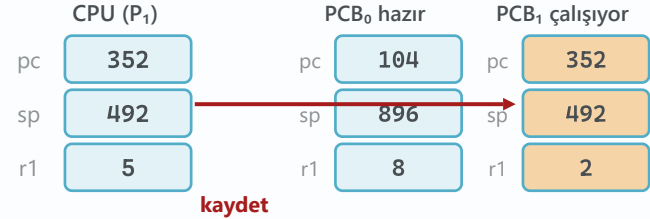
r1 = 5

ÇAĞRI YİĞİNİ

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) {
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 ... */
12 /* kesme */ baglam_degistir(1, 0);
```



P₁'in pc ve sp değerleri PCB₁'e kaydedilir.

Adım 8 / 11

DEĞİŞKENLER

pc = 352

sp = 492

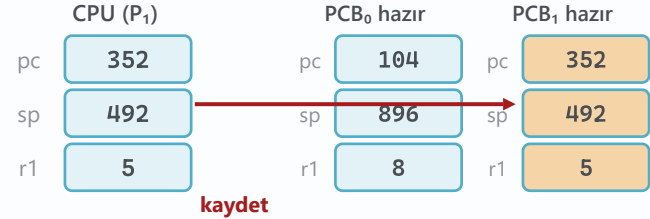
r1 = 5

ÇAĞRI YİĞİNİ

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 */
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; P₁ hazır.

Adım 9 / 11

DEĞİŞKENLER

pc = 352

sp = 492

r1 = 5

ÇAĞRI YİĞİNİ

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 */
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		pc	104	pc	352
sp	896		sp	896	sp	492
r1	5		r1	8	r1	5

PCB₀'daki değerler geri yüklenir: pc = 104, sp = 896.

Adım 10 / 11

DEĞİŞKENLER

pc = 104 sp = 896 r1 = 5

ÇAĞRI YİĞİNİ

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 */
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 çalışıyor			PCB ₁ hazır		
pc	104		pc	104		pc	352	
sp	896		sp	896		sp	492	
r1	8		r1	8		r1	5	

r1 = 8: P₀ tam olarak kesildiği durumdan, **pc = 104**'ten sürer; kesildiğini fark etmez. Gerçekte yazmaç sayısı onlardadır ve adres alanı da değişir.

Adım 11 / 11

DEĞİŞKENLER

pc = 104

sp = 896

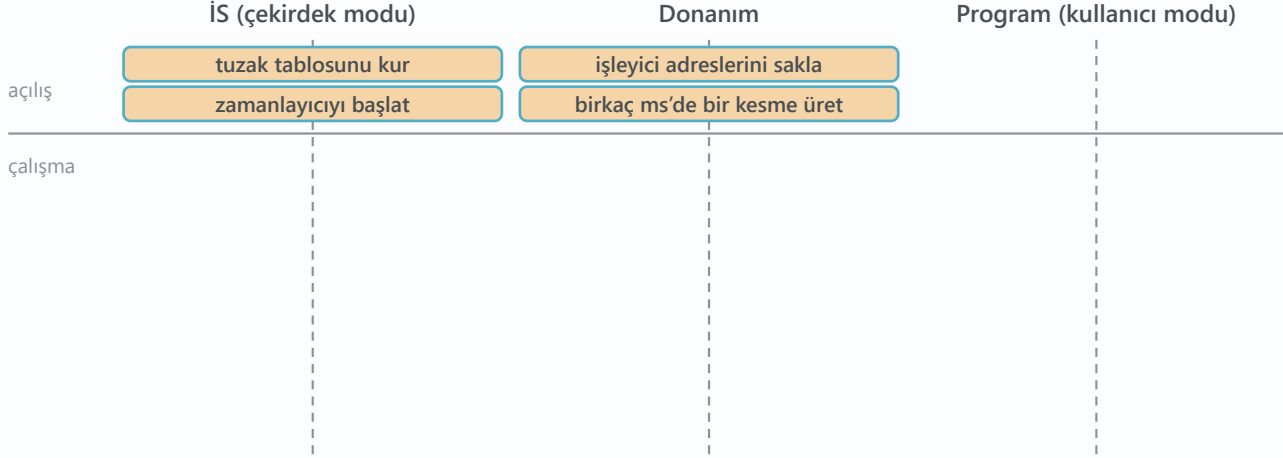
r1 = 8

ÇAĞRI YİĞİNİ

baglam_degistir(1, 0)

Sınırlı Doğrudan Yürütme

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



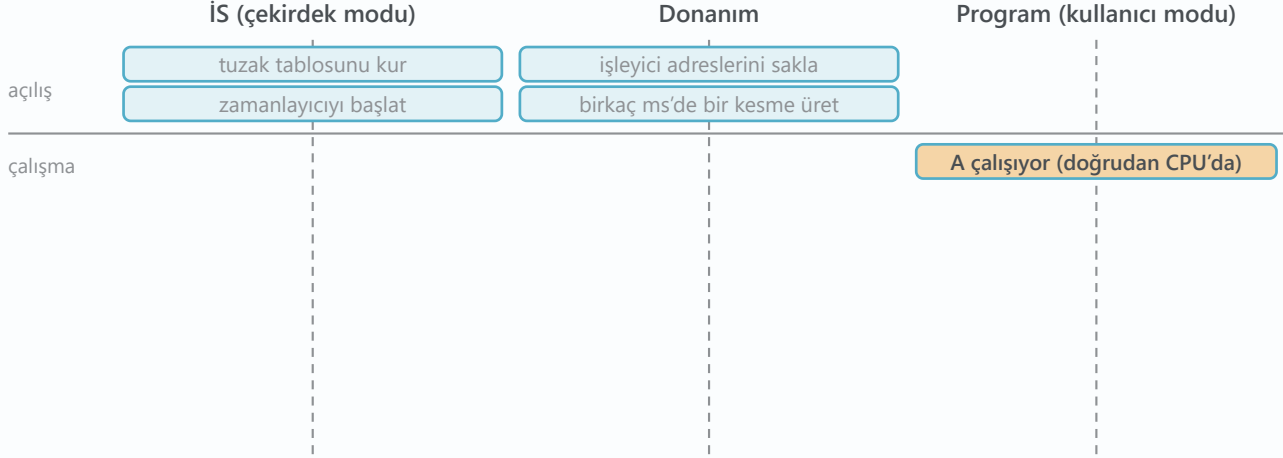
Açılıшта, çekirdek modunda: İS hangi olayda hangi kodun çalışacağını (tuzak/kesme tablosu) donanıma bildirir ve zamanlayıcıyı kurar. İkisi de **ayrıcalıklı** komuttur.

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.



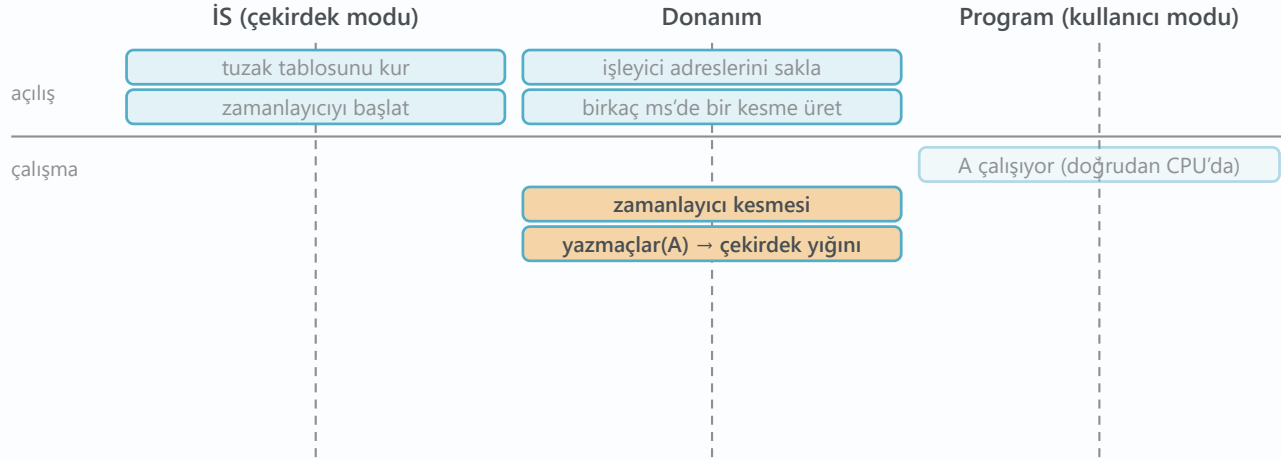
A kullanıcı modunda **doğrudan** CPU'da çalışır: arada yorumlayıcı yok, tam hız. Ayrıcalıklı bir iş için sistem çağrısıyla tuzağa düşmek zorundadır.

Adım 2 / 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.



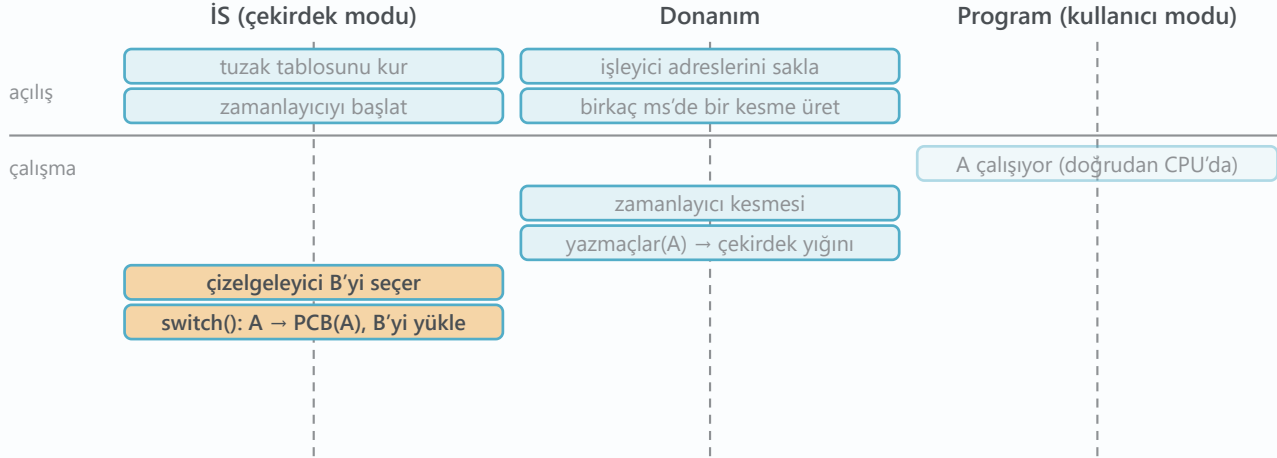
A sonsuz döngüde olsa bile kesme gelir. Donanım A'nın yazmaçlarını kaydeder, çekirdek moduna geçer ve İS'nin işleyicisine atlar: **denetim İS'de**.

Adım 3 / 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.



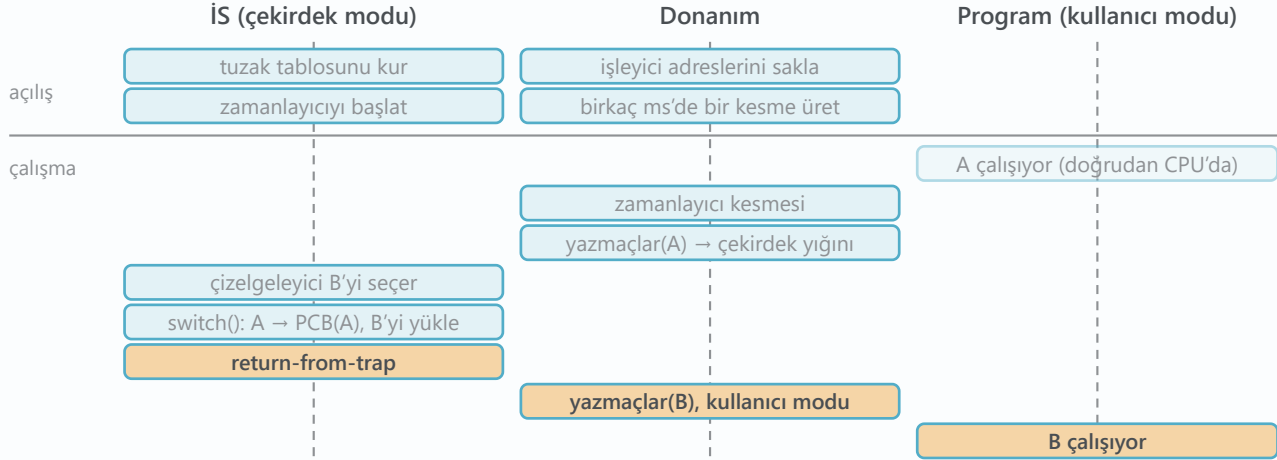
İS, A'nın dilimi dolduğu için B'ye geçmeye karar verir. `switch()` A'nın bağlamını PCB(A)'ya yazar, B'ninkini yükler ve B'nin çekirdek yığına geçer.

Adım 4 / 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.



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

Adım 5 / 5

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 yük**tür (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.

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.

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.

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 yığınınına) yazılır.

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 yığınının çalışır.

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.
- Uyaran 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.

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 (iş parçalamak), modülerlik, güvenlik ve yalıtım (Chrome).
- İşbirlikçi süreçler bir **IPC** (interprocess communication) mekanizmasına ihtiyaç duyar.

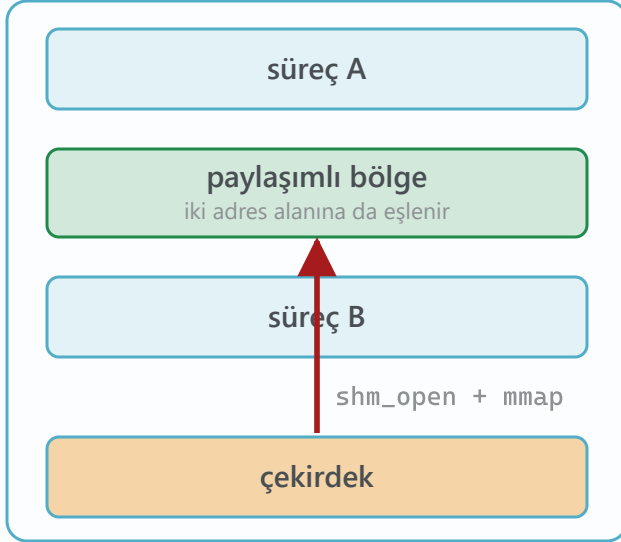
Üç temel soru

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.)

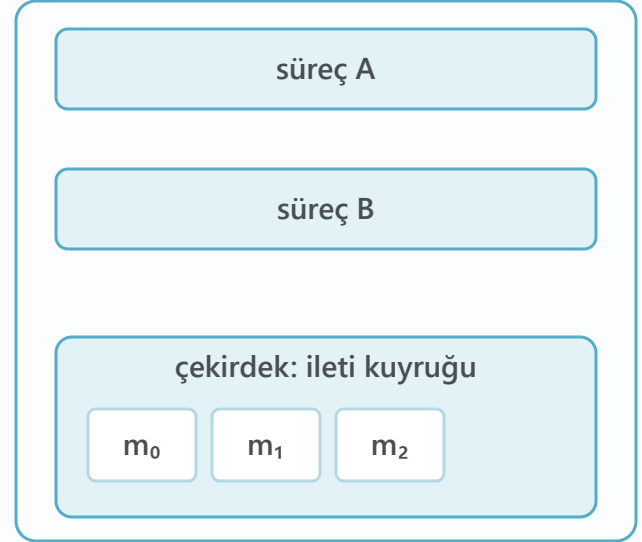
İki temel model: **paylaşımlı bellek** ve **ileti geçirme**. Son iki soru iş parçacıkları için de geçerlidir (Bölüm 6).

Paylaşımlı Bellek ve İleti Geçirme

(a) paylaşımlı bellek



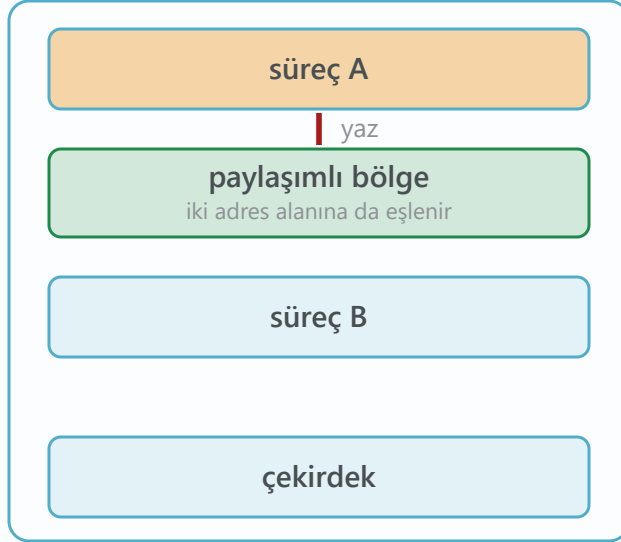
(b) ileti geçirme



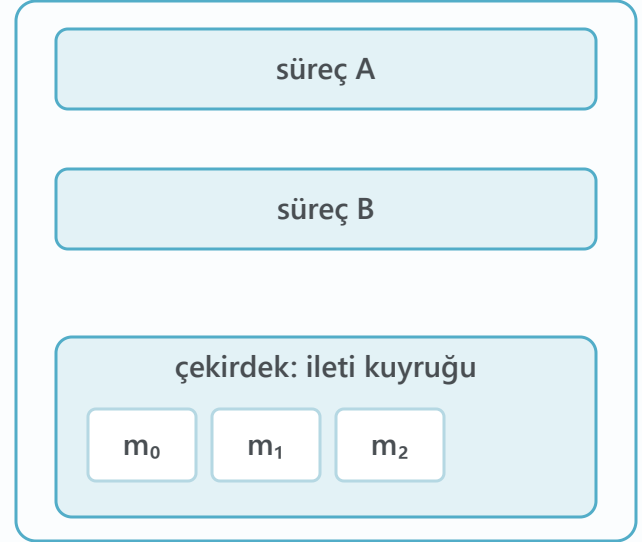
Paylaşımlı bellek: çekirdek bir bölge oluşturur ve iki sürecin adres alanına eşler (yalnızca kurulumda sistem çağırısı).

Paylaşımlı Bellek ve İleti Geçirme

(a) paylaşımlı bellek



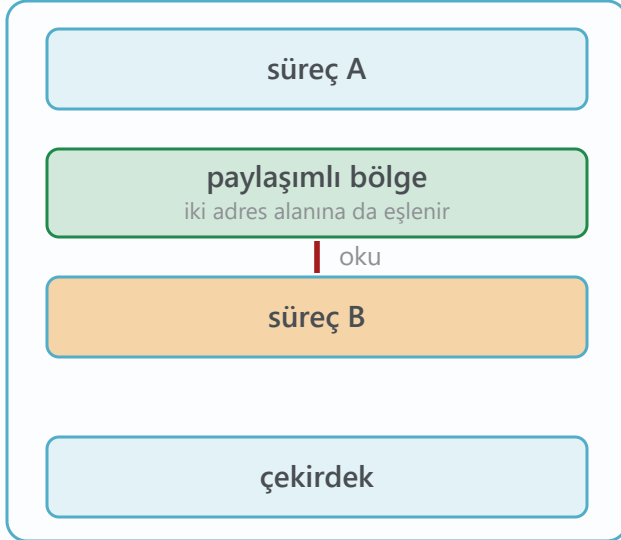
(b) ileti geçirme



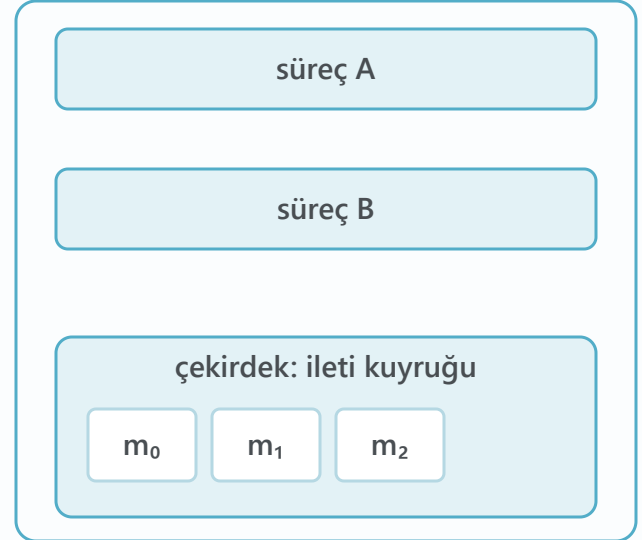
A bölgeye doğrudan yazar: çekirdek devrede **değildir**, bu yüzden çok hızlıdır.

Paylaşımlı Bellek ve İleti Geçirme

(a) paylaşımlı bellek



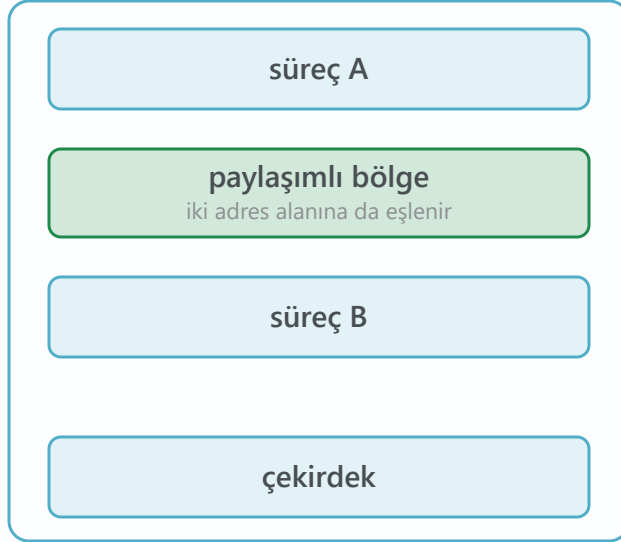
(b) ileti geçirme



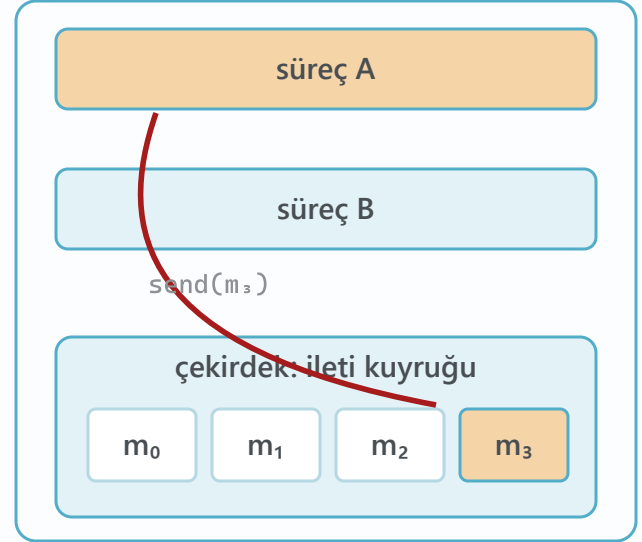
B aynı veriyi okur. Ancak senkronizasyonu (kim ne zaman yazar?) süreçler kendisi sağlamalıdır.

Paylaşımlı Bellek ve İleti Geçirme

(a) paylaşımlı bellek



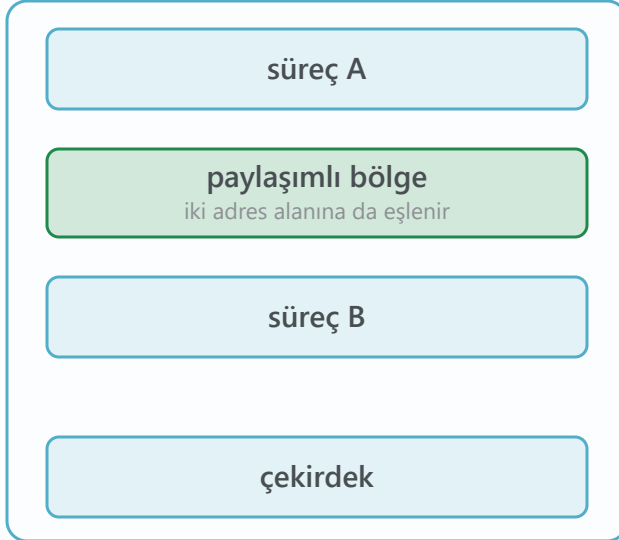
(b) ileti geçirme



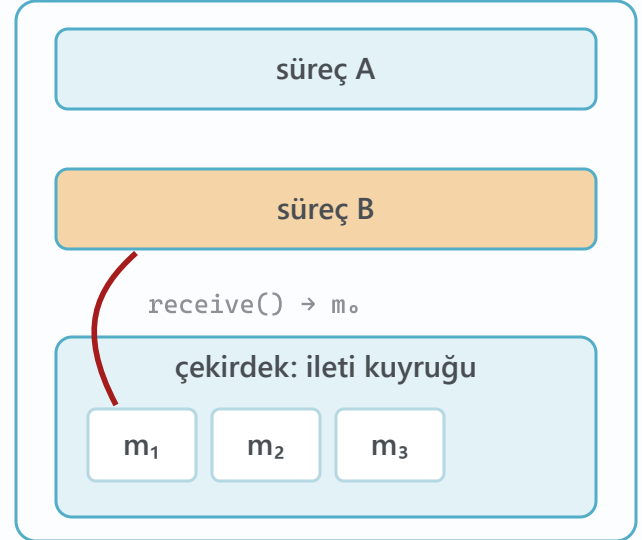
İleti geçirme: A `send( $m_3$ )` der; çekirdek iletiyi kendi kuyruğuna kopyalar.

Paylaşımlı Bellek ve İleti Geçirme

(a) paylaşımlı bellek



(b) ileti geçirme



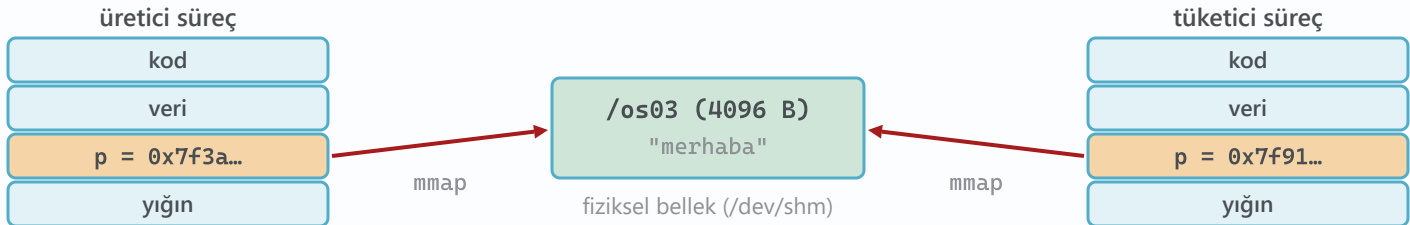
B `receive()` ile sıradaki iletiyi alır. Her iletide sistem çağırısı ve kopyalama vardır; ama senkronizasyon çekirdekçe sağlanır, ağ üzerinden de çalışır.

POSIX Paylaşımlı Bellek

```
/* 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 koddaki semafor gibi bir **senkronizasyon** gerekir.
- Eski System V API: `shmget` / `shmat`.



İleti Geçirme Tasarımı

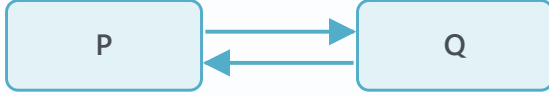
İki temel işlem: `send(mesaj)` ve `receive(mesaj)`. Önce bir **iletişim bağı** kurulmalıdır.

Boyut	Seçenekler
Fiziksel bağ	paylaşımlı bellek, donanım veri yolu, ağ
Adlandırma	doğrudan (sürecin adı) · dolaylı (posta kutusu / port)
Eş zamanlılık	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(Q, m)`



tek bağlantı · bir çift süreç

dolaylı: `send(A, m)`



çok süreç · çok bağlantı

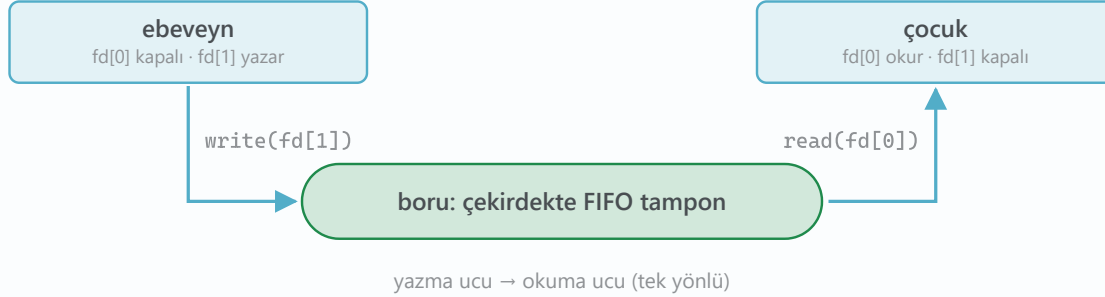
Doğrudan

- `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 paylaşılanlar haberleşir.
- Bir bağlantı çok sürece, bir çift çok bağlantıya sahip olabilir; tek ya da çift yönlü.

Borular (Pipes)



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	<code>mkfifo</code> ; 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 -
9             1);
10        buf[n] = '\0';
11        printf("çocuk aldı: %s\n", buf);
12        close(fd[0]);
13        return 0;
14    }
15    close(fd[0]); /* ebeveyn: yazar */
16    write(fd[1], "merhaba", 7);
17    close(fd[1]);
18    wait(NULL);
19    return 0;
20 }
```

DEĞİŞKENLER

fd[0] = 3

fd[1] = 4

ÇIKTI



ebeveyn 1234
çalışıyor

4 (w)

3 (r)

(boş)

`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);
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 -
9             1);
10        buf[n] = '\0';
11        printf("çocuk aldı: %s\n", buf);
12        close(fd[0]);
13        return 0;
14    }
15    close(fd[0]); /* ebeveyn: yazar */
16    write(fd[1], "merhaba", 7);
17    close(fd[1]);
18    wait(NULL);
19    return 0;
20 }
```

DEĞİŞKENLER

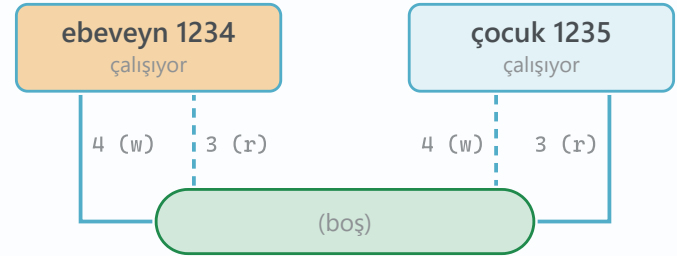
fd[0] = 3

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 */  
7         close(fd[1]);  
8         ssize_t n = read(fd[0], buf, sizeof buf -  
9             1);  
10        buf[n] = '\0';  
11        printf("çocuk aldı: %s\n", buf);  
12        close(fd[0]);  
13        return 0;  
14    }  
15    close(fd[0]); /* ebeveyn: yazar */  
16    write(fd[1], "merhaba", 7);  
17    close(fd[1]);  
18    wait(NULL);  
19    return 0;  
20 }
```

DEĞİŞKENLER

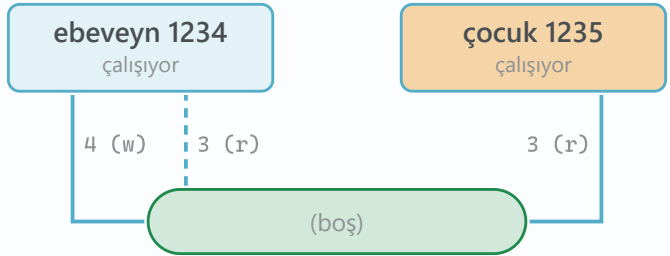
fd[0] = 3

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]);  
8         ▶ ssize_t n = read(fd[0], buf, sizeof buf -  
9             1);  
10        buf[n] = '\\0';  
11        printf("çocuk aldı: %s\\n", buf);  
12        close(fd[0]);  
13        return 0;  
14    }  
15    close(fd[0]); /* ebeveyn: yazar */  
16    write(fd[1], "merhaba", 7);  
17    close(fd[1]);  
18    wait(NULL);  
19    return 0;  
}
```

DEĞİŞKENLER

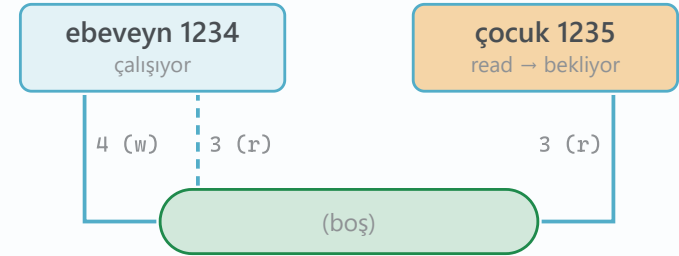
fd[0] = 3

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 -  
9             1);  
10        buf[n] = '\0';  
11        printf("çocuk aldı: %s\n", buf);  
12        close(fd[0]);  
13        return 0;  
14    }  
15    close(fd[0]); /* ebeveyn: yazar */  
16    write(fd[1], "merhaba", 7);  
17    close(fd[1]);  
18    wait(NULL);  
19    return 0;  
}
```

DEĞİŞKENLER

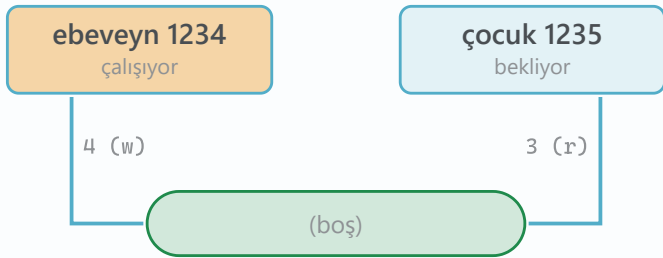
fd[0] = 3

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 -
9             1);
10        buf[n] = '\0';
11        printf("çocuk aldı: %s\n", buf);
12        close(fd[0]);
13        return 0;
14    }
15    close(fd[0]); /* ebeveyn: yazar */
16    write(fd[1], "merhaba", 7);
17    close(fd[1]);
18    wait(NULL);
19    return 0;
20 }
```

DEĞİŞKENLER

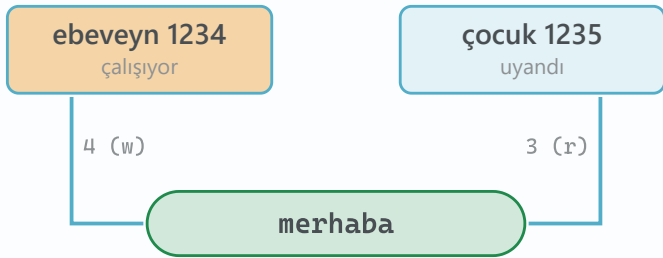
fd[0] = 3

fd[1] = 4

ebeveyn.pid = 1235

çocuk.pid = 0

ÇIKTI



Ebeveyn 7 bayt yazar; $\leq$ 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 -  
9             1);  
10        buf[n] = '\0';  
11        printf("çocuk aldı: %s\n", buf);  
12        close(fd[0]);  
13        return 0;  
14    }  
15    close(fd[0]); /* ebeveyn: yazar */  
16    write(fd[1], "merhaba", 7);  
17    close(fd[1]);  
18    wait(NULL);  
19 }
```

DEĞİŞKENLER

fd[0] = 3

fd[1] = 4

ebeveyn.pid = 1235

çocuk.pid = 0

ÇIKTI



ebeveyn 1234

wait → bekliyor

çocuk 1235

hazır

3 (r)

merhaba

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]);  
8         ssize_t n = read(fd[0], buf, sizeof buf -  
9             1);  
10        buf[n] = '\0';  
11        printf("çocuk aldı: %s\n", buf);  
12        close(fd[0]);  
13        return 0;  
14    }  
15    close(fd[0]); /* ebeveyn: yazar */  
16    write(fd[1], "merhaba", 7);  
17    close(fd[1]);  
18    wait(NULL);  
19    return 0;  
20 }
```

DEĞİŞKENLER

n = 7**buf = "merhaba"**

çocuk.pid = 0

ÇIKTI

**ebeveyn 1234**
bekliyor**çocuk 1235**
çalışıyor

3 (r)

(boş)

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 -
9             1);
10        ▶ buf[n] = '\0';
11        ▶ printf("çocuk aldı: %s\n", buf);
12        close(fd[0]);
13        return 0;
14    }
15    close(fd[0]); /* ebeveyn: yazar */
16    write(fd[1], "merhaba", 7);
17    close(fd[1]);
18    wait(NULL);
19    return 0;
20 }
```

DEĞİŞKENLER

n = 7

buf = "merhaba\0"

çocuk.pid = 0

ÇIKTI

çocuk aldı: merhaba

ebeveyn 1234
bekliyor

çocuk 1235
çalışıyor

3 (r)

(boş)

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 -
9             1);
10        buf[n] = '\0';
11        printf("çocuk aldı: %s\n", buf);
12        close(fd[0]);
13        return 0;
14    }
15    close(fd[0]); /* ebeveyn: yazar */
16    write(fd[1], "merhaba", 7);
17    close(fd[1]);
18    ▶ wait(NULL);
19    ▶ return 0;
20 }
```

DEĞİŞKENLER

ebeveyn.pid = 1235

ÇIKTI

çocuk aldı: merhaba

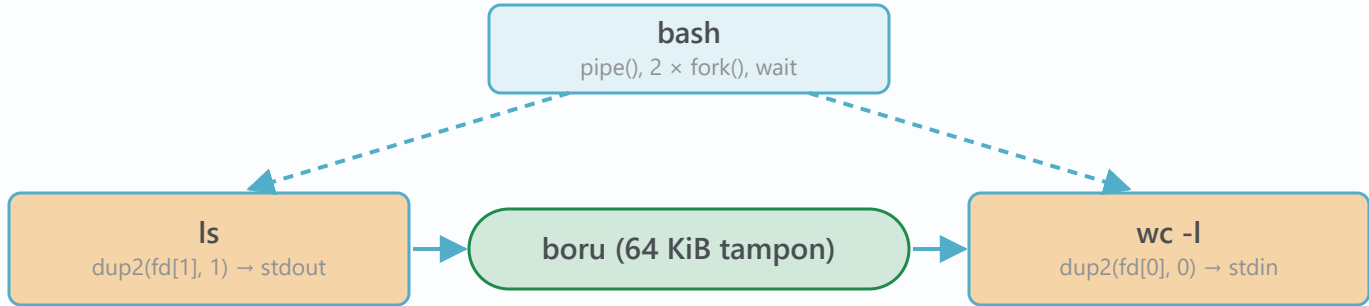
ebeveyn 1234
çalışıyor

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

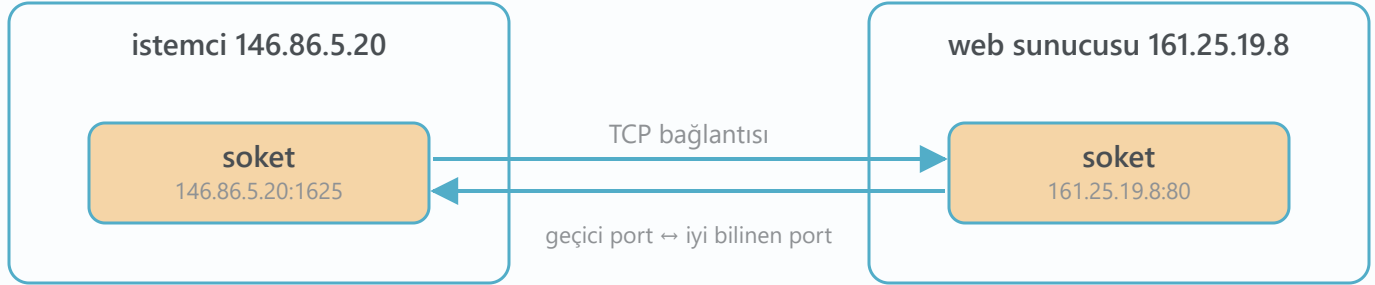


her iki süreç de kullanmadığı uçları kapatmalı; aksi hâlde wc EOF göremez

```
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.

Yarış Durumu: Yazıcı Biriktirme Dizini

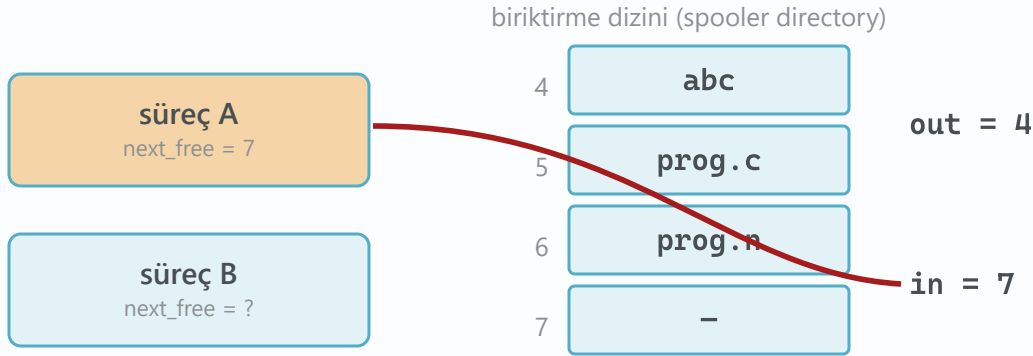


Yazdırılacak dosyalar sırayla yuvalara yazılır. **out** sıradaki yazdırılacak, **in** ilk boş yuva.

Adım 1 / 6

Yarış durumu (race condition): iki ya da daha çok süreç paylaşılan veriye eriştiğinde nihai sonucun yürütme sırasına bağlı olması.

Yarış Durumu: Yazıcı Biriktirme Dizini

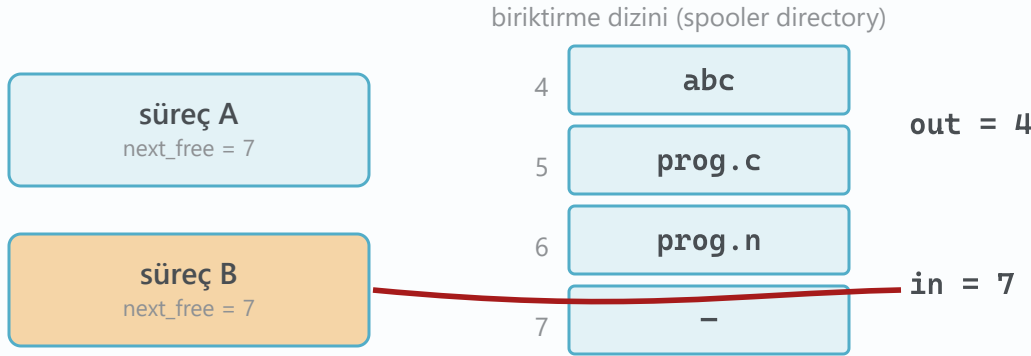


A `in` 'i okur (7) ve yerel `next_free` 'ye koyar; tam o anda zaman dilimi biter.

Adım 2 / 6

Yarış durumu (race condition): iki ya da daha çok süreç paylaşılan veriye eriştiğinde nihai sonucun yürütme sırasına bağlı olması.

Yarış Durumu: Yazıcı Biriktirme Dizini

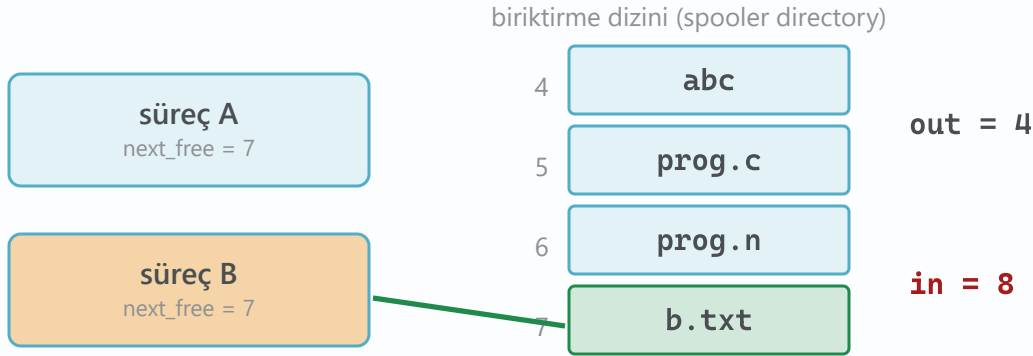


B de **in** 'i okur: yine 7!

Adım 3 / 6

Yarış durumu (race condition): iki ya da daha çok süreç paylaşılan veriye eriştiğinde nihai sonucun yürütme sırasına bağlı olması.

Yarış Durumu: Yazıcı Biriktirme Dizini

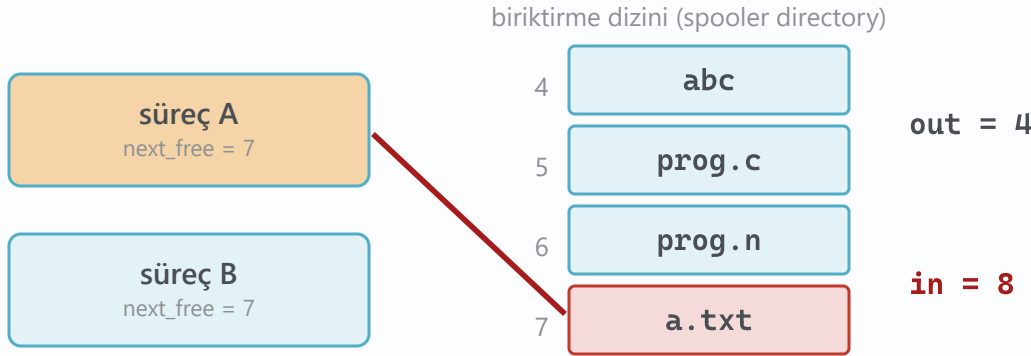


B dosya adını 7. yuvaya yazar, **in = 8** yapar.

Adım 4 / 6

Yarış durumu (race condition): iki ya da daha çok süreç paylaşılan veriye eriştiğinde nihai sonucun yürütme sırasına bağlı olması.

Yarış Durumu: Yazıcı Biriktirme Dizini

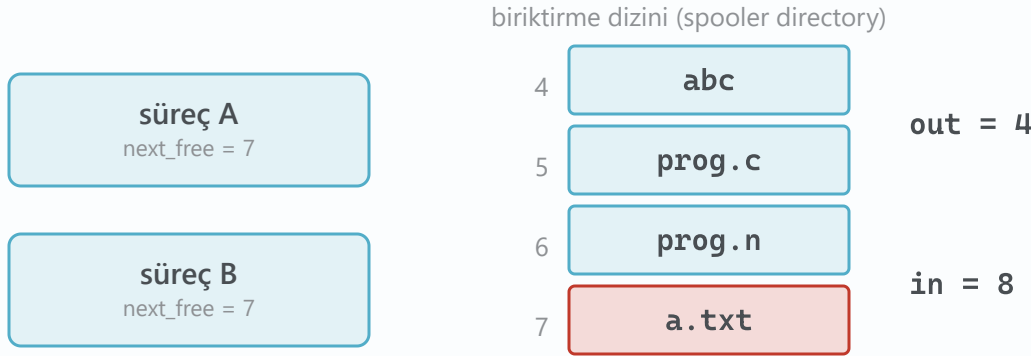


A kaldığı yerden devam eder: `next_free = 7` olduğundan 7. yuvaya yazar, B'nin girdisini **ezer**; `in = 8`.

Adım 5 / 6

Yarış durumu (race condition): iki ya da daha çok süreç paylaşılan veriye eriştiğinde nihai sonucun yürütme sırasına bağlı olması.

Yarış Durumu: Yazıcı Biriktirme Dizini



B'nin dosyası hiç yazdırılmaz!

Dizin tutarlı görünür ama B'nin çıktısı kaybolmuştur. Sonuç kimin ne zaman çalıştığına bağlıdır: **yarış durumu**.

Adım 6 / 6

Yarış durumu (race condition): iki ya da daha çok süreç paylaşılan veriye eriştiğinde nihai sonucun yürütme sırasına bağlı olması.

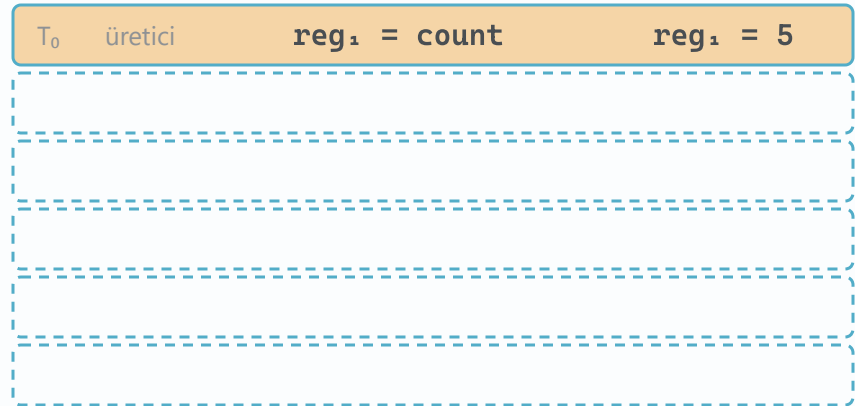
Tekrarlanamaz Yürütme (Non-reproducible)

```
/* Süreç 1 */      /* Süreç 2 */  
repeat              repeat  
  n = n + 1;         print(n);  
                    n = 0;
```

n = 5 iken olası sıralar:

- `n=n+1; print; n=0` → 6 yazar
- `print; n=0; n=n+1` → 5 yazar, n=1
- `print; n=n+1; n=0` → 5 yazar, artış kaybolur

Tek bir C satırı bile (`count++`)
makinede üç komuttur:



Başta `count = 5` . Üretici `count++` için önce
değeri yazmaca yükler.

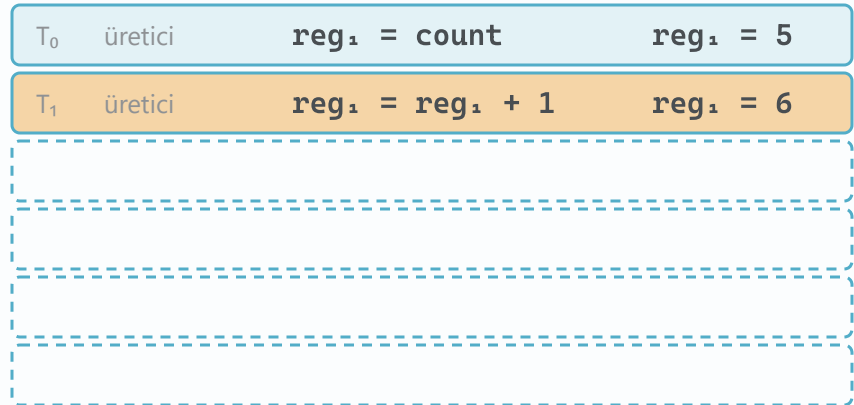
Tekrarlanamaz Yürütme (Non-reproducible)

```
/* Süreç 1 */      /* Süreç 2 */  
repeat             repeat  
  n = n + 1;        print(n);  
                    n = 0;
```

n = 5 iken olası sıralar:

- `n=n+1; print; n=0` → 6 yazar
- `print; n=0; n=n+1` → 5 yazar, n=1
- `print; n=n+1; n=0` → 5 yazar, artış kaybolur

Tek bir C satırı bile (`count++`)
makinede üç komuttur:



Üretici yazmacı artırır: 6. Henüz belleğe yazmadı ve kesildi.

Tekrarlanamaz Yürütme (Non-reproducible)

```
/* Süreç 1 */      /* Süreç 2 */  
repeat              repeat  
  n = n + 1;         print(n);  
                     n = 0;
```

n = 5 iken olası sıralar:

- `n=n+1; print; n=0` → 6 yazar
- `print; n=0; n=n+1` → 5 yazar, n=1
- `print; n=n+1; n=0` → 5 yazar, artış kaybolur

Tek bir C satırı bile (`count++`)
makinede üç komuttur:

T ₀	üretici	<code>reg₁ = count</code>	<code>reg₁ = 5</code>
T ₁	üretici	<code>reg₁ = reg₁ + 1</code>	<code>reg₁ = 6</code>
T ₂	tüketici	<code>reg₂ = count</code>	<code>reg₂ = 5</code>

Tüketici `count--` : bellekteki değer hâlâ 5.

Tekrarlanamaz Yürütme (Non-reproducible)

```
/* Süreç 1 */      /* Süreç 2 */  
repeat              repeat  
  n = n + 1;         print(n);  
                    n = 0;
```

n = 5 iken olası sıralar:

- `n=n+1; print; n=0` → 6 yazar
- `print; n=0; n=n+1` → 5 yazar, n=1
- `print; n=n+1; n=0` → 5 yazar, artış kaybolur

Tek bir C satırı bile (`count++`)
makinede üç komuttur:

T ₀	üretici	reg₁ = count	reg₁ = 5
T ₁	üretici	reg₁ = reg₁ + 1	reg₁ = 6
T ₂	tüketici	reg₂ = count	reg₂ = 5
T ₃	tüketici	reg₂ = reg₂ - 1	reg₂ = 4

Tüketici yazmacı azaltır: 4.

Tekrarlanamaz Yürütme (Non-reproducible)

```
/* Süreç 1 */      /* Süreç 2 */  
repeat              repeat  
  n = n + 1;         print(n);  
                    n = 0;
```

n = 5 iken olası sıralar:

- `n=n+1; print; n=0` → 6 yazar
- `print; n=0; n=n+1` → 5 yazar, n=1
- `print; n=n+1; n=0` → 5 yazar, artış kaybolur

Tek bir C satırı bile (`count++`)
makinede üç komuttur:

T ₀	üretici	<code>reg₁ = count</code>	<code>reg₁ = 5</code>
T ₁	üretici	<code>reg₁ = reg₁ + 1</code>	<code>reg₁ = 6</code>
T ₂	tüketici	<code>reg₂ = count</code>	<code>reg₂ = 5</code>
T ₃	tüketici	<code>reg₂ = reg₂ - 1</code>	<code>reg₂ = 4</code>
T ₄	üretici	<code>count = reg₁</code>	<code>count = 6</code>

Üretici belleğe yazar: `count = 6` .

Adım 5 / 6

Tekrarlanamaz Yürütme (Non-reproducible)

```
/* Süreç 1 */      /* Süreç 2 */  
repeat              repeat  
  n = n + 1;         print(n);  
                     n = 0;
```

n = 5 iken olası sıralar:

- `n=n+1; print; n=0` → 6 yazar
- `print; n=0; n=n+1` → 5 yazar, n=1
- `print; n=n+1; n=0` → 5 yazar, artış kaybolur

Tek bir C satırı bile (`count++`)
makinede üç komuttur:

T ₀	üretici	<code>reg₁ = count</code>	<code>reg₁ = 5</code>
T ₁	üretici	<code>reg₁ = reg₁ + 1</code>	<code>reg₁ = 6</code>
T ₂	tüketici	<code>reg₂ = count</code>	<code>reg₂ = 5</code>
T ₃	tüketici	<code>reg₂ = reg₂ - 1</code>	<code>reg₂ = 4</code>
T ₄	üretici	<code>count = reg₁</code>	<code>count = 6</code>
T ₅	tüketici	<code>count = reg₂</code>	<code>count = 4</code>

Tüketici üzerine yazar: `count = 4` . Doğrusu 5'ti! Sıra değişseydi 6 çıkabilirdi.

Kritik Bölge ve Karşılıklı Dışlama

- **Kritik bölge:** programın paylaşılan veriye eriştiği kod parçası.
- **Karşılıklı dışlama:** bir süreç kritik bölgedeyken diğerlerinin girmesini engellemek.

İyi bir çözümün dört koşulu (Tanenbaum):

1. İki süreç aynı anda kritik bölgede olamaz.
2. CPU hızı ya da sayısı hakkında varsayım yapılmaz.
3. Kritik bölge dışındaki süreç başkasını engelleyemez.
4. Hiçbir süreç girmek için sonsuza dek beklemez.



Peterson Çözümü

```
▶ bool flag[2] = { false, false };
▶ int turn;

3
4 void enter(int i) {
5     int j = 1 - i;
6     flag[i] = true;    /* girmek istiyorum */
7     turn = j;          /* ama önce sen */
8     while (flag[j] && turn == j)
9         ;              /* meşgul bekle */
10 }
11 void leave(int i) {
12     flag[i] = false;
13 }
```

DEĞİŞKENLER

flag[0] = **false**flag[1] = **false**

turn = ?

P0

dışarıda

P1

dışarıda

kritik bölge: boş

Başlangıç: kimse girmek istemiyor.

Adım 1 / 9

Uyarı: modern CPU ve derleyiciler yükleme/saklama sırasını değiştirebilir; Peterson ancak **_Atomic** değişkenler (sıralı tutarlılık) ya da bellek bariyerleriyle doğru çalışır. Pratikte donanım atomik komutları ve kilitler kullanılır.

Peterson Çözümü

```
1 bool flag[2] = { false, false };
2 int turn;
3
4 void enter(int i) {
5     int j = 1 - i;
6     ▶ flag[i] = true; /* girmek istiyorum */
7     turn = j;        /* ama önce sen */
8     while (flag[j] && turn == j)
9         ;            /* meşgul bekle */
10 }
11 void leave(int i) {
12     flag[i] = false;
13 }
```

DEĞİŞKENLER

flag[0] = true

flag[1] = false

turn = ?

P0

flag[0]=true

P1

dışarıda

kritik bölge: boş

P0 girmek istediğini bildirir.

Adım 2 / 9

Uyarı: modern CPU ve derleyiciler yükleme/saklama sırasını değiştirebilir; Peterson ancak `_Atomic` değişkenler (sıralı tutarlılık) ya da bellek bariyerleriyle doğru çalışır. Pratikte donanım atomik komutları ve kilitler kullanılır.

Peterson Çözümü

```
1 bool flag[2] = { false, false };
2 int turn;
3
4 void enter(int i) {
5     int j = 1 - i;
6     ▶ flag[i] = true; /* girmek istiyorum */
7     turn = j;        /* ama önce sen */
8     while (flag[j] && turn == j)
9         ;            /* meşgul bekle */
10 }
11 void leave(int i) {
12     flag[i] = false;
13 }
```

DEĞİŞKENLER

flag[0] = true

flag[1] = true

turn = ?

P0

kesildi

P1

flag[1]=true

kritik bölge: boş

Kesme! P1 de hemen ardından girmek istediğini bildirir.

Adım 3 / 9

Uyarı: modern CPU ve derleyiciler yükleme/saklama sırasını değiştirebilir; Peterson ancak `_Atomic` değişkenler (sıralı tutarlılık) ya da bellek bariyerleriyle doğru çalışır. Pratikte donanım atomik komutları ve kilitler kullanılır.

Peterson Çözümü

```
1 bool flag[2] = { false, false };
2 int turn;
3
4 void enter(int i) {
5     int j = 1 - i;
6     flag[i] = true; /* girmek istiyorum */
7     turn = j;      /* ama önce sen */
8     while (flag[j] && turn == j)
9         ;          /* meşgul bekle */
10 }
11 void leave(int i) {
12     flag[i] = false;
13 }
```

DEĞİŞKENLER

flag[0] = true

flag[1] = true

turn = 1

P0

turn=1

P1

kesildi

kritik bölge: boş

P0 sırayı kibarca P1'e verir.

Adım 4 / 9

Uyarı: modern CPU ve derleyiciler yükleme/saklama sırasını değiştirebilir; Peterson ancak `_Atomic` değişkenler (sıralı tutarlılık) ya da bellek bariyerleriyle doğru çalışır. Pratikte donanım atomik komutları ve kilitler kullanılır.

Peterson Çözümü

```
1 bool flag[2] = { false, false };
2 int turn;
3
4 void enter(int i) {
5     int j = 1 - i;
6     flag[i] = true; /* girmek istiyorum */
7     turn = j;      /* ama önce sen */
8     while (flag[j] && turn == j)
9         ;          /* meşgul bekle */
10 }
11 void leave(int i) {
12     flag[i] = false;
13 }
```

DEĞİŞKENLER

flag[0] = true

flag[1] = true

turn = 0

P0

kesildi

P1

turn=0

kritik bölge: boş

P1 de sırayı P0'a verir. `turn` 'e **en son yazan** bekleyecek.

Adım 5 / 9

Uyarı: modern CPU ve derleyiciler yükleme/saklama sırasını değiştirebilir; Peterson ancak `_Atomic` değişkenler (sıralı tutarlılık) ya da bellek bariyerleriyle doğru çalışır. Pratikte donanım atomik komutları ve kilitler kullanılır.

Peterson Çözümü

```
1 bool flag[2] = { false, false };
2 int turn;
3
4 void enter(int i) {
5     int j = 1 - i;
6     flag[i] = true; /* girmek istiyorum */
7     turn = j;       /* ama önce sen */
8     while (flag[j] && turn == j)
9         ;           /* meşgul bekle */
10 }
11 void leave(int i) {
12     flag[i] = false;
13 }
```

DEĞİŞKENLER

flag[0] = true

flag[1] = true

turn = 0

P0

kritik bölgede

P1

hazır

kritik bölge: P0

P0: `flag[1] && turn == 1` ? turn = 0 → yanlış: P0 girer.

Adım 6 / 9

Uyarı: modern CPU ve derleyiciler yükleme/saklama sırasını değiştirebilir; Peterson ancak `_Atomic` değişkenler (sıralı tutarlılık) ya da bellek bariyerleriyle doğru çalışır. Pratikte donanım atomik komutları ve kilitler kullanılır.

Peterson Çözümü

```
1 bool flag[2] = { false, false };
2 int turn;
3
4 void enter(int i) {
5     int j = 1 - i;
6     flag[i] = true; /* girmek istiyorum */
7     turn = j;       /* ama önce sen */
8     while (flag[j] && turn == j)
9         ;           /* meşgul bekle */
10 }
11 void leave(int i) {
12     flag[i] = false;
13 }
```

DEĞİŞKENLER

flag[0] = true

flag[1] = true

turn = 0

P0

kritik bölgede

P1

dönüyor

kritik bölge: P0

P1: `flag[0] && turn == 0` doğru → döngüde döner (CPU harcar).

Adım 7 / 9

Uyarı: modern CPU ve derleyiciler yükleme/saklama sırasını değiştirebilir; Peterson ancak `_Atomic` değişkenler (sıralı tutarlılık) ya da bellek bariyerleriyle doğru çalışır. Pratikte donanım atomik komutları ve kilitler kullanılır.

Peterson Çözümü

```
1 bool flag[2] = { false, false };
2 int turn;
3
4 void enter(int i) {
5     int j = 1 - i;
6     flag[i] = true; /* girmek istiyorum */
7     turn = j;       /* ama önce sen */
8     while (flag[j] && turn == j)
9         ;           /* meşgul bekle */
10 }
11 void leave(int i) {
12     ▶ flag[i] = false;
13 }
```

DEĞİŞKENLER

flag[0] = false

flag[1] = true

turn = 0

P0

çıktı

P1

dönüyor

kritik bölge: boş

P0 çıkarken flag[0] = false yapar.

Adım 8 / 9

Uyarı: modern CPU ve derleyiciler yükleme/saklama sırasını değiştirebilir; Peterson ancak `_Atomic` değişkenler (sıralı tutarlılık) ya da bellek bariyerleriyle doğru çalışır. Pratikte donanım atomik komutları ve kilitler kullanılır.

Peterson Çözümü

```
1 bool flag[2] = { false, false };
2 int turn;
3
4 void enter(int i) {
5     int j = 1 - i;
6     flag[i] = true; /* girmek istiyorum */
7     turn = j;       /* ama önce sen */
8     while (flag[j] && turn == j)
9         ;           /* meşgul bekle */
10 }
11 void leave(int i) {
12     flag[i] = false;
13 }
```

DEĞİŞKENLER

flag[0] = false

flag[1] = true

turn = 0

P0

dışarıda

P1

kritik bölgede

kritik bölge: P1

P1'in koşulu yanlış olur: P1 girer. Karşılıklı dışlama ve ilerleme sağlandı.

Adım 9 / 9

Uyarı: modern CPU ve derleyiciler yükleme/saklama sırasını değiştirebilir; Peterson ancak `_Atomic` değişkenler (sıralı tutarlılık) ya da bellek bariyerleriyle doğru çalışır. Pratikte donanım atomik komutları ve kilitler kullanılır.

Meşgul Bekleme ve Uyu/Uyandır

Meşgul beklemenin sorunları

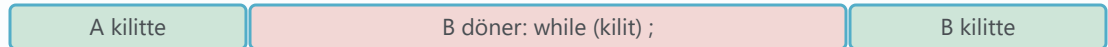
- Kilit beklerken CPU boşa döner.
- **Öncelik terslemesi** (priority inversion): düşük öncelikli L kritik bölgedeyken yüksek öncelikli H gelip dönmeye başlarsa, H hep seçildiği için L hiç çalışıp çıkamaz; H sonsuza dek döner.

Çözüm: bloke et

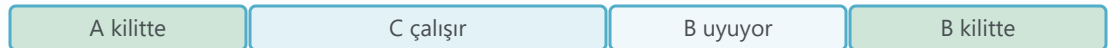
- + `sleep()` : çağıranı bekliyor duruma geçirir, CPU'yu bırakır.
- + `wakeup(p)` : uyuyan `p` 'yi hazır yapar.
- + Ama yanlış kullanılırsa uyandırma sinyali **kaybolabilir**.

Bu durumun literatürdeki adı **öncelik terslemesidir**; kilitlenme (deadlock) ile karıştırılmamalıdır. 1997 Mars Pathfinder sıfırlanmaları bunun ünlü örneğidir; çözüm öncelik kalıtımı (priority inheritance).

meşgul bekleme



uyu / uyandır

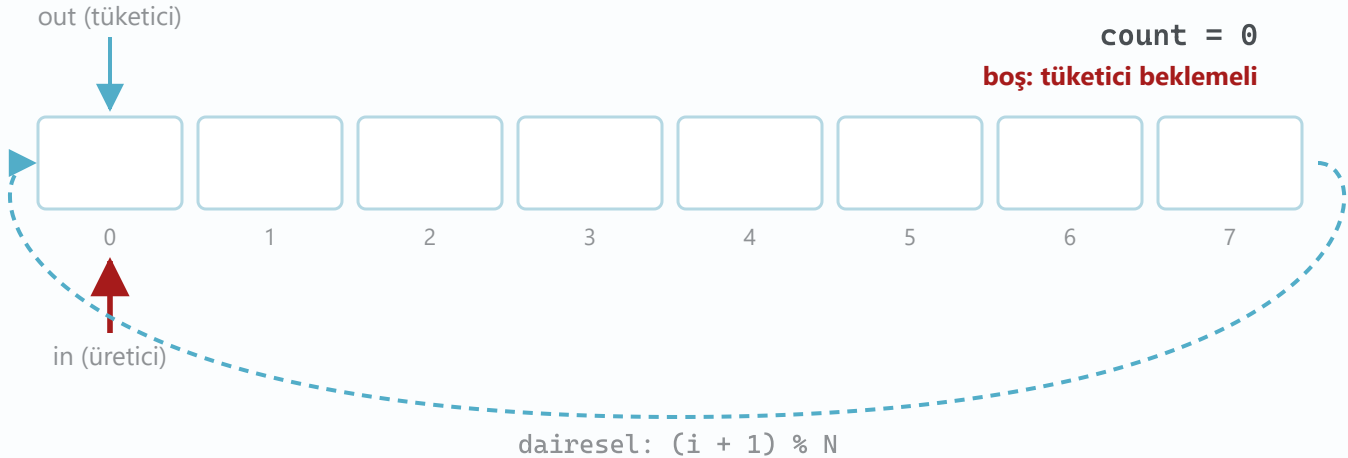


aynı bekleme süresi: üstte CPU boşa döner, altta başka iş yapılır

zaman →

Üretici-Tüketici (Sınırlı Tampon) Problemi

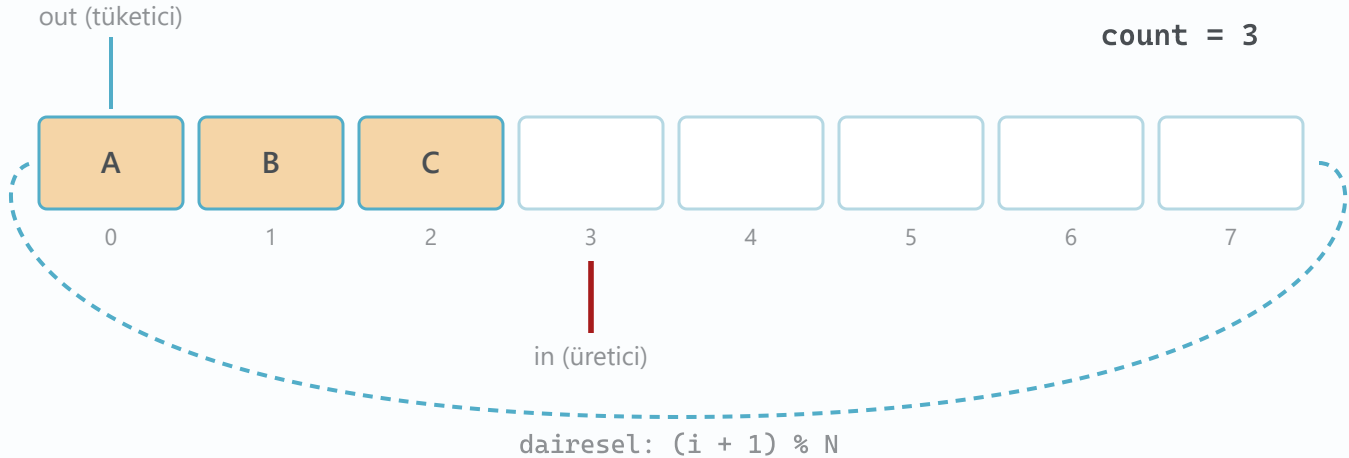
- İki süreç sabit boyutlu (N) bir tamponu paylaşır: **üretici** yazar, **tüketici** okur.
- Sınırlı tampon:** doluyrsa üretici, boşsa tüketici bekler. **Sınırsız tampon:** yalnız tüketici bekler.



Tampon boş (count = 0): tüketici okuyamaz, beklemelidir.

Üretici-Tüketici (Sınırlı Tampon) Problemi

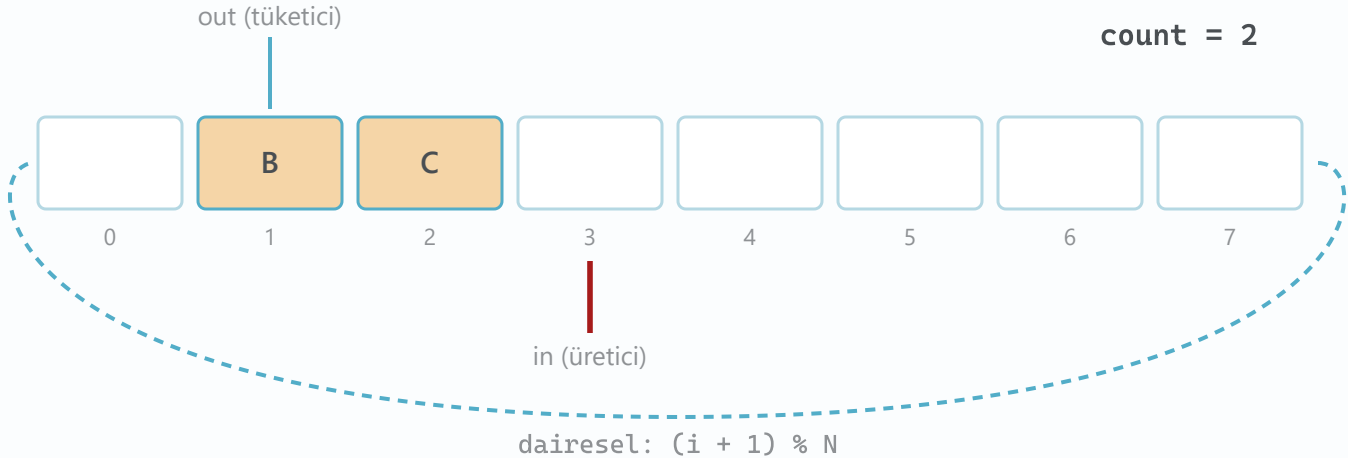
- İki süreç sabit boyutlu (N) bir tamponu paylaşır: **üretici** yazar, **tüketici** okur.
- Sınırlı tampon:** doluyrsa üretici, boşsa tüketici bekler. **Sınırsız tampon:** yalnız tüketici bekler.



Üretici A, B, C ekler: her eklemede `in = (in + 1) % N`.

Üretici-Tüketici (Sınırlı Tampon) Problemi

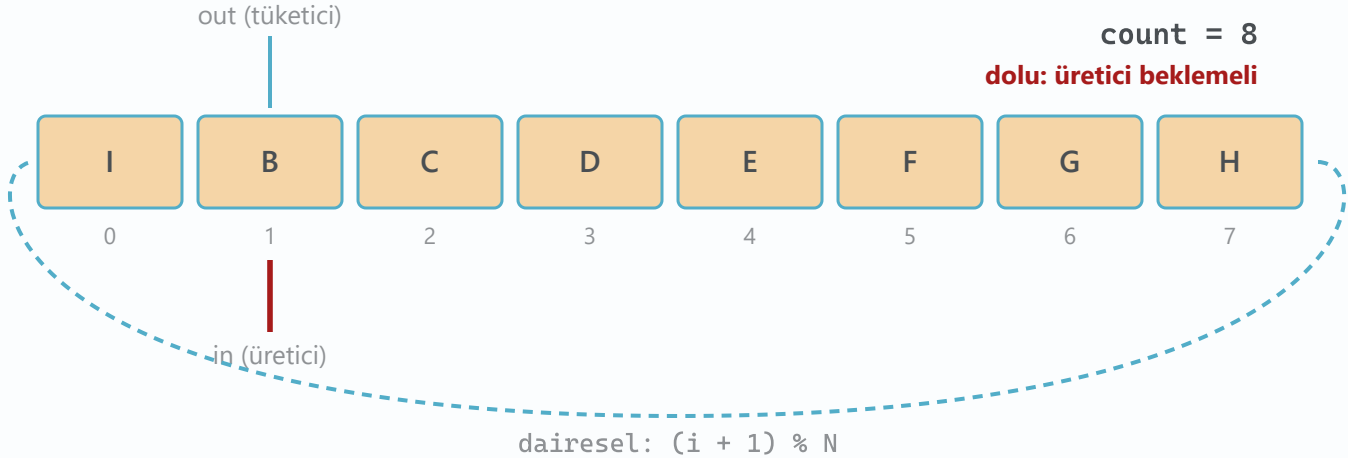
- İki süreç sabit boyutlu (N) bir tamponu paylaşır: **üretici** yazar, **tüketici** okur.
- Sınırlı tampon:** doluyrsa üretici, boşsa tüketici bekler. **Sınırsız tampon:** yalnız tüketici bekler.



Tüketici A'yı alır: **out = 1**, count = 2.

Üretici-Tüketici (Sınırlı Tampon) Problemi

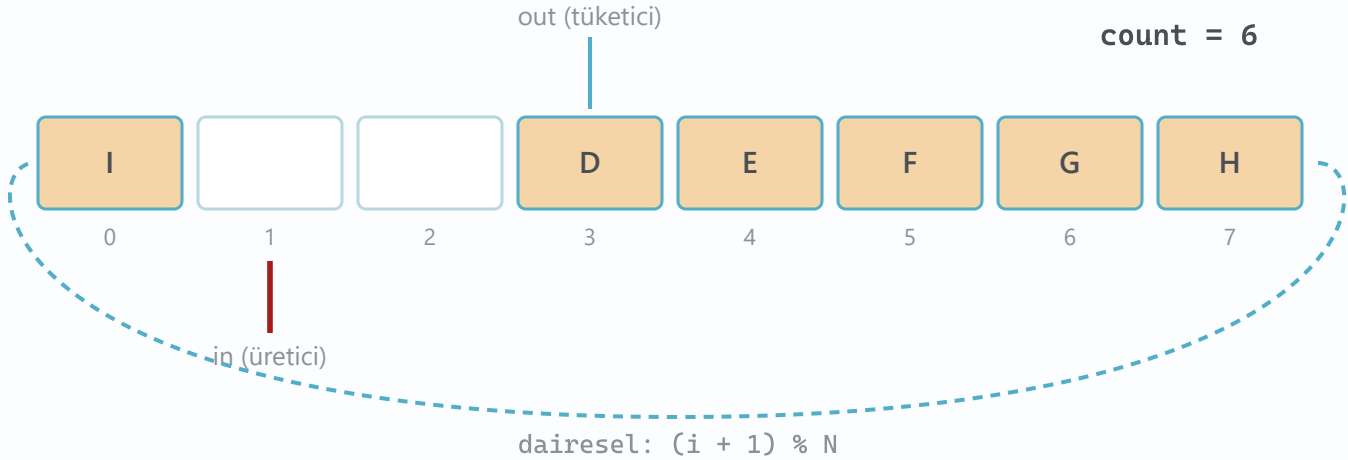
- İki süreç sabit boyutlu (N) bir tamponu paylaşır: **üretici** yazar, **tüketici** okur.
- Sınırlı tampon:** doluyrsa üretici, boşsa tüketici bekler. **Sınırsız tampon:** yalnız tüketici bekler.



Üretici D...I ekler; **in** başa sarar (1). $\text{count} = N = 8$: tampon dolu, üretici beklemelidir.

Üretici-Tüketici (Sınırlı Tampon) Problemi

- İki süreç sabit boyutlu (N) bir tamponu paylaşır: **üretici** yazar, **tüketici** okur.
- Sınırlı tampon:** doluyrsa üretici, boşsa tüketici bekler. **Sınırsız tampon:** yalnız tüketici bekler.



Tüketici B ve C'yi alır: **out = 3**, count = 6; üretici yeniden yazabilir.

Ölümcül Yarış: Kayıp Uyandırma

```
1 #define N 100
2 int count = 0;
3 void producer(void) {
4     while (TRUE) {
5         item = produce_item();
6         if (count == N) sleep();
7         insert_item(item);
8         count = count + 1;
9         if (count == 1) wakeup(consumer);
10    }
11 }
12 void consumer(void) {
13     while (TRUE) {
14         ▶ if (count == 0) sleep();
15         item = remove_item();
16         count = count - 1;
17         if (count == N - 1) wakeup(producer);
18         consume_item(item);
19     }
20 }
```

DEĞİŞKENLER

count = 0

tüketici = uyumak üzere

üretici = hazır

üretici
hazırtüketici
uyumak üzere

count = 0

Tampon boş. Tüketici `count` 'u okur: 0. Tam `sleep()` çağırarak kesilir.

Adım 1 / 7

Ölümcül Yarış: Kayıp Uyandırma

```
1 #define N 100
2 int count = 0;
3 void producer(void) {
4     while (TRUE) {
5         ▶ item = produce_item();
6         if (count == N) sleep();
7         ▶ insert_item(item);
8         count = count + 1;
9         if (count == 1) wakeup(consumer);
10    }
11 }
12 void consumer(void) {
13     while (TRUE) {
14         if (count == 0) sleep();
15         item = remove_item();
16         count = count - 1;
17         if (count == N - 1) wakeup(producer);
18         consume_item(item);
19     }
20 }
```

DEĞİŞKENLER

count = 0

tüketici = uyumak üzere

üretici = çalışıyor

üretici
çalışıyor

tüketici
kesildi

count = 0

Üretici bir öğe üretir ve tampona koyar.

Adım 2 / 7

Ölümcül Yarış: Kayıp Uyandırma

```
1 #define N 100
2 int count = 0;
3 void producer(void) {
4     while (TRUE) {
5         item = produce_item();
6         if (count == N) sleep();
7         insert_item(item);
8         count = count + 1;
9         if (count == 1) wakeup(consumer);
10    }
11 }
12 void consumer(void) {
13     while (TRUE) {
14         if (count == 0) sleep();
15         item = remove_item();
16         count = count - 1;
17         if (count == N - 1) wakeup(producer);
18         consume_item(item);
19    }
20 }
```

DEĞİŞKENLER

count = 1

tüketici = uyumak üzere

üretici = çalışıyor

üretici
çalışıyor

tüketici
kesildi

count = 1

count 1 olur.

Adım 3 / 7

Ölümcül Yarış: Kayıp Uyandırma

```
1 #define N 100
2 int count = 0;
3 void producer(void) {
4     while (TRUE) {
5         item = produce_item();
6         if (count == N) sleep();
7         insert_item(item);
8         count = count + 1;
9         if (count == 1) wakeup(consumer);
10    }
11 }
12 void consumer(void) {
13     while (TRUE) {
14         if (count == 0) sleep();
15         item = remove_item();
16         count = count - 1;
17         if (count == N - 1) wakeup(producer);
18         consume_item(item);
19     }
20 }
```

DEĞİŞKENLER

count = 1

tüketici = uyumak üzere

üretici = çalışıyor

wakeup = kayboldu

üretici

wakeup → kayıp

tüketici

kesildi

count = 1

count == 1 → tüketici uyuyor sanılır,
wakeup(consumer) gönderilir. Ama tüketici henüz
uyumadı: sinyal **kaybolur**.

Adım 4 / 7

Ölümcül Yarış: Kayıp Uyandırma

```
1 #define N 100
2 int count = 0;
3 void producer(void) {
4     while (TRUE) {
5         item = produce_item();
6         if (count == N) sleep();
7         insert_item(item);
8         count = count + 1;
9         if (count == 1) wakeup(consumer);
10    }
11 }
12 void consumer(void) {
13     while (TRUE) {
14         ▶ if (count == 0) sleep();
15         item = remove_item();
16         count = count - 1;
17         if (count == N - 1) wakeup(producer);
18         consume_item(item);
19     }
20 }
```

DEĞİŞKENLER

count = 1

tüketici = uyuyor

üretici = hazır

üretici
hazırtüketici
uyuyor

count = 1

Tüketici kaldığı yerden devam eder: eski kararına göre `sleep()` çağırır ve uyur.

Adım 5 / 7

Ölümcül Yarış: Kayıp Uyandırma

```
1 #define N 100
2 int count = 0;
3 void producer(void) {
4     while (TRUE) {
5         item = produce_item();
6         if (count == N) sleep();
7         insert_item(item);
8         count = count + 1;
9         if (count == 1) wakeup(consumer);
10    }
11 }
12 void consumer(void) {
13     while (TRUE) {
14         if (count == 0) sleep();
15         item = remove_item();
16         count = count - 1;
17         if (count == N - 1) wakeup(producer);
18         consume_item(item);
19    }
20 }
```

DEĞİŞKENLER

count = 100

tüketici = uyuyor

üretici = çalışıyor

üretici
çalışıyortüketici
uyuyor

count = 100

Üretici tamponu doldurur; tüketici uyuduğu için kimse okumuyor.

Adım 6 / 7

Ölümcül Yarış: Kayıp Uyandırma

```
1 #define N 100
2 int count = 0;
3 void producer(void) {
4     while (TRUE) {
5         item = produce_item();
6         ► if (count == N) sleep();
7         insert_item(item);
8         count = count + 1;
9         if (count == 1) wakeup(consumer);
10    }
11 }
12 void consumer(void) {
13     while (TRUE) {
14         if (count == 0) sleep();
15         item = remove_item();
16         count = count - 1;
17         if (count == N - 1) wakeup(producer);
18         consume_item(item);
19    }
20 }
```

DEĞİŞKENLER

count = 100

tüketici = uyuyor

üretici = uyuyor

üretici
uyuyortüketici
uyuyor

count = 100

count == N → üretici de uyur. **İkisi de sonsuza dek uyur.** Sorun: `count` testi ile `sleep` bölünmez (atomik) değil.

Adım 7 / 7

Semaforlarla Üretici-Tüketici

```
semaphore mutex = 1, empty = N, full = 0;

void producer(void) {
    while (TRUE) {
        item = produce_item();
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}

void consumer(void) {
    while (TRUE) {
        down(&full);
        down(&mutex);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}
```

empty / full

Boş ve dolu yuvaları sayar: sıralamayı (kim bekler) sağlar. Sayaç ile bekleme **atomik** olduğu için uyandırma kaybolmaz.

mutex

Tampona aynı anda tek erişim: karşılıklı dışlama.

Dikkat

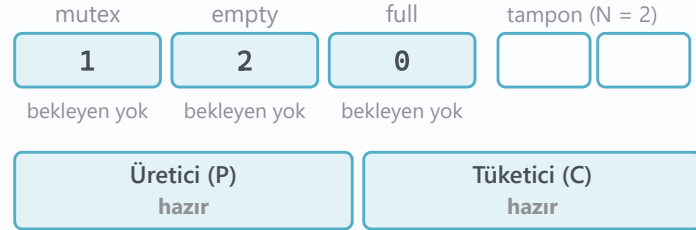
Üreticide `down(&mutex)` ile `down(&empty)` yer değiştirirse tampon doluyken kilitlenme olur.

Semafor, mutex ve monitörler Bölüm 6'da ayrıntılı işlenecektir.

Kod İzleme: Semaforlarla Üretici-Tüketici

```
► semaphore mutex = 1, empty = 2, full = 0;
```

```
2 void producer(void) {  
3   while (TRUE) {  
4     item = produce_item();  
5     down(&empty);  
6     down(&mutex);  
7     insert_item(item);  
8     up(&mutex); up(&full);  
9   } }  
10 void consumer(void) {  
11   while (TRUE) {  
12     down(&full);  
13     down(&mutex); item = remove_item();  
14     up(&mutex); up(&empty);  
15     consume_item(item);  
16   } }
```



Tampon 2 yuvalı ve boş: **empty = 2** boş yuva, **full = 0** dolu yuva sayar; **mutex = 1** tampona tek erişimi sağlar.

Adım 1 / 12

DEĞİŞKENLER

mutex = 1

empty = 2

full = 0

P.item = ?

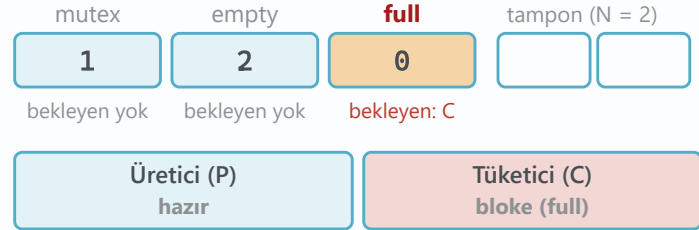
C.item = ?

ÇIKTI



Kod İzleme: Semaforlarla Üretici-Tüketici

```
1 semaphore mutex = 1, empty = 2, full = 0;
2 void producer(void) {
3     while (TRUE) {
4         item = produce_item();
5         down(&empty);
6         down(&mutex);
7         insert_item(item);
8         up(&mutex); up(&full);
9     } }
10 void consumer(void) {
11     while (TRUE) {
12         ▶ down(&full);
13         down(&mutex); item = remove_item();
14         up(&mutex); up(&empty);
15         consume_item(item);
16     } }
```



Tüketici önce çalışır: `down(&full)`, `full = 0` → **bloke** olur ve `full`'ün kuyruğuna girer. Uyandırma sayaçta değil kuyrukta saklandığı için kaybolamaz.

Adım 2 / 12

DEĞİŞKENLER

mutex = 1

empty = 2

full = 0

P.item = ?

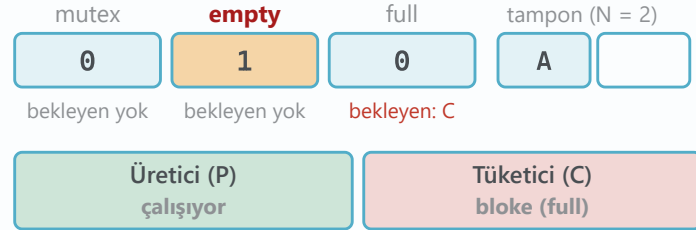
C.item = ?

ÇIKTI



Kod İzleme: Semaforlarla Üretici-Tüketici

```
1 semaphore mutex = 1, empty = 2, full = 0;
2 void producer(void) {
3   while (TRUE) {
4     ▶ item = produce_item();
5     ▶ down(&empty);
6     ▶ down(&mutex);
7     ▶ insert_item(item);
8     up(&mutex); up(&full);
9   } }
10 void consumer(void) {
11   while (TRUE) {
12     down(&full);
13     down(&mutex); item = remove_item();
14     up(&mutex); up(&empty);
15     consume_item(item);
16   } }
```



Üretici A'yı üretir: empty 2 → 1, mutex 1 → 0, A
tampona girer.

Adım 3 / 12

DEĞİŞKENLER

mutex = 0

empty = 1

full = 0

P.item = A

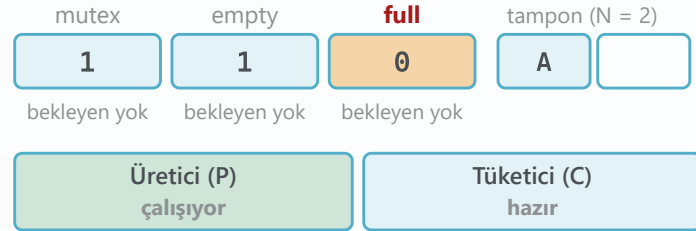
C.item = ?

ÇIKTI



Kod İzleme: Semaforlarla Üretici-Tüketici

```
1 semaphore mutex = 1, empty = 2, full = 0;
2 void producer(void) {
3     while (TRUE) {
4         item = produce_item();
5         down(&empty);
6         down(&mutex);
7         insert_item(item);
8         ► up(&mutex); up(&full);
9     }
10 void consumer(void) {
11     while (TRUE) {
12         down(&full);
13         down(&mutex); item = remove_item();
14         up(&mutex); up(&empty);
15         consume_item(item);
16     } }
```



`up(&full)` : kuyrukta tüketici var, onu **uyandırır**; sayaç artmaz (full = 0). Tüketici hazır ama CPU henüz üreticide.

Adım 4 / 12

DEĞİŞKENLER

mutex = 1

empty = 1

full = 0

P.item = A

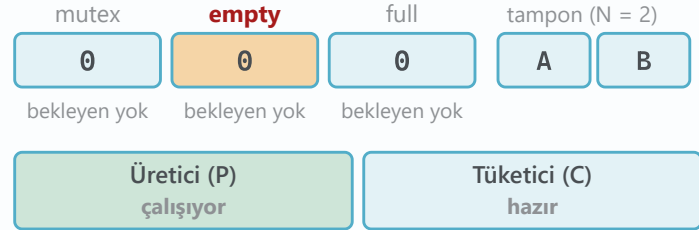
C.item = ?

ÇIKTI



Kod İzleme: Semaforlarla Üretici-Tüketici

```
1 semaphore mutex = 1, empty = 2, full = 0;
2 void producer(void) {
3   while (TRUE) {
4     ▶ item = produce_item();
5     ▶ down(&empty);
6     ▶ down(&mutex);
7     ▶ insert_item(item);
8     up(&mutex); up(&full);
9   }
10 void consumer(void) {
11   while (TRUE) {
12     down(&full);
13     down(&mutex); item = remove_item();
14     up(&mutex); up(&empty);
15     consume_item(item);
16   } }
```



B üretilir: empty 1 → 0, B tampona girer. Tampon dolu.

Adım 5 / 12

DEĞİŞKENLER

mutex = 0

empty = 0

full = 0

P.item = B

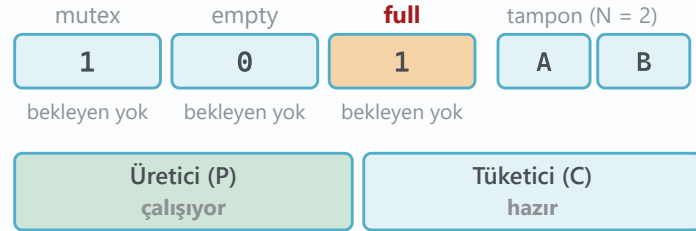
C.item = ?

ÇIKTI



Kod İzleme: Semaforlarla Üretici-Tüketici

```
1 semaphore mutex = 1, empty = 2, full = 0;
2 void producer(void) {
3     while (TRUE) {
4         item = produce_item();
5         down(&empty);
6         down(&mutex);
7         insert_item(item);
8         ► up(&mutex); up(&full);
9     }
10 void consumer(void) {
11     while (TRUE) {
12         down(&full);
13         down(&mutex); item = remove_item();
14         up(&mutex); up(&empty);
15         consume_item(item);
16     } }
```



Bu kez bekleyen yok: **full = 1**. Tampondaki 2 öğeden biri uyanan tüketiciye ayrılmış, biri sayaçta.

Adım 6 / 12

DEĞİŞKENLER

mutex = 1

empty = 0

full = 1

P.item = B

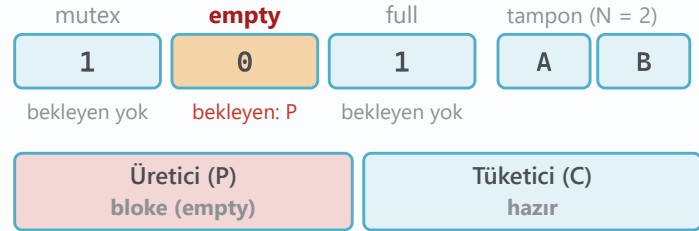
C.item = ?

ÇIKTI



Kod İzleme: Semaforlarla Üretici-Tüketici

```
1 semaphore mutex = 1, empty = 2, full = 0;
2 void producer(void) {
3     while (TRUE) {
4         ▶ item = produce_item();
5         ▶ down(&empty);
6         down(&mutex);
7         insert_item(item);
8         up(&mutex); up(&full);
9     }
10 void consumer(void) {
11     while (TRUE) {
12         down(&full);
13         down(&mutex); item = remove_item();
14         up(&mutex); up(&empty);
15         consume_item(item);
16     } }
```



Üçüncü öğe üretilir ama empty = 0: üretici **bloke**. Dolu tampona yazma böylece engellenir.

Adım 7 / 12

DEĞİŞKENLER

mutex = 1

empty = 0

full = 1

P.item = C

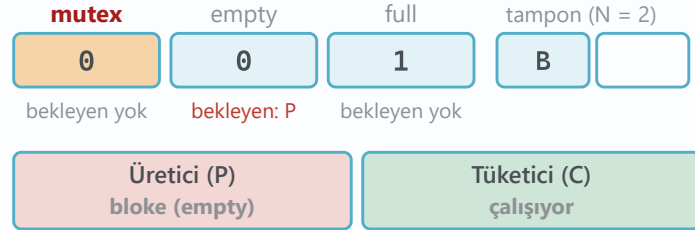
C.item = ?

ÇIKTI



Kod İzleme: Semaforlarla Üretici-Tüketici

```
1 semaphore mutex = 1, empty = 2, full = 0;
2 void producer(void) {
3     while (TRUE) {
4         item = produce_item();
5         down(&empty);
6         down(&mutex);
7         insert_item(item);
8         up(&mutex); up(&full);
9     } }
10 void consumer(void) {
11     while (TRUE) {
12         down(&full);
13         ▶ down(&mutex); item = remove_item();
14         up(&mutex); up(&empty);
15         consume_item(item);
16     } }
```



Tüketici çalışır; `down(&full)` 'dan uyandırılmış olarak döner. mutex 1 → 0 ve **A** tampondan alınır.

Adım 8 / 12

DEĞİŞKENLER

mutex = 0

empty = 0

full = 1

P.item = C

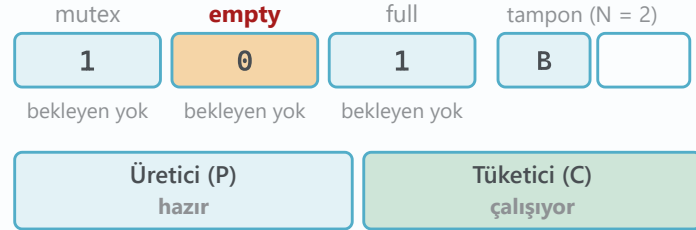
C.item = A

ÇIKTI



Kod İzleme: Semaforlarla Üretici-Tüketici

```
1 semaphore mutex = 1, empty = 2, full = 0;
2 void producer(void) {
3     while (TRUE) {
4         item = produce_item();
5         down(&empty);
6         down(&mutex);
7         insert_item(item);
8         up(&mutex); up(&full);
9     } }
10 void consumer(void) {
11     while (TRUE) {
12         down(&full);
13         down(&mutex); item = remove_item();
14         up(&mutex); up(&empty);
15         consume_item(item);
16     } }
```



`up(&empty)` : bekleyen üretici uyandırılır (empty 0 kalır, boş yuva ona ayrıldı).

Adım 9 / 12

DEĞİŞKENLER

mutex = 1

empty = 0

full = 1

P.item = C

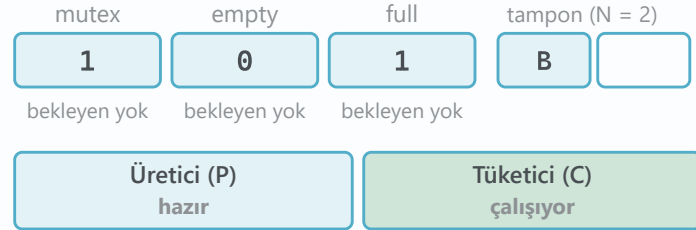
C.item = A

ÇIKTI



Kod İzleme: Semaforlarla Üretici-Tüketici

```
1 semaphore mutex = 1, empty = 2, full = 0;
2 void producer(void) {
3     while (TRUE) {
4         item = produce_item();
5         down(&empty);
6         down(&mutex);
7         insert_item(item);
8         up(&mutex); up(&full);
9     }
10 void consumer(void) {
11     while (TRUE) {
12         down(&full);
13         down(&mutex); item = remove_item();
14         up(&mutex); up(&empty);
15         consume_item(item);
16     } }
```



A tüketilir.

Adım 10 / 12

DEĞİŞKENLER

mutex = 1

empty = 0

full = 1

P.item = C

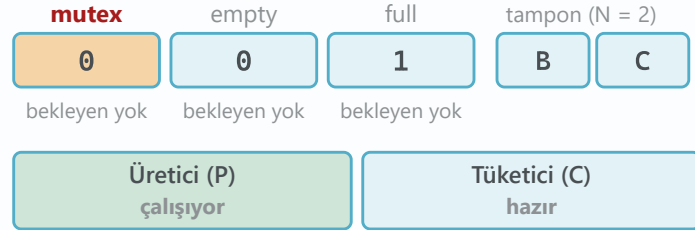
C.item = A

ÇIKTI

A tüketildi

Kod İzleme: Semaforlarla Üretici-Tüketici

```
1 semaphore mutex = 1, empty = 2, full = 0;
2 void producer(void) {
3     while (TRUE) {
4         item = produce_item();
5         down(&empty);
6         down(&mutex);
7         insert_item(item);
8         up(&mutex); up(&full);
9     }
10 void consumer(void) {
11     while (TRUE) {
12         down(&full);
13         down(&mutex); item = remove_item();
14         up(&mutex); up(&empty);
15         consume_item(item);
16     } }
```



Üretici `down(&empty)` 'dan döner, mutex'i alır ve C'yi tampona koyar: [B, C].

Adım 11 / 12

DEĞİŞKENLER

mutex = 0

empty = 0

full = 1

P.item = C

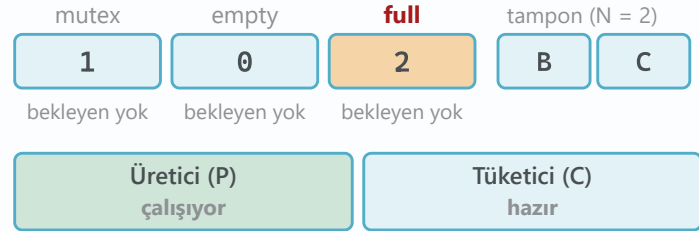
C.item = A

ÇIKTI

A tüketildi

Kod İzleme: Semaforlarla Üretici-Tüketici

```
1 semaphore mutex = 1, empty = 2, full = 0;
2 void producer(void) {
3     while (TRUE) {
4         item = produce_item();
5         down(&empty);
6         down(&mutex);
7         insert_item(item);
8         ▶ up(&mutex); up(&full);
9     }
10 void consumer(void) {
11     while (TRUE) {
12         down(&full);
13         down(&mutex); item = remove_item();
14         up(&mutex); up(&empty);
15         consume_item(item);
16     } }
```



full = 2: tampondaki iki öğeyle tutarlı. Hiçbir uyandırma kaybolmadı, tampon ne taşıtı ne de boşken okundu.

Adım 12 / 12

DEĞİŞKENLER

mutex = 1

empty = 0

full = 2

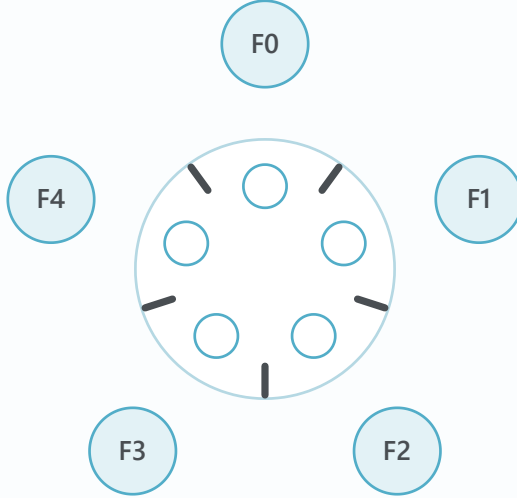
P.item = C

C.item = A

ÇIKTI

A tüketildi

Yemek Yiyen Filozoflar (Dining Philosophers)



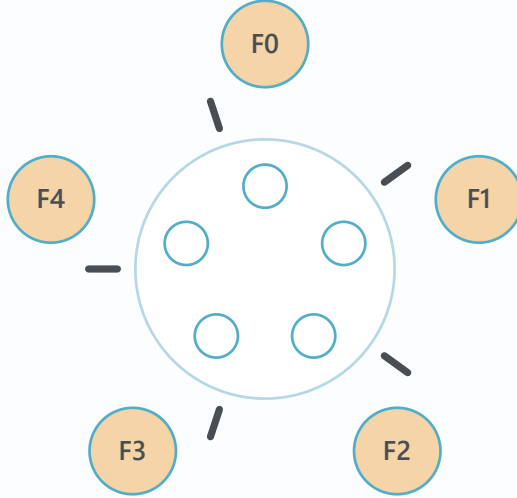
5 filozof, 5 çatal.
Yemek için iki çatal gerekir:
sol ve sağ.

Her filozof düşünür ya da yer; yemek için solundaki ve sağındaki iki çatalı da alması gerekir.

Adım 1 / 3

Dijkstra (1965): sınırlı kaynakların (G/Ç aygıtları gibi) birden çok süreç arasında kilitlenmeden paylaşılmasını modeller.

Yemek Yiyen Filozoflar (Dining Philosophers)



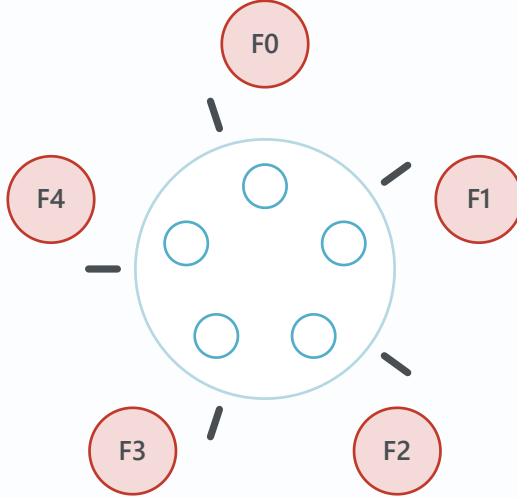
Hepsi aynı anda acıkır ve
sol çatalını alır.
Masada çatal kalmaz.

Saf çözüm: `pick_up_left_fork(); pick_up_right_fork();` . Hepsi aynı anda acıkıp solu alırsa...

Adım 2 / 3

Dijkstra (1965): sınırlı kaynakların (G/Ç aygıtları gibi) birden çok süreç arasında kilitlenmeden paylaşılmasını modeller.

Yemek Yiyen Filozoflar (Dining Philosophers)



Herkes sağındakini bekler:
döngüsel bekleme →
kilitlenme (deadlock).

...her biri sağındakini bekler; kimse bırakmaz. Döngüsel bekleme: **kilitlenme**. Rastgele bekleyip tekrar denemek ise açlığa (starvation) yol açabilir.

Adım 3 / 3

Dijkstra (1965): sınırlı kaynakların (G/Ç aygıtları gibi) birden çok süreç arasında kilitlenmeden paylaşılmasını modeller.

Filozoflar: Durum Dizisi ile Çözüm

```
#define N 5
#define LEFT (i + N - 1) % N
#define RIGHT (i + 1) % N
int state[N]; /* THINKING, HUNGRY, EATING */
semaphore mutex = 1; /* state[] için */
semaphore s[N]; /* filozof başına, başta 0 */

void take_forks(int i) {
    down(&mutex);
    state[i] = HUNGRY;
    test(i); /* 2 çatal al */
    up(&mutex);
    down(&s[i]); /* alamadıysa */
}

void put_forks(int i) {
    down(&mutex);
    state[i] = THINKING;
    test(LEFT); /* komşular */
    test(RIGHT); /* yiyebilir mi? */
    up(&mutex);
}

void test(int i) {
    if (state[i] == HUNGRY &&
        state[LEFT] != EATING &&
        state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}

void philosopher(int i) {
    while (TRUE) {
        think(); take_forks(i);
        eat(); put_forks(i);
    }
}
```

Filozof ancak iki komşusu da yemiyorsa yer; çatalar tek tek değil, durum dizisi üzerinden **birlikte** alınır. Kilitlenme olmaz ve en fazla $\lfloor N/2 \rfloor = 2$ filozof aynı anda yer.

Okur-Yazar Problemi

```
semaphore mutex = 1; /* rc için */
semaphore db = 1; /* veritabanı için */
int rc = 0; /* okuyan sayısı */

void writer(void) {
    while (TRUE) {
        think_up_data();
        down(&db);
        write_data_base();
        up(&db);
    }
}

void reader(void) {
    while (TRUE) {
        down(&mutex);
        rc = rc + 1;
        if (rc == 1) down(&db);
        up(&mutex);
        read_data_base();
        down(&mutex);
        rc = rc - 1;
        if (rc == 0) up(&db);
        up(&mutex);
        use_data_read();
    }
}
```

- Veritabanı erişimini modeller: **çok okur aynı anda** okuyabilir; yazar ise **tek başına** çalışmalıdır.
- İlk okur **db** 'yi kilitler, son okur bırakır.

Dezavantaj: okurlar sürekli gelirse **rc** hiç 0 olmaz, yazar **açlık** çeker. Çözüm: yazar öncelikli sürüm ya da `pthread_rwlock`.

Kod İzleme: Okurlar ve Bir Yazar

```
▶ semaphore mutex = 1, db = 1;
▶ int rc = 0; /* okur sayısı */
3 void reader(void) {
4   down(&mutex); rc = rc + 1;
5   if (rc == 1) down(&db); /* ilk */
6   up(&mutex);
7   read_data_base();
8   down(&mutex); rc = rc - 1;
9   if (rc == 0) up(&db); /* son */
10  up(&mutex);
11 }
12 void writer(void) {
13   down(&db); write_data_base(); up(&db);
14 }
```

rc

0

db

1

db bekleyen: yok

veritabanı

Veritabanı boş: kimse okumuyor ya da yazmıyor. **db** veritabanının kendisini, **mutex** yalnızca **rc** sayacını korur.

Adım 1 / 11

DEĞİŞKENLER

rc = 0

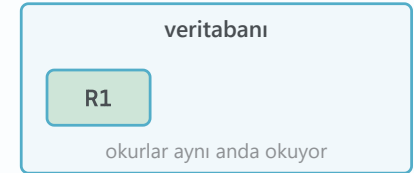
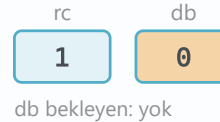
db = 1

ÇAĞRI YIĞINI

boş

Kod İzleme: Okurlar ve Bir Yazar

```
1 semaphore mutex = 1, db = 1;
2 int rc = 0; /* okur sayısı */
3 void reader(void) {
4     ▶ down(&mutex); rc = rc + 1;
5     ▶ if (rc == 1) down(&db); /* ilk */
6     up(&mutex);
7     read_data_base();
8     down(&mutex); rc = rc - 1;
9     if (rc == 0) up(&db); /* son */
10    up(&mutex);
11 }
12 void writer(void) {
13     down(&db); write_data_base(); up(&db);
14 }
```



R1 gelir: rc = 1. İlk okur olduğu için `down(&db)` ile veritabanını **okurlar adına** kilitler: db = 0.

Adım 2 / 11

DEĞİŞKENLER

rc = 1 db = 0

ÇAĞRI YİĞİNİ

R1: reader()

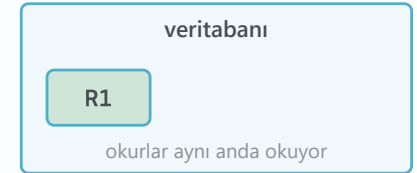
Kod İzleme: Okurlar ve Bir Yazar

```
1 semaphore mutex = 1, db = 1;
2 int rc = 0; /* okur sayısı */
3 void reader(void) {
4     down(&mutex); rc = rc + 1;
5     if (rc == 1) down(&db); /* ilk */
6     ▶ up(&mutex);
7     ▶ read_data_base();
8     down(&mutex); rc = rc - 1;
9     if (rc == 0) up(&db); /* son */
10    up(&mutex);
11 }
12 void writer(void) {
13     down(&db); write_data_base(); up(&db);
14 }
```

rc db

1	0
---	---

db bekleyen: yok



R1 mutex'i bırakır ve okumaya başlar.

Adım 3 / 11

DEĞİŞKENLER

rc = 1 db = 0

ÇAĞRI YİĞİNİ

R1: reader()

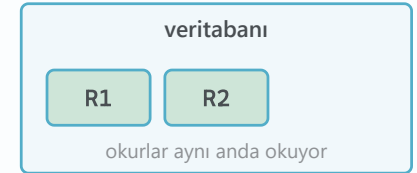
Kod İzleme: Okurlar ve Bir Yazar

```
1 semaphore mutex = 1, db = 1;
2 int rc = 0; /* okur sayısı */
3 void reader(void) {
4     ▶ down(&mutex); rc = rc + 1;
5     ▶ if (rc == 1) down(&db); /* ilk */
6     ▶ up(&mutex);
7     read_data_base();
8     down(&mutex); rc = rc - 1;
9     if (rc == 0) up(&db); /* son */
10    up(&mutex);
11 }
12 void writer(void) {
13     down(&db); write_data_base(); up(&db);
14 }
```

rc db

2	0
---	---

db bekleyen: yok



R2 gelir: rc = 2. İlk okur değil, db'ye dokunmaz; doğrudan okumaya katılır.

Adım 4 / 11

DEĞİŞKENLER

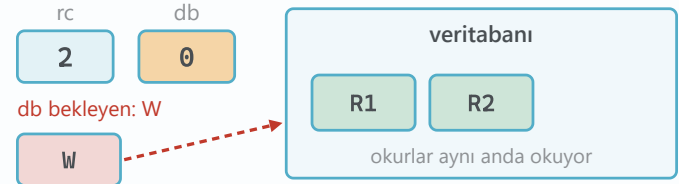
rc = 2 db = 0

ÇAĞRI YİĞİNİ

R2: reader()

Kod İzleme: Okurlar ve Bir Yazar

```
1 semaphore mutex = 1, db = 1;
2 int rc = 0; /* okur sayısı */
3 void reader(void) {
4     down(&mutex); rc = rc + 1;
5     if (rc == 1) down(&db); /* ilk */
6     up(&mutex);
7     read_data_base();
8     down(&mutex); rc = rc - 1;
9     if (rc == 0) up(&db); /* son */
10    up(&mutex);
11 }
12 void writer(void) {
13     down(&db); write_data_base(); up(&db);
14 }
```



Yazar W gelir: `down(&db)`, $db = 0 \rightarrow$ **bloke**.
Okuma sürerken yazma yapılamaz.

Adım 5 / 11

DEĞİŞKENLER

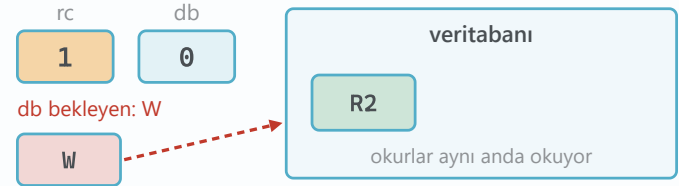
rc = 2 db = 0

ÇAĞRI YİĞİNİ

W: writer()

Kod İzleme: Okurlar ve Bir Yazar

```
1 semaphore mutex = 1, db = 1;
2 int rc = 0; /* okur sayısı */
3 void reader(void) {
4     down(&mutex); rc = rc + 1;
5     if (rc == 1) down(&db); /* ilk */
6     up(&mutex);
7     read_data_base();
8     ▶ down(&mutex); rc = rc - 1;
9     ▶ if (rc == 0) up(&db); /* son */
10    ▶ up(&mutex);
11 }
12 void writer(void) {
13     down(&db); write_data_base(); up(&db);
14 }
```



R1 bitirir: $rc = 1 \neq 0$, db bırakılmaz; içeride hâlâ R2 var.

Adım 6 / 11

DEĞİŞKENLER

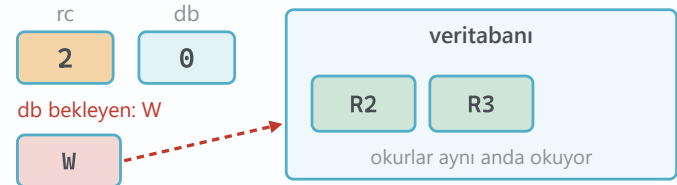
rc = 1, db = 0

ÇAĞRI YİĞİNİ

R1: reader()

Kod İzleme: Okurlar ve Bir Yazar

```
1 semaphore mutex = 1, db = 1;
2 int rc = 0; /* okur sayısı */
3 void reader(void) {
4     ▶ down(&mutex); rc = rc + 1;
5     ▶ if (rc == 1) down(&db); /* ilk */
6     ▶ up(&mutex);
7     read_data_base();
8     down(&mutex); rc = rc - 1;
9     if (rc == 0) up(&db); /* son */
10    up(&mutex);
11 }
12 void writer(void) {
13     down(&db); write_data_base(); up(&db);
14 }
```



R3 gelir: rc = 2. W önce gelmişti ama R3 hemen içeri girer: **okurlar öncelikli**.

Adım 7 / 11

DEĞİŞKENLER

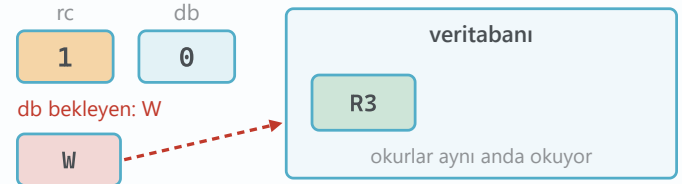
rc = 2, db = 0

ÇAĞRI YİĞİNİ

R3: reader()

Kod İzleme: Okurlar ve Bir Yazar

```
1 semaphore mutex = 1, db = 1;
2 int rc = 0; /* okur sayısı */
3 void reader(void) {
4     down(&mutex); rc = rc + 1;
5     if (rc == 1) down(&db); /* ilk */
6     up(&mutex);
7     read_data_base();
8     ▶ down(&mutex); rc = rc - 1;
9     ▶ if (rc == 0) up(&db); /* son */
10    ▶ up(&mutex);
11 }
12 void writer(void) {
13     down(&db); write_data_base(); up(&db);
14 }
```



R2 bitirir: rc = 1. Okurlar sürekli gelirse rc hiç 0 olmaz ve W **açlık** çeker.

Adım 8 / 11

DEĞİŞKENLER

rc = 1, db = 0

ÇAĞRI YİĞİNİ

R2: reader()

Kod İzleme: Okurlar ve Bir Yazar

```
1 semaphore mutex = 1, db = 1;
2 int rc = 0; /* okur sayısı */
3 void reader(void) {
4     down(&mutex); rc = rc + 1;
5     if (rc == 1) down(&db); /* ilk */
6     up(&mutex);
7     read_data_base();
8     ▶ down(&mutex); rc = rc - 1;
9     ▶ if (rc == 0) up(&db); /* son */
10    up(&mutex);
11 }
12 void writer(void) {
13     down(&db); write_data_base(); up(&db);
14 }
```

rc db

0 0

db bekleyen: yok

veritabanı

R3 son okur: rc = 0 → up(&db). db'de bekleyen W uyandırılır (db 0 kalır, W'ye geçti).

Adım 9 / 11

DEĞİŞKENLER

rc = 0 db = 0

ÇAĞRI YİĞİNİ

R3: reader()

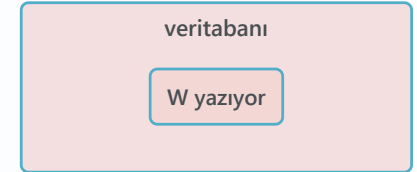
Kod İzleme: Okurlar ve Bir Yazar

```
1 semaphore mutex = 1, db = 1;
2 int rc = 0; /* okur sayısı */
3 void reader(void) {
4     down(&mutex); rc = rc + 1;
5     if (rc == 1) down(&db); /* ilk */
6     up(&mutex);
7     read_data_base();
8     down(&mutex); rc = rc - 1;
9     if (rc == 0) up(&db); /* son */
10    up(&mutex);
11 }
12 void writer(void) {
13     ▶ down(&db); write_data_base(); up(&db);
14 }
```

rc db

0 0

db bekleyen: yok



W veritabanına **tek başına** yazar; bu sırada gelen okurlar ilk okurun `down(&db)` 'sinde bekler.

Adım 10 / 11

DEĞİŞKENLER

rc = 0 db = 0

ÇAĞRI YİĞİNİ

W: writer()

Kod İzleme: Okurlar ve Bir Yazar

```
1 semaphore mutex = 1, db = 1;
2 int rc = 0; /* okur sayısı */
3 void reader(void) {
4     down(&mutex); rc = rc + 1;
5     if (rc == 1) down(&db); /* ilk */
6     up(&mutex);
7     read_data_base();
8     down(&mutex); rc = rc - 1;
9     if (rc == 0) up(&db); /* son */
10    up(&mutex);
11 }
12 void writer(void) {
13     down(&db); write_data_base(); up(&db);
14 }
```

rc

0

db

1

db bekleyen: yok

veritabanı

W `up(&db)` ile bırakır: db = 1, sistem başlangıç durumunda.

Adım 11 / 11

DEĞİŞKENLER

rc = 0

db = 1

ÇAĞRI YİĞİNİ

W: writer()

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, öbek, 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
socketler çekirdek
aracılığıyla çalışır.

Eş zamanlılık

Paylaşılan veride yarış
durumu; kritik bölgeler
karşılıklı dışlamayla
korunmalı.

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](#)