Bölüm 6: Senkronizasyon

İşletim Sistemleri

Yarış durumu | Kritik bölge gereksinimleri | Peterson ve bellek modeli | Bellek bariyerleri | test_and_set, CAS | Atomik değişkenler | Spinlock ve mutex | Semaforlar | Klasik problemler | Monitör ve koşul değişkeni | POSIX ve Java API | Öncelik terslemesi, kilitsiz yapılar

Sercan KÜLCÜ · Giresun Üniversitesi · Bilgisayar Mühendisliği

© Sercan Külcü. Tüm hakları saklıdır. · Sunumun etkileşimli HTML sürümü: https://sercankulcu.github.io/files/os/slides/Bolum_06_Senkronizasyon.html

GİRİŞ

Öğrenme Çıktıları

Problem
Yarış durumunu, kritik bölgeyi ve bir çözümün sağlaması gereken üç koşulu (karşılıklı dışlama, ilerleme, sınırlı bekleme) açıklamak.

Yazılım ve bellek modeli
Kilit değişkeni, katı sıra ve Peterson çözümünü değerlendirmek; modern CPU'larda yeniden sıralamanın etkisini ve bellek bariyerlerini açıklamak.

Donanım desteği
`test_and_set`, `compare_and_swap`, XCHG ve atomik değişkenlerle kilit ve sayaç gerçekleştirmek.

Araçlar
Spinlock, mutex, semafor, monitör ve koşul değişkenini karşılaştırmak; Mesa anlamındaki `while` kuralını uygulamak.

Klasik problemler
Sınırlı tampon, okur-yazar ve yemek yiyen filozoflar problemlerini çözmek; açlık ve kilitlenme risklerini görmek.

Gerçek sistemler
POSIX ve Java API'lerini kullanmak; öncelik terslemesini (Mars Pathfinder) ve kilitsiz yapıların temelini yorumlamak.

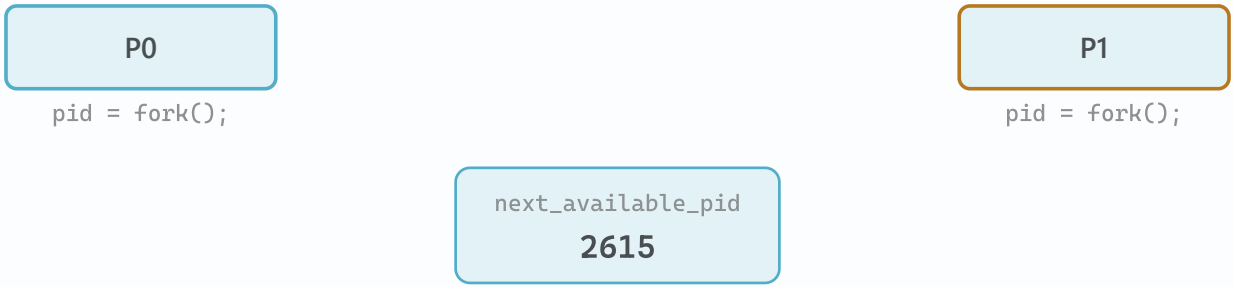
Neden Senkronizasyon?

- **Senkronizasyon:** eş zamanlı yürüyen süreç ya da iş parçacıklarının paylaşılan verilere erişimini **düzenleyerek tutarlılığı** korumak.
- Bir süreç komutlarının **herhangi bir noktasında** kesilebilir; yaptığı güncelleme yarım kalmış olabilir.
- Çok çekirdekli sistemlerde kesme bile gerekmez: iki çekirdek **gerçekten aynı anda** aynı belleğe yazabilir.
- Belirtiler:
  - tutarsız veri, kayıp güncelleme, bozuk bağlı liste;
  - yeniden girilemeyen (non-reentrant) fonksiyonun ortak tamponunun ezilmesi;
  - birbirini sonsuza dek bekleyen iş parçacıkları (**kilitlenme**);
  - hiç sıra gelmeyen iş parçacığı (**açlık**).



GİRİŞ

Yarış Durumu: fork() ve pid



Silberschatz Şekil 6.2'den uyarlanmıştır

Çekirdekte sıradaki boş pid `next_available_pid` değişkeninde tutulur: **2615**. P0 ve P1 aynı anda `fork()` çağırır.

Adım 1 / 4

Aynı yarış, kullanıcı programında `count++` ifadesinin yazmaç düzeyinde iç içe geçmesiyle de oluşur.

Yarış Durumu: fork() ve pid



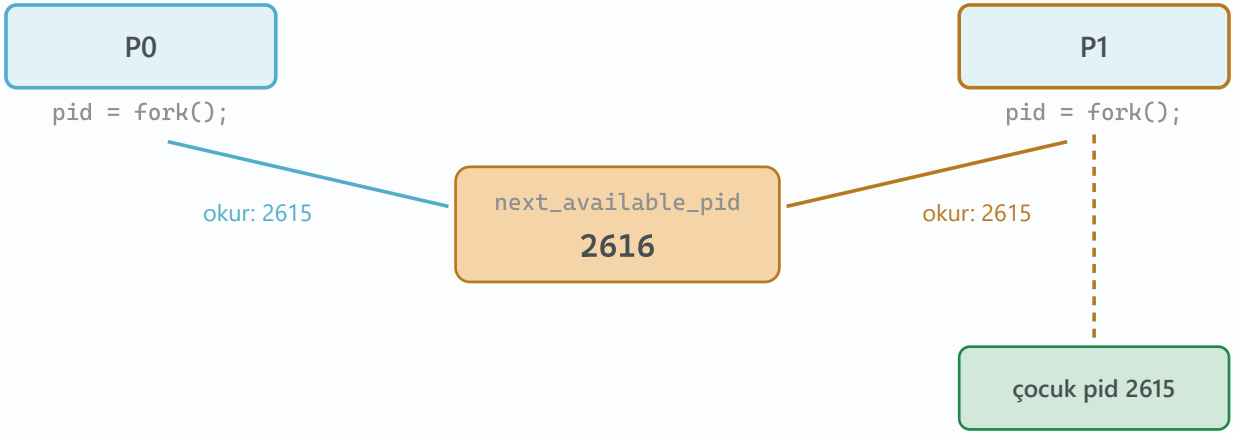
Silberschatz Şekil 6.2'den uyarlanmıştır

P0'ın çekirdek kodu değeri okur (2615) ama artırıp yazmadan önce kesilir.

Adım 2 / 4

Aynı yarış, kullanıcı programında `count++` ifadesinin yazmaç düzeyinde iç içe geçmesiyle de oluşur.

Yarış Durumu: fork() ve pid



Silberschatz Şekil 6.2'den uyarlanmıştır

P1 de 2615 okur, çocuğuna verir ve 2616 yazar.

Adım 3 / 4

Aynı yarış, kullanıcı programında `count++` ifadesinin yazmaç düzeyinde iç içe geçmesiyle de oluşur.

GİRİŞ

Yarış Durumu: fork() ve pid



Silberschatz Şekil 6.2'den uyarlanmıştır

P0 kaldığı yerden devam eder: çocuğuna **yine 2615** verir. Sonuç, işlemlerin hangi sırayla çalıştığına bağlıdır: **yarış durumu**.

Adım 4 / 4

Aynı yarış, kullanıcı programında `count++` ifadesinin yazmaç düzeyinde iç içe geçmesiyle de oluşur.

GİRİŞ

Yarış Durumu: Yazıcı Biriktirme Dizini

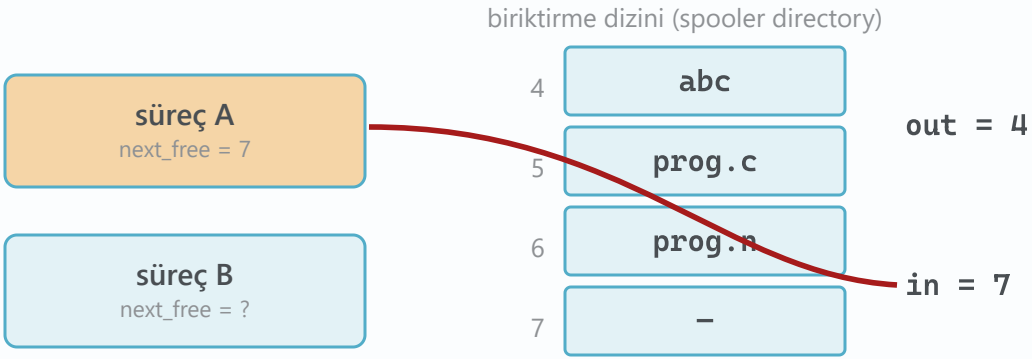


Yazdırılacak dosyalar sırayla yuvalara yazılır. `out` sıradaki yazdırılacak, `in` ilk boş yuva.

Adım 1 / 6

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

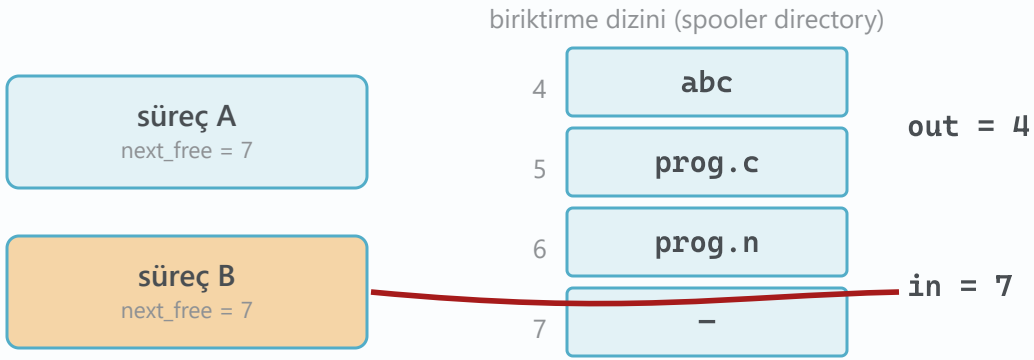
Yarış Durumu: Yazıcı Biriktirme Dizini



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

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

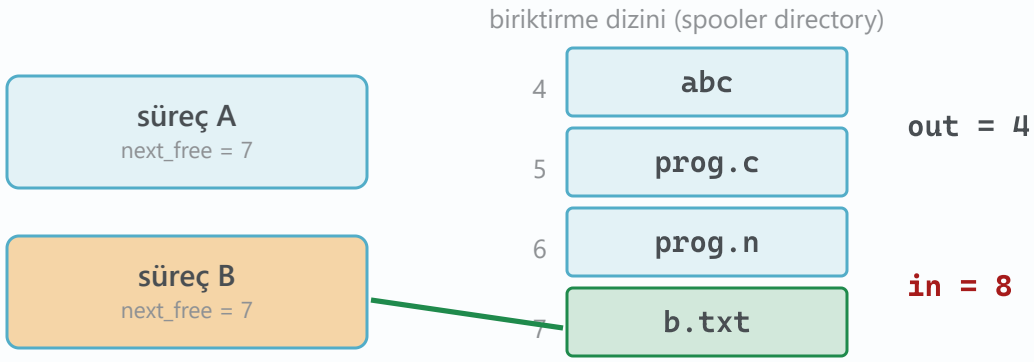
Yarış Durumu: Yazıcı Biriktirme Dizini



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

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

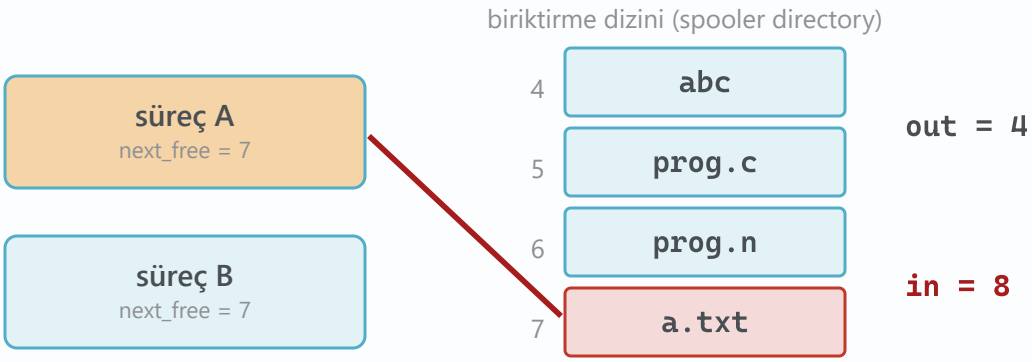
Yarış Durumu: Yazıcı Biriktirme Dizini



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

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`.

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**.

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

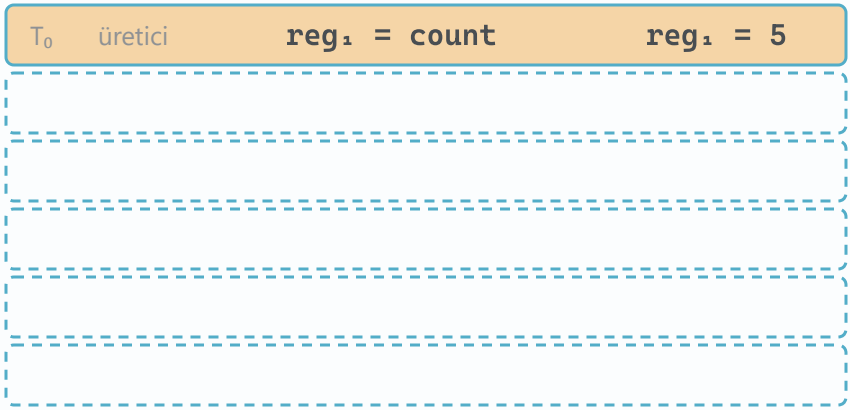
Tekrarlanamaz Yürütme (Non-reproducible)

```
/* 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.

Adım 1 / 6

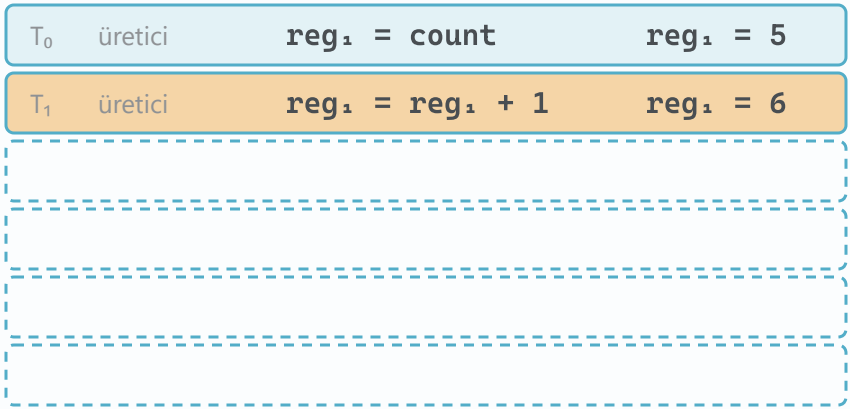
Tekrarlanamaz Yürütme (Non-reproducible)

```
/* 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:



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

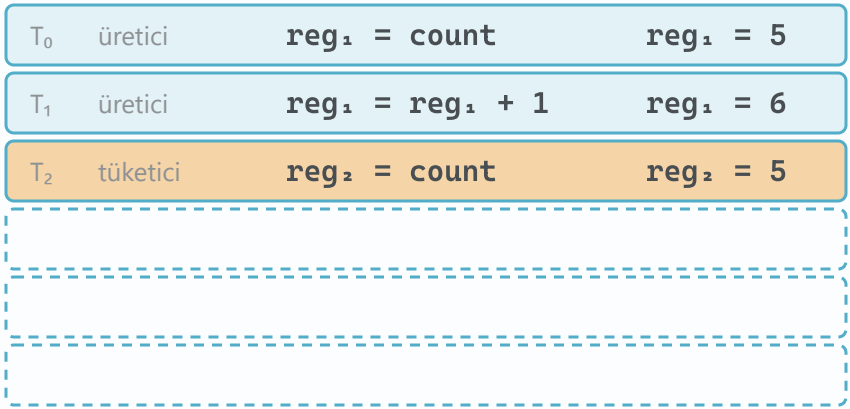
Tekrarlanamaz Yürütme (Non-reproducible)

```
/* 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:



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	reg_1 = count	reg_1 = 5
T_1	üretici	reg_1 = reg_1 + 1	reg_1 = 6
T_2	tüketici	reg_2 = count	reg_2 = 5
T_3	tüketici	reg_2 = reg_2 − 1	reg_2 = 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	reg_1 = count	reg_1 = 5
T_1	üretici	reg_1 = reg_1 + 1	reg_1 = 6
T_2	tüketici	reg_2 = count	reg_2 = 5
T_3	tüketici	reg_2 = reg_2 − 1	reg_2 = 4
T_4	üretici	count = reg_1	count = 6

Üretici belleğe yazar: `count = 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:

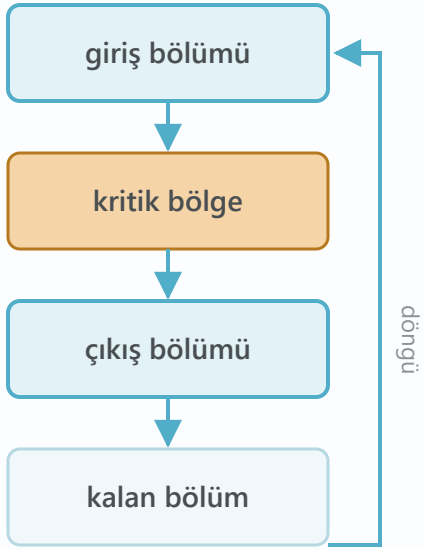
T_0	üretici	reg_1 = count	reg_1 = 5
T_1	üretici	reg_1 = reg_1 + 1	reg_1 = 6
T_2	tüketici	reg_2 = count	reg_2 = 5
T_3	tüketici	reg_2 = reg_2 − 1	reg_2 = 4
T_4	üretici	count = reg_1	count = 6
T_5	tüketici	count = reg_2	count = 4

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

Kritik Bölge Problemi

- n süreç `{P0, P1, …, Pn−1}`; her birinin paylaşılan veriye eriştiği bir **kritik bölgesi** vardır: global değişkene yazmak, tablo güncellemek, dosyaya yazmak…
- **Kural:** bir süreç kritik bölgedeyken başka hiçbir süreç *kendi* kritik bölgesinde olmamalıdır.
- Her süreç girmek için **giriş bölümünde** izin ister, çıkarken **çıkış bölümünde** bunu duyurur.
- **Yarış durumu:** sonucun yürütme sırasına bağlı olması. **Karşılıklı dışlama:** paylaşılan veriye aynı anda tek erişim.

```
while (true) {
    /* giriş bölümü  (entry)     */
        /* kritik bölge          */
    /* çıkış bölümü  (exit)      */
        /* kalan bölüm (remainder) */
}
```



izin iste → kullan → bırak

Bir Çözümün Üç Koşulu

1. Karşılıklı dışlama
P_i kritik bölgesindeyken başka hiçbir süreç kendi kritik bölgesinde yürütülemez.

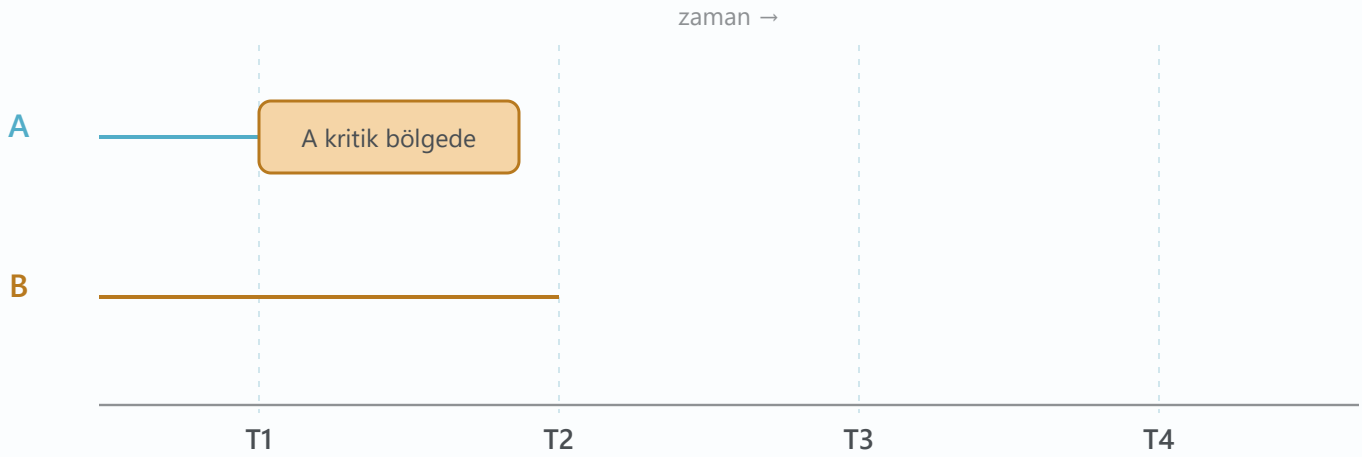
2. İlerleme
Kritik bölge boşsa ve girmek isteyenler varsa, kimin gireceğine yalnızca *kalan bölümde olmayanlar* karar verir ve bu karar süresiz ertelenemez.

3. Sınırlı bekleme
Bir süreç istek yaptıktan sonra, isteği karşılanana kadar diğerlerinin kaç kez girebileceğinin bir **üst sınırı** vardır (açlık olmaz).

Tanenbaum'un dört koşulu	**Karşılığı**
İki süreç aynı anda kritik bölgede olmamalı.	Karşılıklı dışlama
CPU hızı ve sayısı hakkında varsayım yapılmamalı.	Her süreç sıfırdan büyük hızla ilerler; göreli hız bilinmez
Kritik bölge dışındaki süreç diğerlerini engellememeli.	İlerleme
Hiçbir süreç sonsuza dek beklememeli.	Sınırlı bekleme

Silberschatz 10e §6.2; Tanenbaum MOS §2.3.2. Dışlama bir *güvenlik*, diğer ikisi *canlılık* özelliğidir.

Kritik Bölge Zaman Çizelgesi

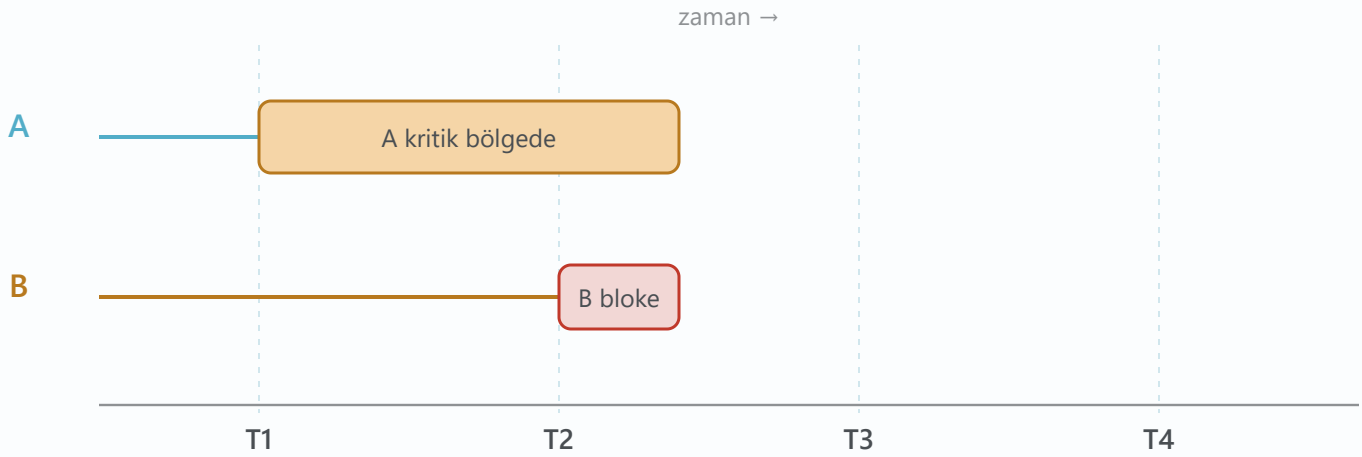


T1: A kritik bölgesine girer.

Adım 1 / 4

Tanenbaum MOS Şekil 2-22'den uyarlanmıştır.

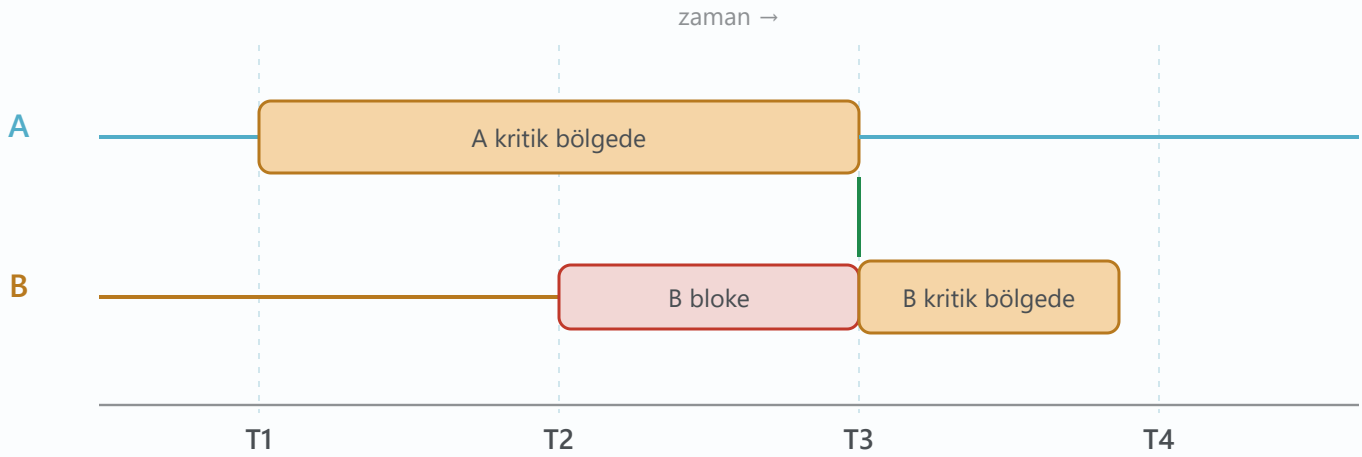
Kritik Bölge Zaman Çizelgesi



T2: B girmek ister; bölge dolu olduğu için **bloke olur** (ya da meşgul bekler).

Tanenbaum MOS Şekil 2-22'den uyarlanmıştır.

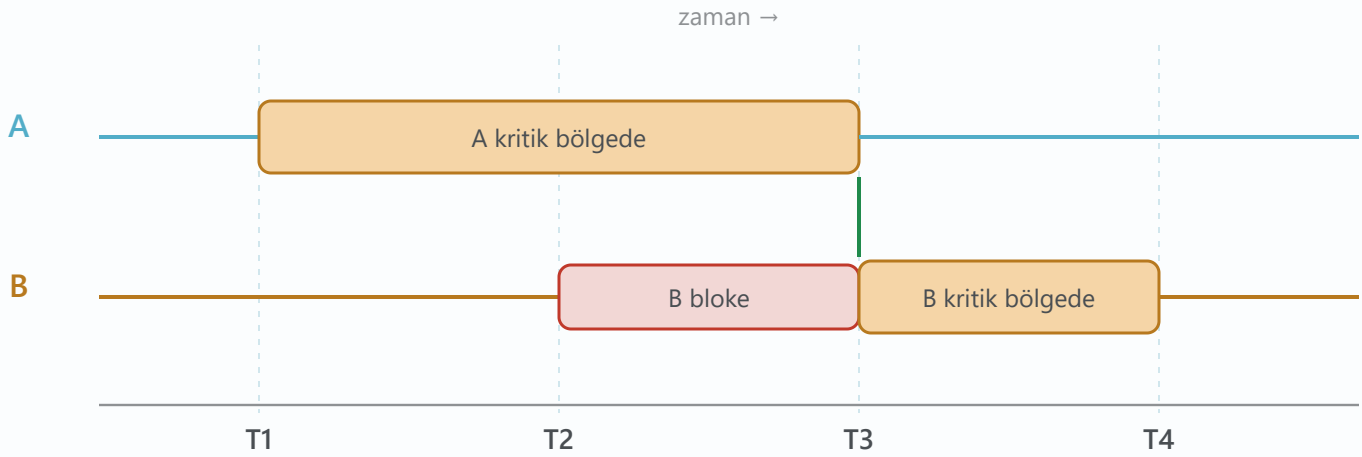
Kritik Bölge Zaman Çizelgesi



T3: A çıkar; bekleyen B hemen girer. Hiçbir anda iki süreç bölgede değildir.

Tanenbaum MOS Şekil 2-22'den uyarlanmıştır.

Kritik Bölge Zaman Çizelgesi

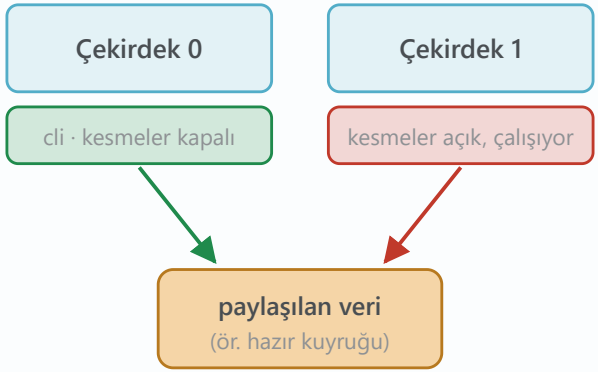


T4: B çıkar. İyi bir çözüm bu davranışı *hangi zamanlama olursa olsun* garanti etmelidir.

Tanenbaum MOS Şekil 2-22'den uyarlanmıştır.

Kesmeleri Devre Dışı Bırakmak

- Süreç kritik bölgeye girmeden kesmeleri kapatır, çıkınca açar: zamanlayıcı kesmesi gelmediği için **bağlam değiştirme olmaz**.
- **Sorunlar:**
  - Kesmeleri açmayı unutan (ya da sonsuz döngüye giren) süreç **sistemi kilitler**.
  - Bu sürede G/Ç ve zamanlayıcı kesmeleri bekler; uzun kritik bölgede sistem tepkisizleşir.
  - **Çok çekirdekte işe yaramaz:** diğer çekirdekler çalışmaya devam eder.
  - `cli` / `sti` ayrıcalıklı komutlardır; kullanıcı programı çalıştıramaz.
- Çekirdek içinde **kısa süreli** ve yerel kullanım yaygındır: Linux `local_irq_save()`, kesme işleyicisiyle paylaşılan veride `spin_lock_irqsave()`.



kesme kapatmak yalnız yerel çekirdeği durdurur

Kilit Değişkeni

```
int lock = 0;            /* paylaşılan */

void gir(void) {
  while (lock != 0)
    ;                    /* meşgul bekle */
  lock = 1;
}
void cik(void) {
  lock = 0;
}

/* her süreç:  gir(); kritik_bolge(); cik(); */
```

P0 — kritik bölge dışında

P1 — kritik bölge dışında

Paylaşılan `lock` 0: bölge boş.

Adım 1 / 6

DEĞİŞKENLER

`lock = 0`

Kilidin kendisi paylaşılan bir değişkendir; onu korumak için yine kilide ihtiyaç duyulur. Bu yüzden yazılım kilidi tek başına yetmez.

Kilit Değişkeni

```
1  int lock = 0;            /* paylaşılan */
2
3  void gir(void) {
▶    while (lock != 0)
5      ;                    /* meşgul bekle */
6    lock = 1;
7  }
8  void cik(void) {
9    lock = 0;
10 }
11
12 /* her süreç:  gir(); kritik_bolge(); cik(); */
```

P0 — lock != 0 ? → hayır

P1 — hazır

P0 `lock`'u okur: 0. Döngüden çıkar...

Adım 2 / 6

DEĞİŞKENLER

`lock = 0`

Kilidin kendisi paylaşılan bir değişkendir; onu korumak için yine kilide ihtiyaç duyulur. Bu yüzden yazılım kilidi tek başına yetmez.

Kilit Değişkeni

```
1  int lock = 0;            /* paylaşılan */
2
3  void gir(void) {
▶    while (lock != 0)
5      ;                    /* meşgul bekle */
6    lock = 1;
7  }
8  void cik(void) {
9    lock = 0;
10 }
11
12 /* her süreç:  gir(); kritik_bolge(); cik(); */
```

P0 kesildi (satır 6'dan önce)

P1 lock != 0 ? → hayır

...ama `lock = 1` yazmadan önce kesilir. P1 de 0 okur.

Adım 3 / 6

DEĞİŞKENLER

`lock = 0`

Kilidin kendisi paylaşılan bir değişkendir; onu korumak için yine kilide ihtiyaç duyulur. Bu yüzden yazılım kilidi tek başına yetmez.

Kilit Değişkeni

```
1  int lock = 0;            /* paylaşılan */
2
3  void gir(void) {
4    while (lock != 0)
5      ;                    /* meşgul bekle */
▶    lock = 1;
7  }
8  void cik(void) {
9    lock = 0;
10 }
11
12 /* her süreç:  gir(); kritik_bolge(); cik(); */
```

P0 — hazır

P1 — lock = 1 → girdi

P1 kilidi 1 yapar ve kritik bölgeye girer.

Adım 4 / 6

DEĞİŞKENLER

`lock = 1`

Kilidin kendisi paylaşılan bir değişkendir; onu korumak için yine kilide ihtiyaç duyulur. Bu yüzden yazılım kilidi tek başına yetmez.

Kilit Değişkeni

```
1  int lock = 0;            /* paylaşılan */
2
3  void gir(void) {
4    while (lock != 0)
5      ;                    /* meşgul bekle */
▶    lock = 1;
7  }
8  void cik(void) {
9    lock = 0;
10 }
11
12 /* her süreç:  gir(); kritik_bolge(); cik(); */
```

P0 — lock = 1 → girdi

P1 — kritik bölgede

P0 devam eder; zaten **test ettiği için** o da 1 yazar ve girer.

Adım 5 / 6

DEĞİŞKENLER

`lock = 1`

Kilidin kendisi paylaşılan bir değişkendir; onu korumak için yine kilide ihtiyaç duyulur. Bu yüzden yazılım kilidi tek başına yetmez.

Kilit Değişkeni

```
1  int lock = 0;            /* paylaşılan */
2
3  void gir(void) {
4    while (lock != 0)
5      ;                    /* meşgul bekle */
6    lock = 1;
7  }
8  void cik(void) {
9    lock = 0;
10 }
11
12 /* her süreç:  gir(); kritik_bolge(); cik(); */
```

P0 — kritik bölgede

P1 — kritik bölgede

iki süreç birden kritik bölgede!

Karşılıklı dışlama bozuldu: **test et** ve **ata** ayrı adımlar. Çözüm: ikisini tek atomik komutta yapmak (TSL).

Adım 6 / 6

DEĞİŞKENLER

`lock = 1`

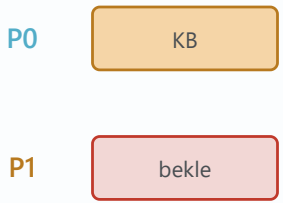
Kilidin kendisi paylaşılan bir değişkendir; onu korumak için yine kilide ihtiyaç duyulur. Bu yüzden yazılım kilidi tek başına yetmez.

Katı Sırayla Değişim

```
/* P0 */
while (true) {
  while (turn != 0) ;  /* bekle */
  kritik_bolge();
  turn = 1;
  kritik_disi();
}
```

```
/* P1 */
while (true) {
  while (turn != 1) ;  /* bekle */
  kritik_bolge();
  turn = 0;
  kritik_disi();
}
```

`turn = 0`



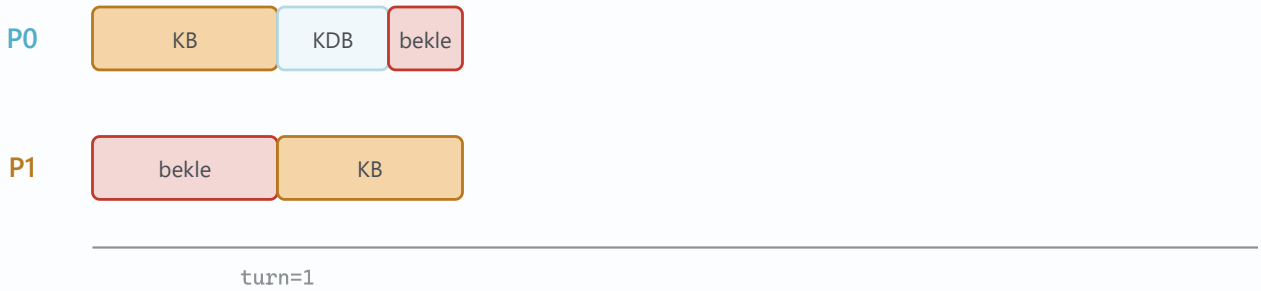
`turn = 0` : P0 girer, P1 meşgul bekler (**KB** kritik bölge, **KDB** kritik dışı bölüm).

Katı Sırayla Değişim

```
/* P0 */
while (true) {
  while (turn != 0) ;  /* bekle */
  kritik_bolge();
  turn = 1;
  kritik_disi();
}
```

```
/* P1 */
while (true) {
  while (turn != 1) ;  /* bekle */
  kritik_bolge();
  turn = 0;
  kritik_disi();
}
```

`turn = 1`

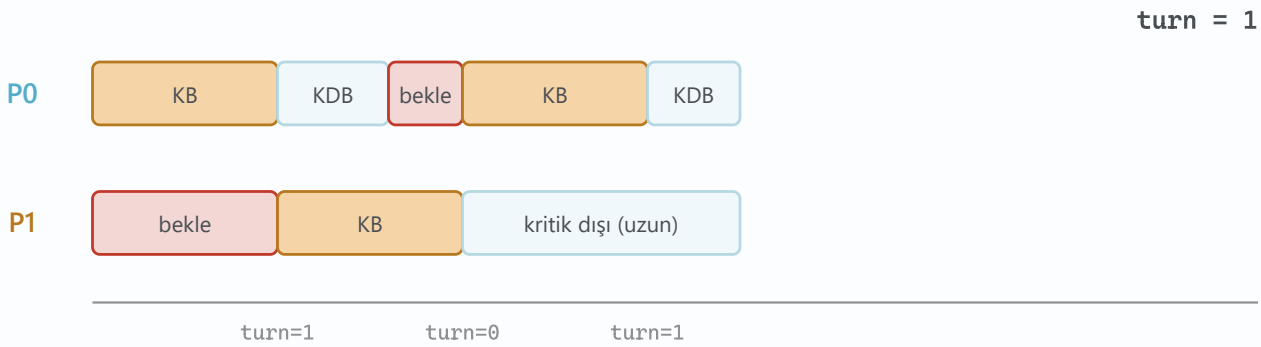


P0 çıkarken `turn = 1` yapar; P1 girer. Karşılıklı dışlama sağlanır.

Katı Sırayla Değişim

```
/* P0 */
while (true) {
  while (turn != 0) ;  /* bekle */
  kritik_bolge();
  turn = 1;
  kritik_disi();
}
```

```
/* P1 */
while (true) {
  while (turn != 1) ;  /* bekle */
  kritik_bolge();
  turn = 0;
  kritik_disi();
}
```

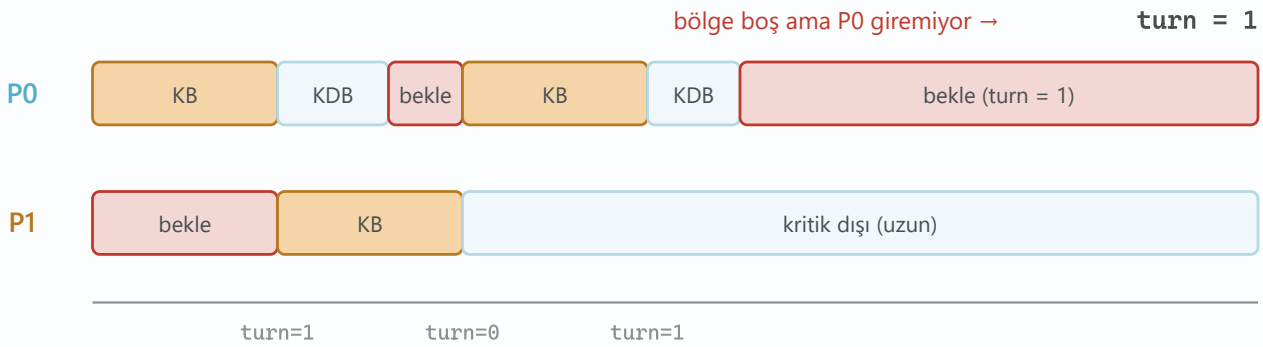


P1 çıkar (`turn = 0`) ve uzun bir kritik dışı işe başlar. P0 bir kez daha girip çıkar: `turn = 1`.

Katı Sırayla Değişim

```
/* P0 */
while (true) {
  while (turn != 0) ;  /* bekle */
  kritik_bolge();
  turn = 1;
  kritik_disi();
}
```

```
/* P1 */
while (true) {
  while (turn != 1) ;  /* bekle */
  kritik_bolge();
  turn = 0;
  kritik_disi();
}
```



P0 yeniden girmek ister ama sıra P1'de. Bölge boş olduğu hâlde **kritik bölge dışındaki P1, P0'ı engelliyor**: **ilerleme** koşulu bozulur.

Peterson Algoritması

```
/* paylaşılan */
bool flag[2] = { false, false };
int  turn;

/* P_i  (j = 1 - i) */
while (true) {
  flag[i] = true;     /* girmek istiyorum */
  turn = j;           /* önce sen buyur  */
  while (flag[j] && turn == j)
    ;                 /* meşgul bekle    */
  /* kritik bölge */
  flag[i] = false;
  /* kalan bölüm */
}
```

G. L. Peterson, 1981. Yalnız iki süreç içindir; n süreç için fırıncı (bakery) algoritması vardır.

Karşılıklı dışlama

İkisi de içerideyse `flag[0] = flag[1] = true` olmalı; ama `turn` aynı anda hem 0 hem 1 olamaz. `turn`'e son yazan bekler.

İlerleme

P_j ilgilenmiyorsa `flag[j] = false`: P_i hiç beklemeden girer.

Sınırlı bekleme

P_j çıkıp yeniden girmek isterse `turn = i` yapar; P_i en fazla **bir** giriş bekler.

Bedeli: **meşgul bekleme** CPU yakar. Daha önemlisi, kanıt *sıralı tutarlılık* varsayar; modern donanımda bu varsayım yoktur →

Peterson Algoritması: Kod İzleme

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



İki süreç de kalan bölümde; iki bayrak `false`. Bellek sıralı tutarlı varsayılıyor (her yazma hemen görünür).

Adım 1 / 13

DEĞİŞKENLER

`flag[0] = false` `flag[1] = false` `turn = ?` `P0 = kalan` `P1 = kalan`

Peterson Algoritması: Kod İzleme

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



P0 girmek ister: `flag[0] = true` .

Adım 2 / 13

DEĞİŞKENLER

`flag[0] = true` `flag[1] = false` `turn = ?` `P0 = giriyor` `P1 = kalan`

Peterson Algoritması: Kod İzleme

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



P1 de neredeyse aynı anda ister: `flag[1] = true`.

Adım 3 / 13

DEĞİŞKENLER

`flag[0] = true` `flag[1] = true` `turn = ?` `P0 = giriyor` `P1 = giriyor`

Peterson Algoritması: Kod İzleme

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



P0 nezaket gösterir: `turn = 1` (“önce sen”).

Adım 4 / 13

DEĞİŞKENLER

`flag[0] = true` `flag[1] = true`

`turn = 1` `P0 = giriyor` `P1 = giriyor`

Peterson Algoritması: Kod İzleme

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



P1 de `turn = 0` yazar. `turn`'e **son yazan P1** olduğu için beklemek ona düşer.

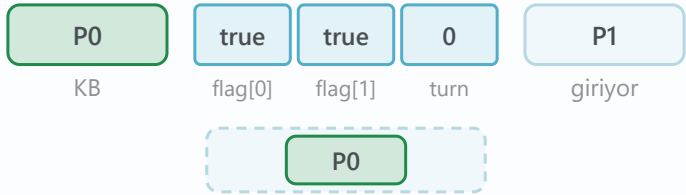
Adım 5 / 13

DEĞİŞKENLER

`flag[0] = true` `flag[1] = true` `turn = 0` `P0 = giriyor` `P1 = giriyor`

Peterson Algoritması: Kod İzleme

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



P0: `flag[1] && turn == 1` → turn 0 olduğu için yanlış; döngüden çıkar, **kritik bölgeye girer**.

Adım 6 / 13

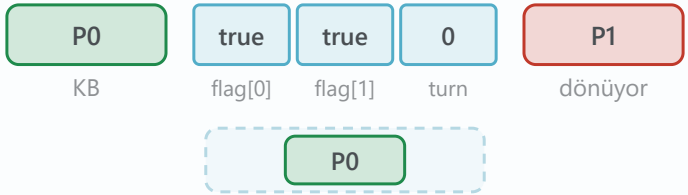
DEĞİŞKENLER

`flag[0] = true` `flag[1] = true`

`turn = 0` `P0 = KB` `P1 = giriyor`

Peterson Algoritması: Kod İzleme

```
1  bool flag[2] = { false, false };
2  int turn;
3  void gir(int i) {
4    int j = 1 - i;
5    flag[i] = true;    /* istiyorum */
6    turn = j;          /* sen buyur */
▶    while (flag[j] && turn == j)
▶      ;                /* meşgul bekle */
9  }
10 void cik(int i) {
11   flag[i] = false;
12 }
```



P1: `flag[0] && turn == 0` doğru → meşgul bekler. Karşılıklı dışlama korunur.

Adım 7 / 13

DEĞİŞKENLER

`flag[0] = true` `flag[1] = true` `turn = 0` `P0 = KB` `P1 = dönüyor`

Peterson Algoritması: Kod İzleme

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



P0 çıkar: `flag[0] = false`.

Adım 8 / 13

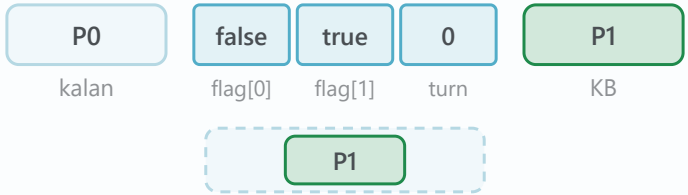
DEĞİŞKENLER

`flag[0] = false` `flag[1] = true`

`turn = 0` `P0 = kalan` `P1 = dönüyor`

Peterson Algoritması: Kod İzleme

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



P1'in koşulu artık yanlış (`flag[0]` false) → P1 girer. İlerleme sağlanır.

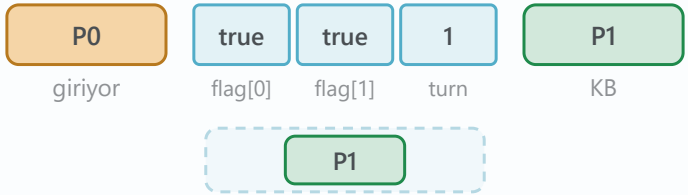
Adım 9 / 13

DEĞİŞKENLER

`flag[0] = false` `flag[1] = true` `turn = 0` `P0 = kalan` `P1 = KB`

Peterson Algoritması: Kod İzleme

```
1  bool flag[2] = { false, false };
2  int turn;
3  void gir(int i) {
4    int j = 1 - i;
▶    flag[i] = true;    /* istiyorum */
▶    turn = j;          /* sen buyur */
7    while (flag[j] && turn == j)
8      ;                /* meşgul bekle */
9  }
10 void cik(int i) {
11   flag[i] = false;
12 }
```



P0 yeniden ister: `flag[0] = true`, `turn = 1`.

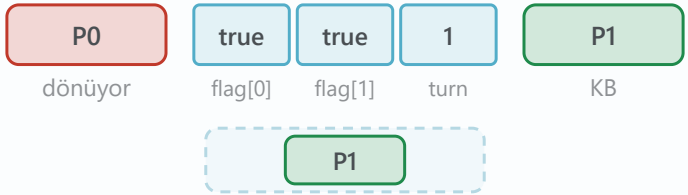
Adım 10 / 13

DEĞİŞKENLER

`flag[0] = true` `flag[1] = true` `turn = 1` `P0 = giriyor` `P1 = KB`

Peterson Algoritması: Kod İzleme

```
1  bool flag[2] = { false, false };
2  int turn;
3  void gir(int i) {
4    int j = 1 - i;
5    flag[i] = true;    /* istiyorum */
6    turn = j;          /* sen buyur */
▶    while (flag[j] && turn == j)
▶      ;                /* meşgul bekle */
9  }
10 void cik(int i) {
11   flag[i] = false;
12 }
```



P0: `flag[1] && turn == 1` doğru → bekler. Şimdi turn'e son yazan P0.

Adım 11 / 13

DEĞİŞKENLER

`flag[0] = true` `flag[1] = true` `turn = 1` `P0 = dönüyor` `P1 = KB`

Peterson Algoritması: Kod İzleme

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



P1 çıkar: `flag[1] = false`.

Adım 12 / 13

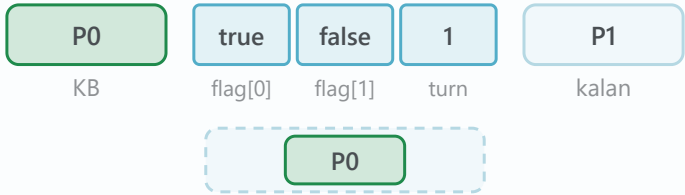
DEĞİŞKENLER

`flag[0] = true` `flag[1] = false`

`turn = 1` `P0 = dönüyor` `P1 = kalan`

Peterson Algoritması: Kod İzleme

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



P0'ın koşulu yanlış olur ve girer. P1 yeniden isteseydi `turn = 0` yazıp bekleyecekti: P0 en fazla **bir** giriş bekler.

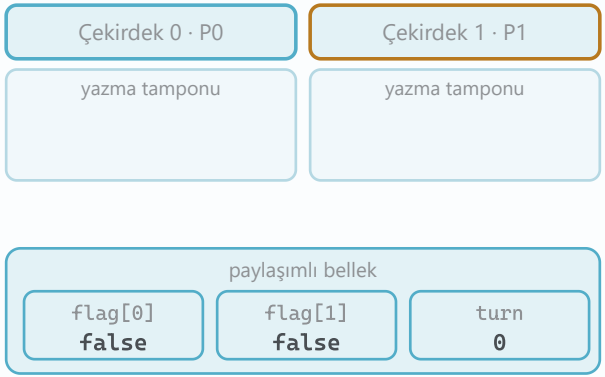
Adım 13 / 13

DEĞİŞKENLER

`flag[0] = true` `flag[1] = false` `turn = 1` `P0 = KB` `P1 = kalan`

Peterson Modern CPU'da Neden Bozulur?

```
1 /* P_i  (i = 0 ya da 1, j = 1 - i) */
2 flag[i] = true;
3 turn = j;
4 while (flag[j] && turn == j)
5   ;
6 /* kritik bölge */
7 flag[i] = false;
```



Başlangıç: bellekte iki bayrak `false`. Her çekirdeğin yazmaları önce kendi **yazma tamponuna** gider.

Adım 1 / 6

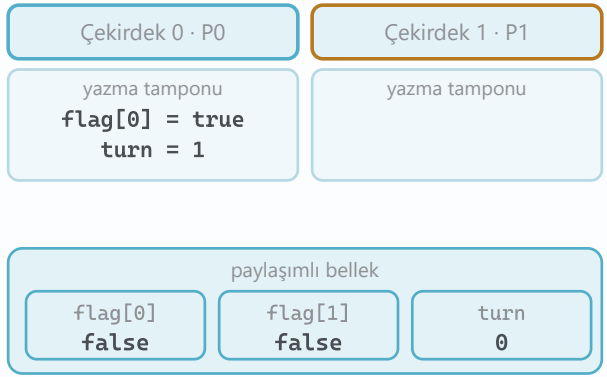
DEĞİŞKENLER

—

Derleyici de bu yüklemeyi öne alabilir. C11'de `_Atomic` + varsayılan `memory_order_seq_cst`, Java'da `volatile` alanlar bu sıralamayı geri getirir.

Peterson Modern CPU'da Neden Bozulur?

```
1 /* P_i  (i = 0 ya da 1, j = 1 - i) */
▶ flag[i] = true;
▶ turn = j;
4 while (flag[j] && turn == j)
5   ;
6 /* kritik bölge */
7 flag[i] = false;
```



P0 iki yazmasını yapar; ikisi de **tamponda bekler**, bellekte henüz görünmez.

Adım 2 / 6

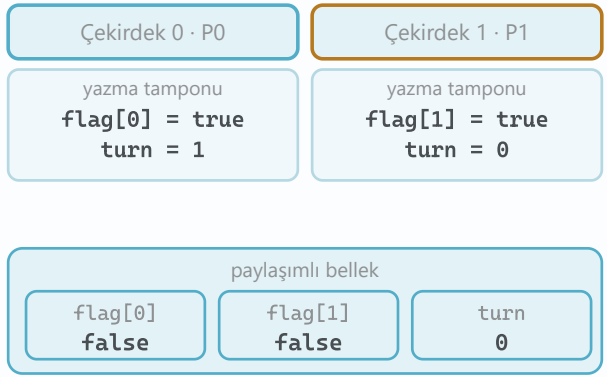
DEĞİŞKENLER

—

Derleyici de bu yüklemeyi öne alabilir. C11'de `_Atomic` + varsayılan `memory_order_seq_cst`, Java'da `volatile` alanlar bu sıralamayı geri getirir.

Peterson Modern CPU'da Neden Bozulur?

```
1 /* P_i  (i = 0 ya da 1, j = 1 - i) */
▶ flag[i] = true;
▶ turn = j;
4 while (flag[j] && turn == j)
5   ;
6 /* kritik bölge */
7 flag[i] = false;
```



P1 de aynısını yapar. x86 bile (TSO) yazmayı sonraki okumadan **sonraya** bırakabilir.

Adım 3 / 6

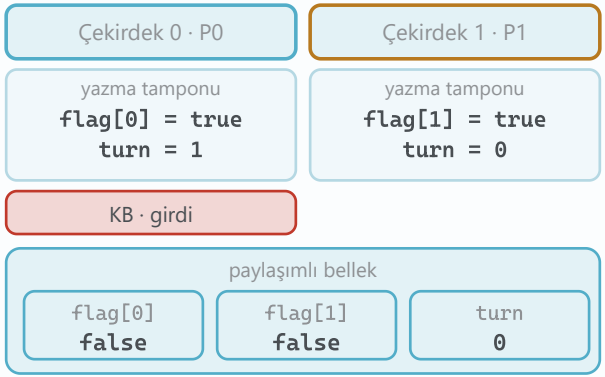
DEĞİŞKENLER

—

Derleyici de bu yüklemeyi öne alabilir. C11'de `_Atomic` + varsayılan `memory_order_seq_cst`, Java'da `volatile` alanlar bu sıralamayı geri getirir.

Peterson Modern CPU'da Neden Bozulur?

```
1 /* P_i  (i = 0 ya da 1, j = 1 - i) */
2 flag[i] = true;
3 turn = j;
▶ while (flag[j] && turn == j)
5   ;
6 /* kritik bölge */
7 flag[i] = false;
```



P0 `flag[1]`'i bellekten okur: **false** → döngüden çıkar.

Adım 4 / 6

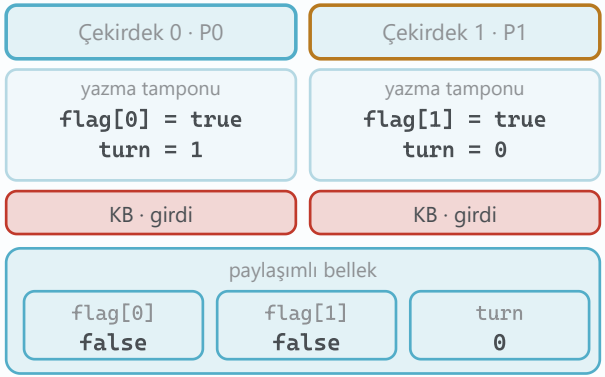
DEĞİŞKENLER

`P0: flag[1] = false`

Derleyici de bu yüklemeyi öne alabilir. C11'de `_Atomic` + varsayılan `memory_order_seq_cst`, Java'da `volatile` alanlar bu sıralamayı geri getirir.

Peterson Modern CPU'da Neden Bozulur?

```
1 /* P_i  (i = 0 ya da 1, j = 1 - i) */
2 flag[i] = true;
3 turn = j;
▶ while (flag[j] && turn == j)
5   ;
6 /* kritik bölge */
7 flag[i] = false;
```



P1 de `flag[0]`'ı **false** okur ve girer: **ikisi birden kritik bölgede!**

Adım 5 / 6

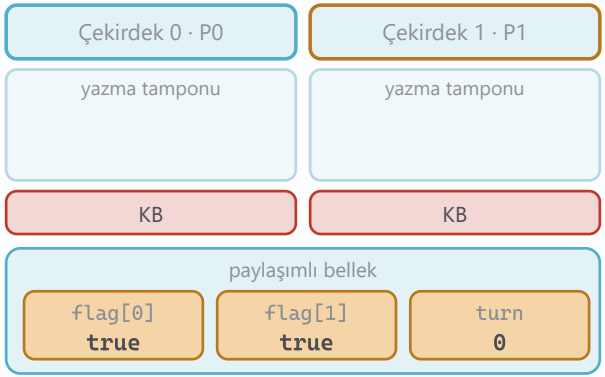
DEĞİŞKENLER

`P0: flag[1] = false` `P1: flag[0] = false`

Derleyici de bu yüklemeyi öne alabilir. C11'de `_Atomic` + varsayılan `memory_order_seq_cst`, Java'da `volatile` alanlar bu sıralamayı geri getirir.

Peterson Modern CPU'da Neden Bozulur?

```
1 /* P_i  (i = 0 ya da 1, j = 1 - i) */
2 flag[i] = true;
3 turn = j;
4 while (flag[j] && turn == j)
5   ;
▶ /* kritik bölge */
7 flag[i] = false;
```



Tamponlar sonra boşalır ama iş işten geçmiştir. Çözüm: yazma ile okuma arasına **tam bellek bariyeri** ya da sıralı tutarlı atomikler.

Adım 6 / 6

DEĞİŞKENLER

—

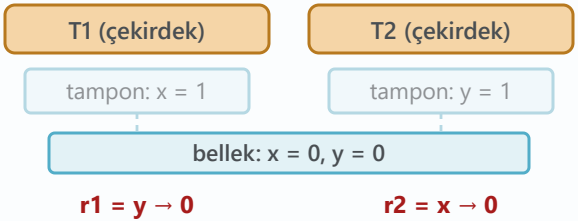
Derleyici de bu yüklemeyi öne alabilir. C11'de `_Atomic` + varsayılan `memory_order_seq_cst`, Java'da `volatile` alanlar bu sıralamayı geri getirir.

Bellek Modelleri ve Yeniden Sıralama

```
/* x = y = 0 */
/* T1 */          /* T2 */
x = 1;            y = 1;
r1 = y;           r2 = x;

/* r1 == 0 && r2 == 0 ?  */
```

Sezgiye göre en az biri 1 görmeli. Ama x86, ARM ve RISC-V'de **r1 = r2 = 0** gözlenebilir (store-buffer testi).



Model	İzin verilen yeniden sıralama
Sıralı tutarlılık (SC)	Hiçbiri; tüm çekirdekler tek bir ortak sıra görür (Lamport 1979).
x86-64 (TSO)	Yalnız *yazma → sonraki okuma*; yazmalar kendi aralarında sıralı.
ARMv8, RISC-V (zayıf)	Okuma ve yazmalar bağımlılık yoksa birbirini geçebilir.
Derleyici	Tek iş parçacığı için eşdeğerse her erişimi taşıyabilir, kaydedebilir ya da silebilir.

C'deki `volatile` iş parçacıkları için **ne atomiklik ne de sıralama** sağlar. Doğru araçlar: C11 `<stdatomic.h>`, Java `volatile` / `java.util.concurrent`, kilitler.

Bellek Bariyerleri

İş parçacığı 1 (okuyan)

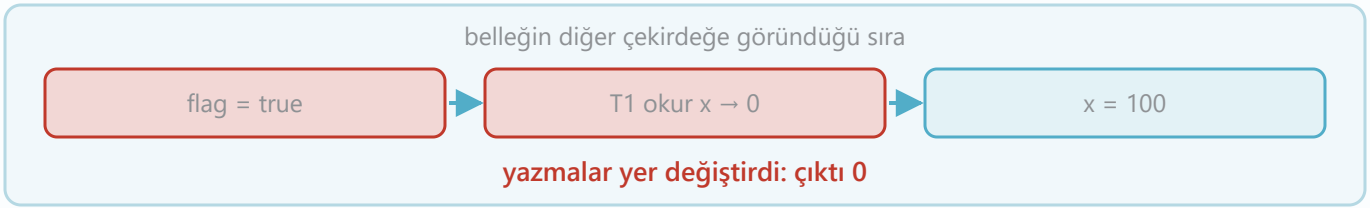
```
while (!flag) ;

print(x);
```

İş parçacığı 2 (yazan)

```
x = 100;

flag = true;
```



Bariyer yok: zayıf modelde T2'nin yazmaları başka sırayla görünebilir; T1 `flag`'i true görür ama `x` hâlâ 0'dır.

Silberschatz 10e §6.4.1'deki örnek. Kilit ve semafor gerçeklemeleri bu bariyerleri zaten içerir: kilitle korunan kodda ayrıca bariyer yazmak gerekmez.

Bellek Bariyerleri

İş parçacığı 1 (okuyan)

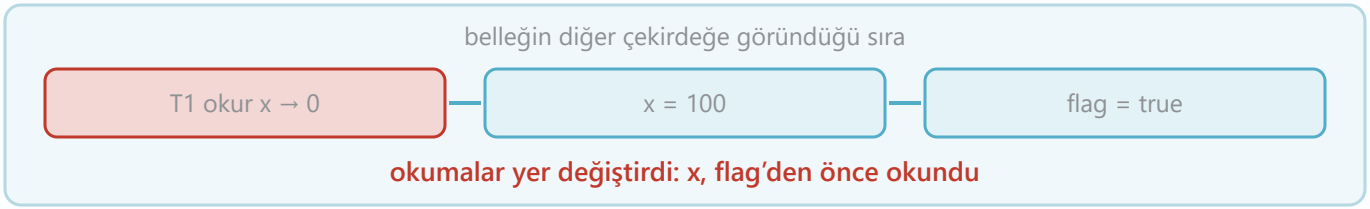
```
while (!flag) ;

print(x);
```

İş parçacığı 2 (yazan)

```
x = 100;

flag = true;
```



Ya da T1'in kendisi `x`'i `flag`'den **önce** okuyabilir (okuma-okuma yeniden sıralaması).

Silberschatz 10e §6.4.1'deki örnek. Kilit ve semafor gerçeklemeleri bu bariyerleri zaten içerir: kilitle korunan kodda ayrıca bariyer yazmak gerekmez.

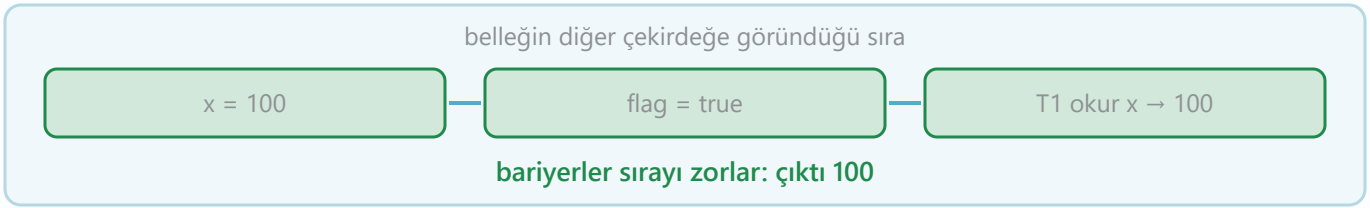
Bellek Bariyerleri

İş parçacığı 1 (okuyan)

```
while (!flag) ;
memory_barrier();
print(x);
```

İş parçacığı 2 (yazan)

```
x = 100;
memory_barrier();
flag = true;
```



Bellek bariyeri (fence): öncesindeki tüm okuma/yazmalar tamamlanmadan sonrakiler başlamaz. T2: `x` önce; T1: `flag` önce.

Silberschatz 10e §6.4.1'deki örnek. Kilit ve semafor gerçeklemeleri bu bariyerleri zaten içerir: kilitle korunan kodda ayrıca bariyer yazmak gerekmez.

Bellek Bariyerleri

İş parçacığı 1 (okuyan)

```
while (!flag) ;
memory_barrier();
print(x);
```

İş parçacığı 2 (yazan)

```
x = 100;
memory_barrier();
flag = true;
```



Gerçek kod: x86 `mfence`, ARM `dmb`, Linux `smp_mb()`, C11 `atomic_thread_fence`; çoğu zaman release/acquire atomikleri yeterlidir.

Silberschatz 10e §6.4.1'deki örnek. Kilit ve semafor gerçeklemeleri bu bariyerleri zaten içerir: kilitle korunan kodda ayrıca bariyer yazmak gerekmez.

Peterson Java'da: Doğru Gerçekleme

```
public class Peterson {
  static volatile boolean flag0, flag1;
  static volatile int turn;
  static int sayac = 0;

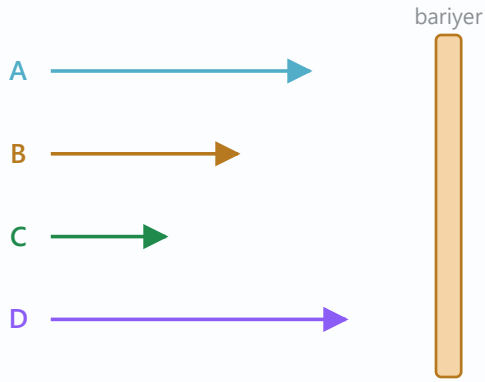
  static void kilitle(int i) {
    if (i == 0) { flag0 = true; turn = 1;
      while (flag1 && turn == 1) Thread.onSpinWait(); }
    else        { flag1 = true; turn = 0;
      while (flag0 && turn == 0) Thread.onSpinWait(); }
  }
  static void birak(int i) {
    if (i == 0) flag0 = false; else flag1 = false;
  }
  /* main: i = 0 ve i = 1 için iki iş parçacığı,
     her biri 100_000 kez: kilitle(i); sayac++; birak(i);
     ikisi join edilir, sayac yazdırılır */
}
```

- Sık yapılan üç hata:
  - `volatile boolean[] flag` yalnız **dizi referansını** volatile yapar, elemanları değil.
  - `getId() % N` iki iş parçacığına **aynı** `i` 'yi verebilir.
  - Kritik bölgedeki `print` ölçümü bozar.
- Java'da `volatile` okuma/yazmaları **sıralı tutarlıdır**; ayrı volatile alanlarla Peterson doğru çalışır.
- `Thread.onSpinWait()` (Java 9+) CPU'ya "meşgul bekliyorum" ipucu verir (x86 `pause`).

ÇIKTI (JDK 21, ÜÇ ÇALIŞTIRMA)

```
sayac = 200000
sayac = 200000
sayac = 200000
```

Senkronizasyon Bariyeri



her süreç kendi fazını hesaplıyor

Fazlara bölünmüş paralel hesap (ör. ızgara simülasyonu): her süreç bir dilimi hesaplar.

Adım 1 / 3

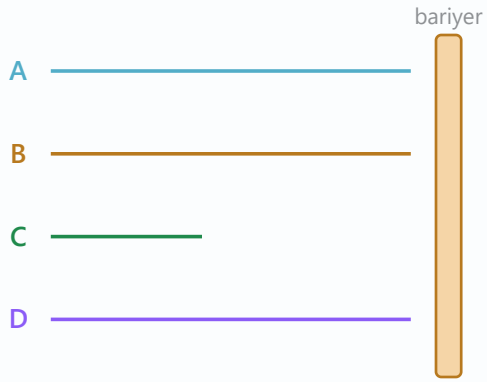
```
var b = new CyclicBarrier(3,
  () -> System.out.println(
          "faz " + faz[0]++ + " tamam"));
/* 3 iş parçacığı, her biri:
   for (f = 0; f < 2; f++) {
     hesapla(f); b.await();
   } */
```

ÇIKTI (JDK 21)

```
faz 0 tamam
faz 1 tamam
```

"Bariyer" iki farklı kavramın adıdır: **senkronizasyon bariyeri** bir süreç grubunu buluşturur (`pthread_barrier_wait` , `CyclicBarrier`); **bellek bariyeri** tek bir çekirdeğin bellek erişim sırasını zorlar.

Senkronizasyon Bariyeri



A, B, D bariyerde bloke · C çalışıyor

Bitiren süreç `barrier_wait()` çağırır ve **bloke olur**; en yavaş süreç (C) beklenir.

Adım 2 / 3

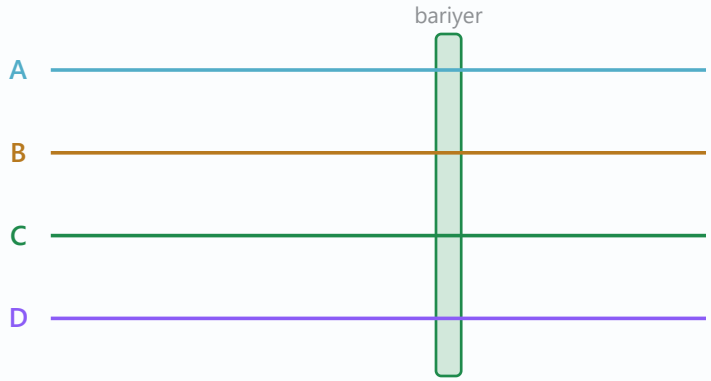
```
var b = new CyclicBarrier(3,
  () -> System.out.println(
          "faz " + faz[0]++ + " tamam"));
/* 3 iş parçacığı, her biri:
   for (f = 0; f < 2; f++) {
     hesapla(f); b.await();
   } */
```

ÇIKTI (JDK 21)

```
faz 0 tamam
faz 1 tamam
```

"Bariyer" iki farklı kavramın adıdır: **senkronizasyon bariyeri** bir süreç grubunu buluşturur (`pthread_barrier_wait` , `CyclicBarrier`); **bellek bariyeri** tek bir çekirdeğin bellek erişim sırasını zorlar.

Senkronizasyon Bariyeri



son süreç geldi: hepsi sonraki faza

N’inci süreç gelince bariyer açılır; hepsi bir sonraki faza birlikte geçer.


```
var b = new CyclicBarrier(3,
  () -> System.out.println(
          "faz " + faz[0]++ + " tamam"));
/* 3 iş parçacığı, her biri:
   for (f = 0; f < 2; f++) {
     hesapla(f); b.await();
   } */
```

ÇIKTI (JDK 21)

```
faz 0 tamam
faz 1 tamam
```

“Bariyer” iki farklı kavramın adıdır: **senkronizasyon bariyeri** bir süreç grubunu buluşturur (`pthread_barrier_wait` , `CyclicBarrier`); **bellek bariyeri** tek bir çekirdeğin bellek erişim sırasını zorlar.

DONANIM DESTEĞİ

Atomik Donanım Komutları

```
/* tümü TEK atomik komuttur */
bool test_and_set(bool *hedef) {
  bool eski = *hedef;
  *hedef = true;
  return eski;
}

/* kullanım: basit spinlock */
bool lock = false;
while (test_and_set(&lock))
  ;            /* meşgul bekle */
/* kritik bölge */
lock = false;
```

- Yazılım çözümlerinin sorunu: *oku* ve *yaz* ayrı adımlar. Donanım, bir bellek sözcüğünü **kesintisiz** test edip değiştiren komutlar sunar.
- Atomiklik çok çekirdekte de geçerlidir: önbellek tutarlılığı protokolü o satırı komut boyunca tek çekirdeğe ayırır (eskiden bellek veri yolu kilitlenirdi).

Mimari	Atomik okuma-değiştirme-yazma
x86-64	`XCHG`, `LOCK CMPXCHG`, `LOCK XADD`, `LOCK BTS`
ARMv8	`LDXR/STXR` (LL/SC); v8.1 LSE: `CAS`, `LDADD`, `SWP`
RISC-V “A”	`LR/SC`, `AMOSWAP`, `AMOADD`
Tanenbaum’un sanal CPU’su	`TSL REGISTER,LOCK`

TSL ile Kritik Bölge

```
1 enter_region:
2   TSL REGISTER,LOCK   | R = LOCK; LOCK = 1
3   CMP REGISTER,#0     | LOCK 0 mıydı?
4   JNE enter_region    | değilse: tekrar dene
5   RET                 | kritik bölgeye girildi
6
7 leave_region:
8   MOVE LOCK,#0        | LOCK = 0
9   RET
```



Kilit boş: `LOCK = 0`.

Adım 1 / 6

DEĞİŞKENLER

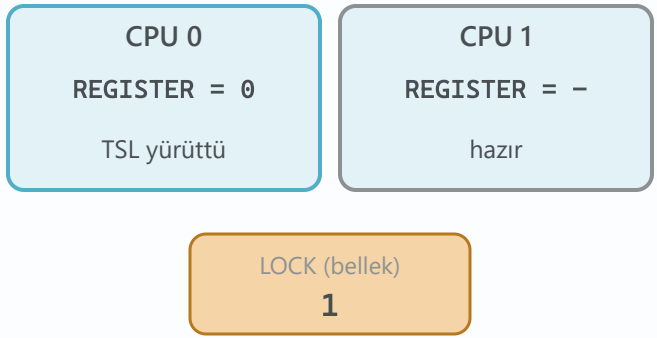
`LOCK = 0`

Tanenbaum MOS Şekil 2-25. TSL çok çekirdekte de çalışır; kesme kapatmanın aksine diğer CPU'ların o sözcüğe erişimini de engeller.

DONANIM DESTEĞİ · KODU ADIM ADIM ÇALIŞTIR

TSL ile Kritik Bölge

```
1 enter_region:
▶   TSL REGISTER,LOCK   | R = LOCK; LOCK = 1
3   CMP REGISTER,#0     | LOCK 0 mıydı?
4   JNE enter_region    | değilse: tekrar dene
5   RET                 | kritik bölgeye girildi
6
7 leave_region:
8   MOVE LOCK,#0        | LOCK = 0
9   RET
```



CPU 0 `TSL` : eski değer (0) yazmaca, LOCK'a 1 — **tek bölünmez adımda**.

Adım 2 / 6

DEĞİŞKENLER

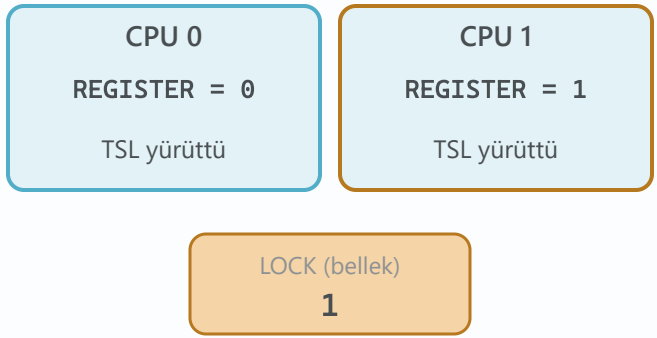
`LOCK = 1` `R0 = 0`

Tanenbaum MOS Şekil 2-25. TSL çok çekirdekte de çalışır; kesme kapatmanın aksine diğer CPU'ların o sözcüğe erişimini de engeller.

DONANIM DESTEĞİ · KODU ADIM ADIM ÇALIŞTIR

TSL ile Kritik Bölge

```
1 enter_region:
▶   TSL REGISTER,LOCK   | R = LOCK; LOCK = 1
3   CMP REGISTER,#0     | LOCK 0 mıydı?
4   JNE enter_region    | değilse: tekrar dene
5   RET                 | kritik bölgeye girildi
6
7 leave_region:
8   MOVE LOCK,#0        | LOCK = 0
9   RET
```



CPU 1 de `TSL` yürütür: yazmaca **1** gelir; LOCK zaten 1'dir.

Adım 3 / 6

DEĞİŞKENLER

`LOCK = 1` `R0 = 0` `R1 = 1`

Tanenbaum MOS Şekil 2-25. TSL çok çekirdekte de çalışır; kesme kapatmanın aksine diğer CPU'ların o sözcüğe erişimini de engeller.

DONANIM DESTEĞİ · KODU ADIM ADIM ÇALIŞTIR

TSL ile Kritik Bölge

```
1 enter_region:
2   TSL REGISTER,LOCK   | R = LOCK; LOCK = 1
▶   CMP REGISTER,#0     | LOCK 0 mıydı?
▶   JNE enter_region    | değilse: tekrar dene
5   RET                 | kritik bölgeye girildi
6
7 leave_region:
8   MOVE LOCK,#0        | LOCK = 0
9   RET
```

CPU 0

REGISTER = 0

CMP: 0 → RET, KB'de

CPU 1

REGISTER = 1

JNE → döngü (spin)

LOCK (bellek)

1

CPU 0 sıfır gördü: kritik bölgede. CPU 1 sıfır olmayan değer gördü: **meşgul bekler**.

Adım 4 / 6

DEĞİŞKENLER

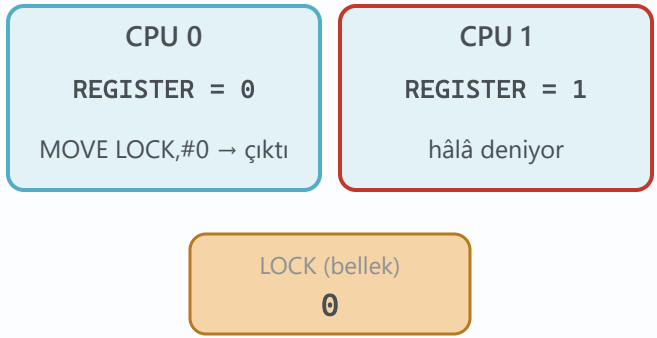
`LOCK = 1` `R0 = 0` `R1 = 1`

Tanenbaum MOS Şekil 2-25. TSL çok çekirdekte de çalışır; kesme kapatmanın aksine diğer CPU'ların o sözcüğe erişimini de engeller.

DONANIM DESTEĞİ · KODU ADIM ADIM ÇALIŞTIR

TSL ile Kritik Bölge

```
1 enter_region:
2   TSL REGISTER,LOCK   | R = LOCK; LOCK = 1
3   CMP REGISTER,#0     | LOCK 0 mıydı?
4   JNE enter_region    | değilse: tekrar dene
5   RET                 | kritik bölgeye girildi
6
7 leave_region:
▶   MOVE LOCK,#0        | LOCK = 0
9   RET
```



CPU 0 çıkarken sıradan bir yazmayla LOCK'u 0 yapar.

Adım 5 / 6

DEĞİŞKENLER

LOCK = 0 | R0 = 0 | R1 = 1

Tanenbaum MOS Şekil 2-25. TSL çok çekirdekte de çalışır; kesme kapatmanın aksine diğer CPU'ların o sözcüğe erişimini de engeller.

DONANIM DESTEĞİ · KODU ADIM ADIM ÇALIŞTIR

TSL ile Kritik Bölge

```
1 enter_region:
  TSL REGISTER,LOCK   | R = LOCK; LOCK = 1
3   CMP REGISTER,#0     | LOCK 0 mıydı?
4   JNE enter_region    | değilse: tekrar dene
5   RET                 | kritik bölgeye girildi
6
7 leave_region:
8   MOVE LOCK,#0        | LOCK = 0
9   RET
```

CPU 0

REGISTER = 0

kalan bölüm

CPU 1

REGISTER = 0

TSL → 0 gördü, giriyor

LOCK (bellek)

1

CPU 1'in sonraki TSL'i 0 okur ve kilidi alır. İki CPU hiçbir an birlikte içeride değildir.

Adım 6 / 6

DEĞİŞKENLER

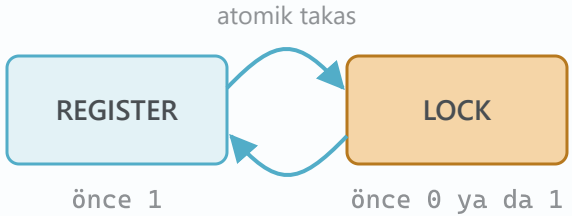
`LOCK = 1` `R0 = 0` `R1 = 0`

Tanenbaum MOS Şekil 2-25. TSL çok çekirdekte de çalışır; kesme kapatmanın aksine diğer CPU'ların o sözcüğe erişimini de engeller.

XCHG Komutu

```
enter_region:
  MOVE REGISTER,#1     | yazmaca 1 koy
  XCHG REGISTER,LOCK   | yazmaç ile LOCK'u takas et
  CMP REGISTER,#0      | LOCK 0 mıydı?
  JNE enter_region     | değilse: döngü
  RET                  | kritik bölgeye girildi
leave_region:
  MOVE LOCK,#0         | LOCK = 0
  RET
```

- İki işlenenin içeriğini **tek adımda** değiştirir; etkisi TSL ile aynıdır.
- Intel x86'da kilitleme için kullanılan komuttur: bellek işlenenli `XCHG` örtük olarak `LOCK` önekli, yani atomiktir.
- C11 karşılığı: `atomic_exchange(&lock, 1)`.



sonra: REGISTER = eski LOCK

LOCK = 1 (her durumda)

eski LOCK 0 ise kilit alındı

compare_and_swap ile Kilit

```
1  int compare_and_swap(int *v, int bekl, int yeni) {
2    int eski = *v;          /* üç satır birlikte */
3    if (eski == bekl)       /* TEK atomik komut  */
4      *v = yeni;
5    return eski;
6  }
7
▶ int lock = 0;
9  void gir(void) {
10   while (compare_and_swap(&lock, 0, 1) != 0)
11     ;                     /* meşgul bekle */
12 }
13 void cik(void) { lock = 0; }
```

T0 dışarıda

T1 dışarıda

Kilit boş.

Adım 1 / 6

DEĞİŞKENLER

`lock = 0`

Silberschatz 10e §6.4.2. x86: `LOCK CMPXCHG`; C11: `atomic_compare_exchange_strong`; Java: `AtomicInteger.compareAndSet`.

compare_and_swap ile Kilit

```
1  int compare_and_swap(int *v, int bekl, int yeni) {
▶    int eski = *v;          /* üç satır birlikte */
▶    if (eski == bekl)       /* TEK atomik komut  */
▶      *v = yeni;
5    return eski;
6  }
7
8  int lock = 0;
9  void gir(void) {
▶    while (compare_and_swap(&lock, 0, 1) != 0)
11     ;                     /* meşgul bekle */
12 }
13 void cik(void) { lock = 0; }
```

T0 — CAS(0→1): eski 0 → kazandı

T1 — hazır

T0'ın CAS'ı `lock`'u 0 bulur, 1 yazar ve **0 döndürür**: döngüden çıkar, kritik bölgede.

Adım 2 / 6

DEĞİŞKENLER

`lock = 1` `T0.eski = 0`

Silberschatz 10e §6.4.2. x86: `LOCK CMPXCHG`; C11: `atomic_compare_exchange_strong`; Java: `AtomicInteger.compareAndSet`.

compare_and_swap ile Kilit

```
1  int compare_and_swap(int *v, int bekl, int yeni) {
▶    int eski = *v;          /* üç satır birlikte */
▶    if (eski == bekl)       /* TEK atomik komut  */
4      *v = yeni;
5    return eski;
6  }
7
8  int lock = 0;
9  void gir(void) {
▶    while (compare_and_swap(&lock, 0, 1) != 0)
11     ;                     /* meşgul bekle */
12 }
13 void cik(void) { lock = 0; }
```

T0 — kritik bölgede

T1 — CAS: eski 1 ≠ 0 → yazmaz

T1'in CAS'ı 1 bulur: beklenen değer değil, **hiçbir şey yazmaz**, 1 döndürür.

Adım 3 / 6

DEĞİŞKENLER

`lock = 1` `T1.eski = 1`

Silberschatz 10e §6.4.2. x86: `LOCK CMPXCHG`; C11: `atomic_compare_exchange_strong`; Java: `AtomicInteger.compareAndSet`.

compare_and_swap ile Kilit

```
1  int compare_and_swap(int *v, int bekl, int yeni) {
2    int eski = *v;          /* üç satır birlikte */
3    if (eski == bekl)       /* TEK atomik komut  */
4      *v = yeni;
5    return eski;
6  }
7
8  int lock = 0;
9  void gir(void) {
▶    while (compare_and_swap(&lock, 0, 1) != 0)
▶      ;                     /* meşgul bekle */
12 }
13 void cik(void) { lock = 0; }
```

T0 — kritik bölgede

T1 — döngüde yeniden deniyor

T1 meşgul bekler; her deneme bellek sözcüğünü değiştirmeden döner.

Adım 4 / 6

DEĞİŞKENLER

`lock = 1` `T1.eski = 1`

Silberschatz 10e §6.4.2. x86: `LOCK CMPXCHG`; C11: `atomic_compare_exchange_strong`; Java: `AtomicInteger.compareAndSet`.

compare_and_swap ile Kilit

```
1  int compare_and_swap(int *v, int bekl, int yeni) {
2    int eski = *v;          /* üç satır birlikte */
3    if (eski == bekl)       /* TEK atomik komut  */
4      *v = yeni;
5    return eski;
6  }
7
8  int lock = 0;
9  void gir(void) {
10   while (compare_and_swap(&lock, 0, 1) != 0)
11     ;                     /* meşgul bekle */
12 }
▶ void cik(void) { lock = 0; }
```

T0 — cik(): lock = 0

T1 — döngüde

T0 çıkarken kilidi 0 yapar (gerçek kodda release sıralamalı yazma).

Adım 5 / 6

DEĞİŞKENLER

`lock = 0`

Silberschatz 10e §6.4.2. x86: `LOCK CMPXCHG`; C11: `atomic_compare_exchange_strong`; Java: `AtomicInteger.compareAndSet`.

compare_and_swap ile Kilit

```
1  int compare_and_swap(int *v, int bekl, int yeni) {
▶    int eski = *v;          /* üç satır birlikte */
▶    if (eski == bekl)       /* TEK atomik komut  */
▶      *v = yeni;
5    return eski;
6  }
7
8  int lock = 0;
9  void gir(void) {
▶    while (compare_and_swap(&lock, 0, 1) != 0)
11     ;                     /* meşgul bekle */
12 }
13 void cik(void) { lock = 0; }
```

T0 — kalan bölüm

T1 — CAS(0→1) kazandı → girdi

T1'in sıradaki CAS'ı başarılı olur. Not: bu kilit **sınırlı beklemeyi** garanti etmez; şanssız bir iş parçacığı hep kaybedebilir.

Adım 6 / 6

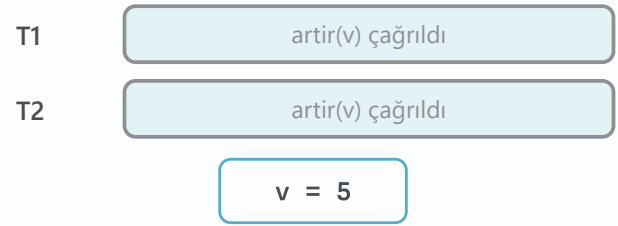
DEĞİŞKENLER

`lock = 1` `T1.eski = 0`

Silberschatz 10e §6.4.2. x86: `LOCK CMPXCHG`; C11: `atomic_compare_exchange_strong`; Java: `AtomicInteger.compareAndSet`.

CAS ile Atomik Artırma

```
▶ void artir(int *v) {
2   int temp;
3   do {
4     temp = *v;              /* oku              */
5   } while (temp != compare_and_swap(v, temp, temp +
  1));
6 }                           /* değişmediyse yaz */
7
8 /* C11: sayaç atomic_int ise tek satır:
  atomic_fetch_add(v, 1); */
```



İki iş parçacığı aynı sayacı artırmak istiyor; başlangıçta `v = 5`.

Adım 1 / 7

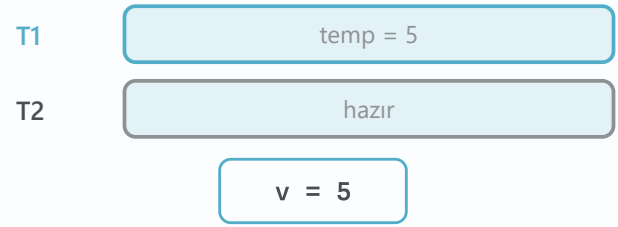
DEĞİŞKENLER

v = **5**

Bu “oku → hesapla → CAS → başarısızsa yeniden dene” döngüsü kilitsiz veri yapılarının temel kalıbıdır.

CAS ile Atomik Artırma

```
1 void artir(int *v) {
2   int temp;
3   do {
▶     temp = *v;              /* oku                */
5   } while (temp != compare_and_swap(v, temp, temp +
  1));
6 }                           /* değişmediyse yaz */
7
8 /* C11: sayaç atomic_int ise tek satır:
  atomic_fetch_add(v, 1); */
```



T1 değeri okur: `temp = 5`. Burada kesilebilir; kilit yoktur.

Adım 2 / 7

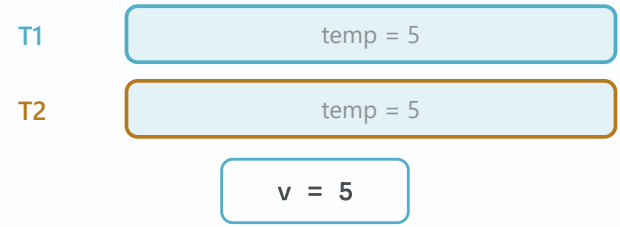
DEĞİŞKENLER

`v = 5` `T1.temp = 5`

Bu “oku → hesapla → CAS → başarısızsa yeniden dene” döngüsü kilitsiz veri yapılarının temel kalıbıdır.

CAS ile Atomik Artırma

```
1 void artir(int *v) {
2   int temp;
3   do {
▶     temp = *v;              /* oku                */
5   } while (temp != compare_and_swap(v, temp, temp +
  1));
6 }                           /* değişmediyse yaz */
7
8 /* C11: sayaç atomic_int ise tek satır:
  atomic_fetch_add(v, 1); */
```



T2 de 5 okur. Düz `v++` olsaydı bir artış kaybolacaktı.

Adım 3 / 7

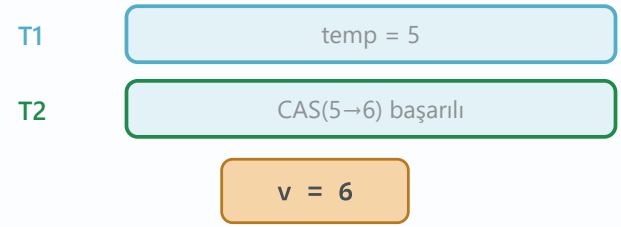
DEĞİŞKENLER

`v = 5` `T1.temp = 5` `T2.temp = 5`

Bu “oku → hesapla → CAS → başarısızsa yeniden dene” döngüsü kilitsiz veri yapılarının temel kalıbıdır.

CAS ile Atomik Artırma

```
1 void artir(int *v) {
2   int temp;
3   do {
4     temp = *v;              /* oku                */
▶   } while (temp != compare_and_swap(v, temp, temp +
  1));
6 }                           /* değişmediyse yaz */
7
8 /* C11: sayaç atomic_int ise tek satır:
  atomic_fetch_add(v, 1); */
```



T2'nin CAS'ı v'yi hâlâ 5 bulur, 6 yazar ve 5 döndürür: `temp`'e eşit → çıkar.

Adım 4 / 7

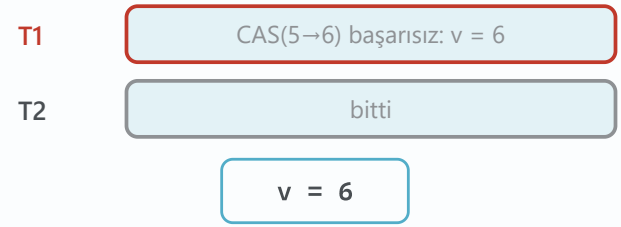
DEĞİŞKENLER

`v = 6` `T1.temp = 5` `T2.temp = 5`

Bu "oku → hesapla → CAS → başarısızsa yeniden dene" döngüsü kilitsiz veri yapılarının temel kalıbıdır.

CAS ile Atomik Artırma

```
1 void artir(int *v) {
2   int temp;
3   do {
4     temp = *v;              /* oku              */
▶   } while (temp != compare_and_swap(v, temp, temp +
  1));
6 }                           /* değişmediyse yaz */
7
8 /* C11: sayaç atomic_int ise tek satır:
  atomic_fetch_add(v, 1); */
```



T1'in CAS'ı v'yi 6 bulur: beklenen 5 değil, yazmaz ve 6 döndürür → **tekrar dene**.

Adım 5 / 7

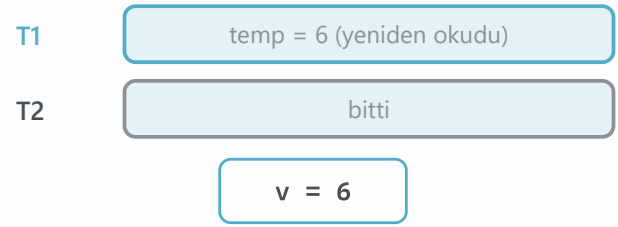
DEĞİŞKENLER

`v = 6` `T1.temp = 5` `T2.temp = 5`

Bu "oku → hesapla → CAS → başarısızsa yeniden dene" döngüsü kilitsiz veri yapılarının temel kalıbıdır.

CAS ile Atomik Artırma

```
1 void artir(int *v) {
2   int temp;
3   do {
▶     temp = *v;              /* oku                */
5   } while (temp != compare_and_swap(v, temp, temp +
  1));
6 }                           /* değişmediyse yaz */
7
8 /* C11: sayaç atomic_int ise tek satır:
  atomic_fetch_add(v, 1); */
```



T1 güncel değeri okur.

Adım 6 / 7

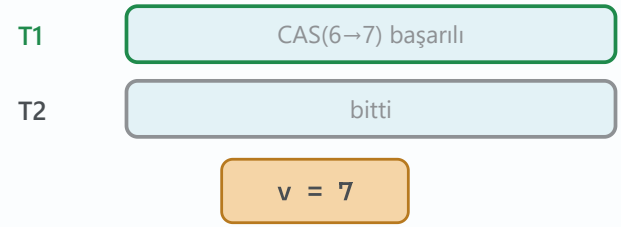
DEĞİŞKENLER

`v = 6` `T1.temp = 6`

Bu “oku → hesapla → CAS → başarısızsa yeniden dene” döngüsü kilitsiz veri yapılarının temel kalıbıdır.

CAS ile Atomik Artırma

```
1 void artir(int *v) {
2   int temp;
3   do {
4     temp = *v;              /* oku              */
▶   } while (temp != compare_and_swap(v, temp, temp +
  1));
6 }                           /* değişmediyse yaz */
7
8 /* C11: sayaç atomic_int ise tek satır:
  atomic_fetch_add(v, 1); */
```



Bu kez başarılı: **v = 7**. Hiçbir artış kaybolmadı ve kimse kilit tutmadı (*kilitsiz* güncelleme).

Adım 7 / 7

DEĞİŞKENLER

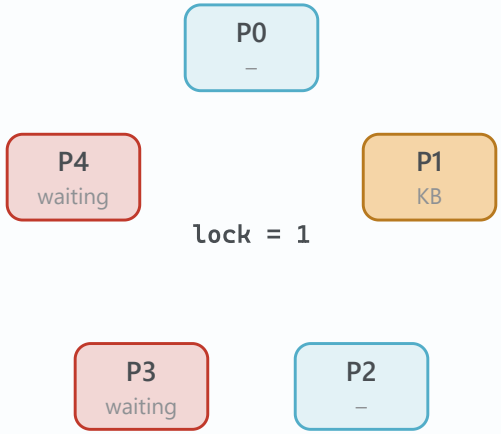
`v = 7` `T1.temp = 6`

Bu “oku → hesapla → CAS → başarısızsa yeniden dene” döngüsü kilitsiz veri yapılarının temel kalıbıdır.

DONANIM DESTEĞİ

Sınırlı Bekleme ile CAS

```
while (true) {
  waiting[i] = true;
  key = 1;
  while (waiting[i] && key == 1)
    key = compare_and_swap(&lock, 0, 1);
  waiting[i] = false;
  /* kritik bölge */
  j = (i + 1) % n;
  while ((j != i) && !waiting[j])
    j = (j + 1) % n;
  if (j == i)
    lock = 0;            /* bekleyen yok */
  else
    waiting[j] = false;  /* sırayı devret */
  /* kalan bölüm */
}
```



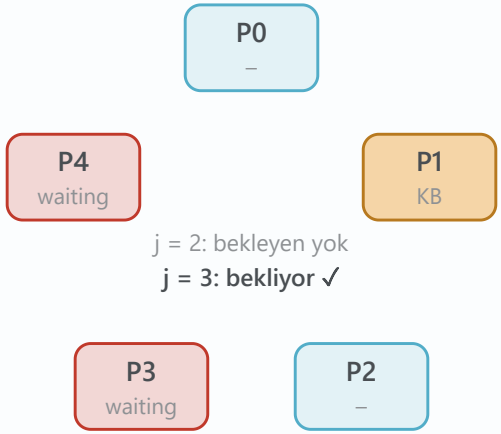
en fazla n − 1 = 4 sıra beklenir

P1 kritik bölgede; P3 ve P4 `waiting` bayrağıyla bekliyor.

Adım 1 / 3

Sınırlı Bekleme ile CAS

```
while (true) {
  waiting[i] = true;
  key = 1;
  while (waiting[i] && key == 1)
    key = compare_and_swap(&lock, 0, 1);
  waiting[i] = false;
  /* kritik bölge */
  j = (i + 1) % n;
  while ((j != i) && !waiting[j])
    j = (j + 1) % n;
  if (j == i)
    lock = 0;            /* bekleyen yok */
  else
    waiting[j] = false;  /* sırayı devret */
  /* kalan bölüm */
}
```

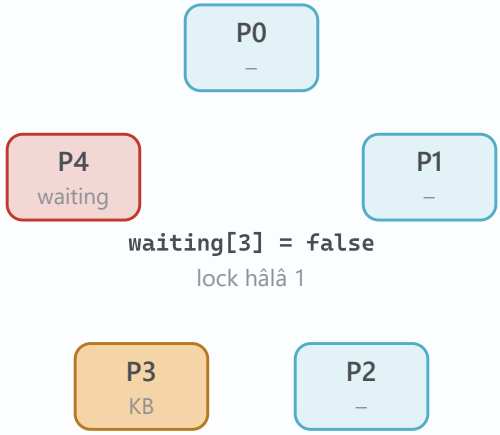


en fazla n − 1 = 4 sıra beklenir

P1 çıkarken `j = 2, 3, …` sırasıyla tarar; ilk bekleyen P3'tür.

Sınırlı Bekleme ile CAS

```
while (true) {
  waiting[i] = true;
  key = 1;
  while (waiting[i] && key == 1)
    key = compare_and_swap(&lock, 0, 1);
  waiting[i] = false;
  /* kritik bölge */
  j = (i + 1) % n;
  while ((j != i) && !waiting[j])
    j = (j + 1) % n;
  if (j == i)
    lock = 0;            /* bekleyen yok */
  else
    waiting[j] = false;  /* sırayı devret */
  /* kalan bölüm */
}
```

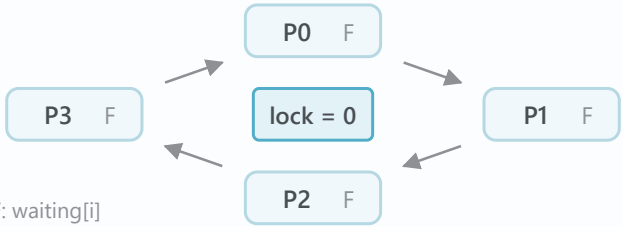


en fazla n − 1 = 4 sıra beklenir

Kilidi bırakmaz, `waiting[3] = false` yapar: P3'ün döngüsü biter ve girer. Döngüsel tarama **sınırlı beklemeyi** sağlar.

Sınırlı Bekleme ile CAS: Kod İzleme

```
1  /* P_i; n = 4 süreç */
2  waiting[i] = true; key = 1;
3  while (waiting[i] && key == 1)
4    key = compare_and_swap(&lock, 0, 1);
5  waiting[i] = false;
6  /* kritik bölge */
7  j = (i + 1) % n;
8  while ((j != i) && !waiting[j])
9    j = (j + 1) % n;
10 if (j == i)
11   lock = 0;            /* bekleyen yok */
12 else
13   waiting[j] = false;  /* sırayı devret */
```



Kilit boş (`lock = 0`), kimse beklemiyor. Dört süreç halka sırasıyla taranacak.

Adım 1 / 11

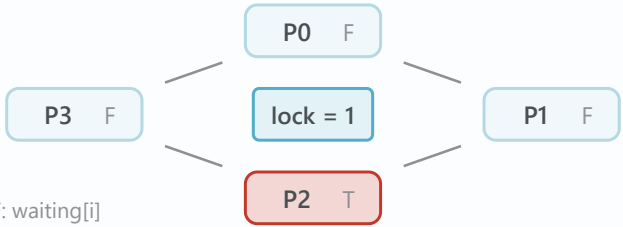
DEĞİŞKENLER

`lock = 0` `key = –` `j = –`

`waiting = [F F F F]` `KB = –`

Sınırlı Bekleme ile CAS: Kod İzleme

```
1  /* P_i; n = 4 süreç */
2  waiting[i] = true; key = 1;
3  while (waiting[i] && key == 1)
4    key = compare_and_swap(&lock, 0, 1);
5  waiting[i] = false;
6  /* kritik bölge */
7  j = (i + 1) % n;
8  while ((j != i) && !waiting[j])
9    j = (j + 1) % n;
10 if (j == i)
11   lock = 0;            /* bekleyen yok */
12 else
13   waiting[j] = false;  /* sırayı devret */
```



P2 gelir: `waiting[2] = true`; CAS `lock`'u 0 bulur, 1 yazar ve 0 döndürür → `key = 0`.

Adım 2 / 11

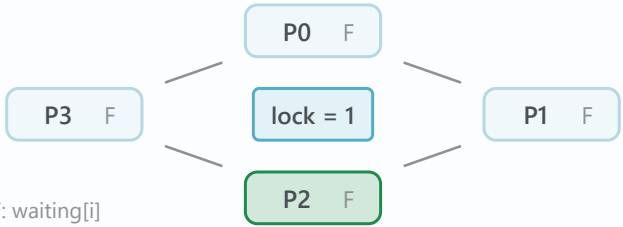
DEĞİŞKENLER

`lock = 1` `key = 0` `j = –`

`waiting = [F F T F]` `KB = –`

Sınırlı Bekleme ile CAS: Kod İzleme

```
1  /* P_i; n = 4 süreç */
2  waiting[i] = true; key = 1;
3  while (waiting[i] && key == 1)
4    key = compare_and_swap(&lock, 0, 1);
▶  waiting[i] = false;
▶  /* kritik bölge */
7  j = (i + 1) % n;
8  while ((j != i) && !waiting[j])
9    j = (j + 1) % n;
10 if (j == i)
11   lock = 0;            /* bekleyen yok */
12 else
13   waiting[j] = false;  /* sırayı devret */
```



Döngü biter; P2 `waiting[2] = false` yapar ve kritik bölgeye girer.

Adım 3 / 11

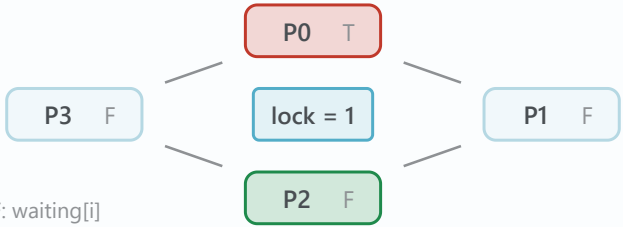
DEĞİŞKENLER

`lock = 1` `key = 0` `j = –`

`waiting = [F F F F]` `KB = P2`

Sınırlı Bekleme ile CAS: Kod İzleme

```
1  /* P_i; n = 4 süreç */
2  waiting[i] = true; key = 1;
▶  while (waiting[i] && key == 1)
▶    key = compare_and_swap(&lock, 0, 1);
5  waiting[i] = false;
6  /* kritik bölge */
7  j = (i + 1) % n;
8  while ((j != i) && !waiting[j])
9    j = (j + 1) % n;
10 if (j == i)
11   lock = 0;            /* bekleyen yok */
12 else
13   waiting[j] = false;  /* sırayı devret */
```



P0 gelir: CAS 1 döndürür (kilit dolu) → `key = 1`, döner.

Adım 4 / 11

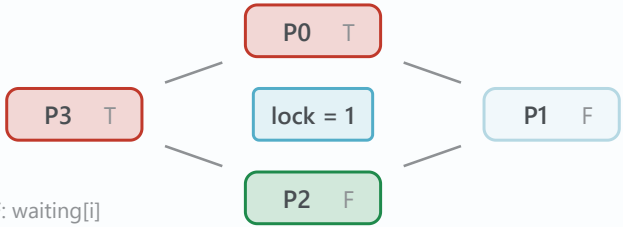
DEĞİŞKENLER

`lock = 1` `key = 1` `j = –`

`waiting = [T F F F]` `KB = P2`

Sınırlı Bekleme ile CAS: Kod İzleme

```
1  /* P_i; n = 4 süreç */
2  waiting[i] = true; key = 1;
▶  while (waiting[i] && key == 1)
▶    key = compare_and_swap(&lock, 0, 1);
5  waiting[i] = false;
6  /* kritik bölge */
7  j = (i + 1) % n;
8  while ((j != i) && !waiting[j])
9    j = (j + 1) % n;
10 if (j == i)
11   lock = 0;            /* bekleyen yok */
12 else
13   waiting[j] = false;  /* sırayı devret */
```



P3 de gelir ve döner. Bekleyenler: P0 ve P3.

Adım 5 / 11

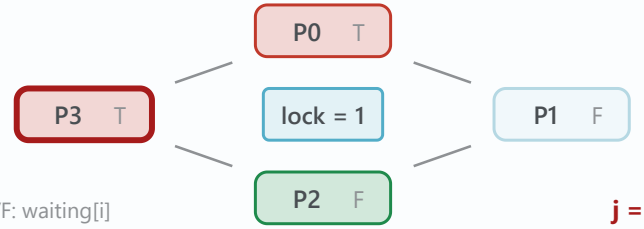
DEĞİŞKENLER

`lock = 1` `key = 1` `j = –`

`waiting = [T F F T]` `KB = P2`

Sınırlı Bekleme ile CAS: Kod İzleme

```
1  /* P_i; n = 4 süreç */
2  waiting[i] = true; key = 1;
3  while (waiting[i] && key == 1)
4    key = compare_and_swap(&lock, 0, 1);
5  waiting[i] = false;
6  /* kritik bölge */
▶  j = (i + 1) % n;
▶  while ((j != i) && !waiting[j])
9    j = (j + 1) % n;
10 if (j == i)
11   lock = 0;            /* bekleyen yok */
12 else
13   waiting[j] = false;  /* sırayı devret */
```



P2 çıkarken taramaya kendinden sonraki **j = 3**'ten başlar; `waiting[3]` true → tarama durur.

Adım 6 / 11

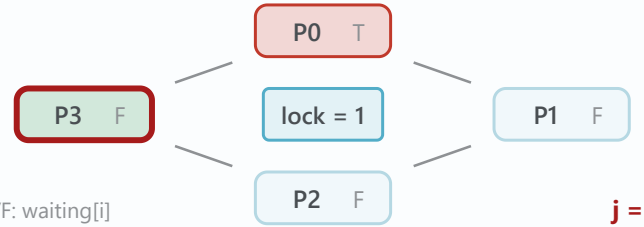
DEĞİŞKENLER

`lock = 1` `key = –` `j = 3`

`waiting = [T F F T]` `KB = P2`

Sınırlı Bekleme ile CAS: Kod İzleme

```
1  /* P_i; n = 4 süreç */
2  waiting[i] = true; key = 1;
3  while (waiting[i] && key == 1)
4    key = compare_and_swap(&lock, 0, 1);
5  waiting[i] = false;
6  /* kritik bölge */
7  j = (i + 1) % n;
8  while ((j != i) && !waiting[j])
9    j = (j + 1) % n;
10 if (j == i)
11   lock = 0;            /* bekleyen yok */
▶  else
▶    waiting[j] = false;  /* sırayı devret */
```



P2 kilidi **bırakmaz**, `waiting[3] = false` yapar: P3'ün döngüsü biter, P3 girer. `lock` hâlâ 1.

Adım 7 / 11

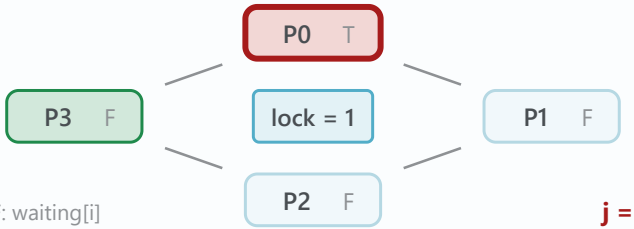
DEĞİŞKENLER

`lock = 1` `key = –` `j = 3`

`waiting = [T F F F]` `KB = P3`

Sınırlı Bekleme ile CAS: Kod İzleme

```
1  /* P_i; n = 4 süreç */
2  waiting[i] = true; key = 1;
3  while (waiting[i] && key == 1)
4    key = compare_and_swap(&lock, 0, 1);
5  waiting[i] = false;
6  /* kritik bölge */
▶  j = (i + 1) % n;
▶  while ((j != i) && !waiting[j])
9    j = (j + 1) % n;
10 if (j == i)
11   lock = 0;            /* bekleyen yok */
12 else
13   waiting[j] = false;  /* sırayı devret */
```



P3 çıkar: j = (3 + 1) % 4 = **0**; `waiting[0]` true → tarama durur.

Adım 8 / 11

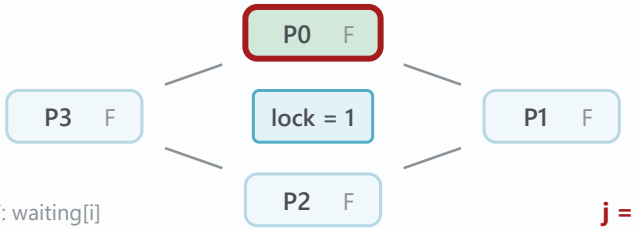
DEĞİŞKENLER

`lock = 1` `key = –` `j = 0`

`waiting = [T F F F]` `KB = P3`

Sınırlı Bekleme ile CAS: Kod İzleme

```
1  /* P_i; n = 4 süreç */
2  waiting[i] = true; key = 1;
3  while (waiting[i] && key == 1)
4    key = compare_and_swap(&lock, 0, 1);
5  waiting[i] = false;
6  /* kritik bölge */
7  j = (i + 1) % n;
8  while ((j != i) && !waiting[j])
9    j = (j + 1) % n;
10 if (j == i)
11   lock = 0;            /* bekleyen yok */
12 else
▶    waiting[j] = false;  /* sırayı devret */
```



Sıra P0'a devredilir. P0, P3'ten önce gelmişti ama halka sırası P3'ü önce seçti; yine de en fazla n − 1 = 3 giriş bekler.

Adım 9 / 11

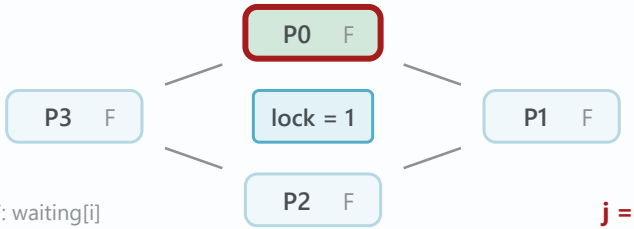
DEĞİŞKENLER

`lock = 1` `key = –` `j = 0`

`waiting = [F F F F]` `KB = P0`

Sınırlı Bekleme ile CAS: Kod İzleme

```
1  /* P_i; n = 4 süreç */
2  waiting[i] = true; key = 1;
3  while (waiting[i] && key == 1)
4    key = compare_and_swap(&lock, 0, 1);
5  waiting[i] = false;
6  /* kritik bölge */
▶  j = (i + 1) % n;
▶  while ((j != i) && !waiting[j])
▶    j = (j + 1) % n;
10 if (j == i)
11   lock = 0;            /* bekleyen yok */
12 else
13   waiting[j] = false;  /* sırayı devret */
```



P0 çıkar: j = 1, 2, 3 hiçbiri beklemiyor; j yeniden **0 = i** olur.

Adım 10 / 11

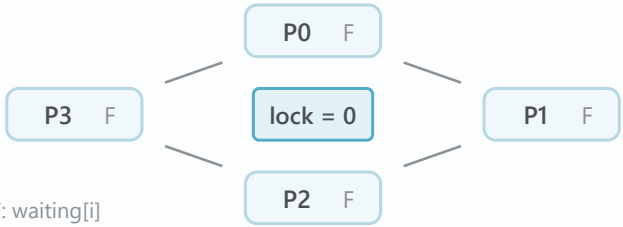
DEĞİŞKENLER

`lock = 1` `key = –` `j = 0`

`waiting = [F F F F]` `KB = P0`

Sınırlı Bekleme ile CAS: Kod İzleme

```
1  /* P_i; n = 4 süreç */
2  waiting[i] = true; key = 1;
3  while (waiting[i] && key == 1)
4    key = compare_and_swap(&lock, 0, 1);
5  waiting[i] = false;
6  /* kritik bölge */
7  j = (i + 1) % n;
8  while ((j != i) && !waiting[j])
9    j = (j + 1) % n;
▶ if (j == i)
▶   lock = 0;              /* bekleyen yok */
12 else
13   waiting[j] = false;   /* sırayı devret */
```



Bekleyen yok → `lock = 0`. Kilit ancak şimdi gerçekten boşalır.

Adım 11 / 11

DEĞİŞKENLER

`lock = 0` `key = –` `j = –`

`waiting = [F F F F]` `KB = –`

Atomik Değişkenler

C11 `<stdatomic.h>`

```c
#include <stdatomic.h>
atomic_int sayac = 0;      /* _Atomic int */

void *is(void *arg) {
  for (int n = 0; n < 100000; n++)
    atomic_fetch_add(&sayac, 1);
  return NULL;
}
/* 2 iş parçacığı + join → 200000 */
```

`_Atomic` türde `sayac++` de atomiktir (varsayılan `seq_cst`). x86'da `lock xadd`, ARMv8.1'de `ldadd` olur.

Java `AtomicInteger`

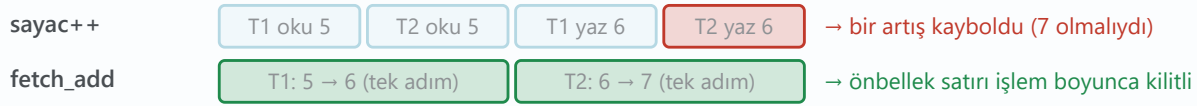
```java
static AtomicInteger atom = new AtomicInteger();
/* iki iş parçacığı, her biri 100_000 kez: */
atom.incrementAndGet();

var v = new AtomicInteger(5);
v.compareAndSet(5, 6);   /* true,  v = 6 */
v.compareAndSet(5, 7);   /* false, v = 6 */
```

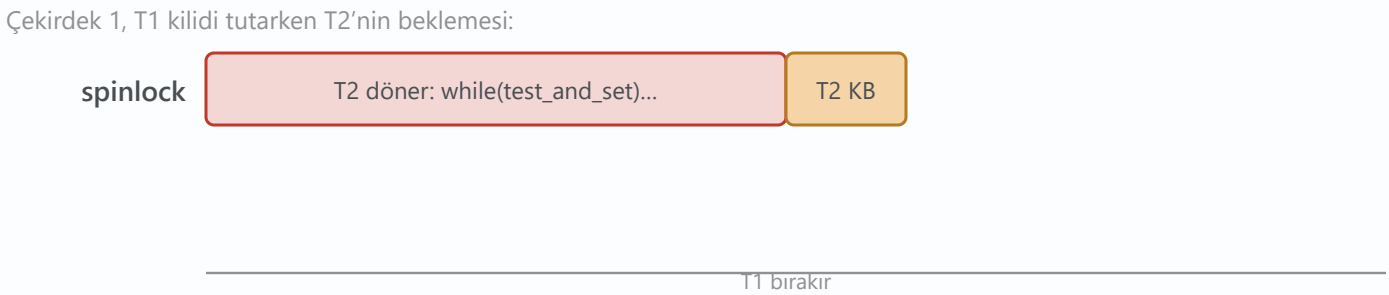
ÇIKTI (JDK 21, HER ÇALIŞTIRMADA AYNI)

```
AtomicInteger = 200000
synchronized  = 200000
true 6
false 6
```

Atomik değişken **tek bir değişkenin** güncellemesini korur. Birden çok değişkeni birlikte tutarlı tutmak (ör. tampon dizisi + sayaç) için yine kilit ya da monitör gerekir.



Spinlock ve Mutex



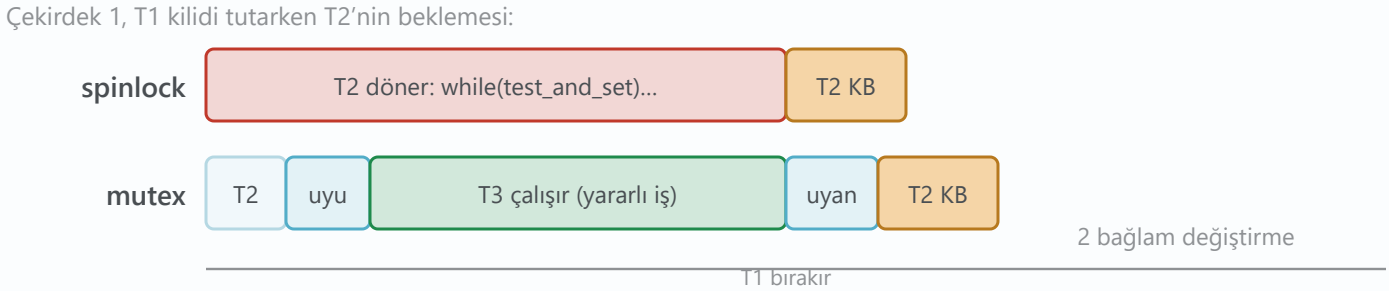
Spinlock: bekleyen CPU'yu bırakmaz; kilit boşalınca bağlam değiştirme olmadan girer.

Adım 1 / 2

	Spinlock	**Mutex (uyuyan)**
Uygun	Çok çekirdek, KB iki bağlam değiştirmeden kısa	Uzun ya da bloke olabilen kritik bölge
Tek çekirdek	Anlamsız: tutan çalışamaz, bekleyen dilimini yakar	Doğru seçim
Örnek	Linux `spin_lock`, `pthread_spin_lock`	`pthread_mutex_t`, Java `synchronized`

Çoğu mutex gerçeklemesi önce kısa süre döner, sonra uyur.

Spinlock ve Mutex



Uyuyan mutex: bekleyen bloke olur, CPU T3'e geçer; bedeli iki bağlam değiştirme.

Adım 2 / 2

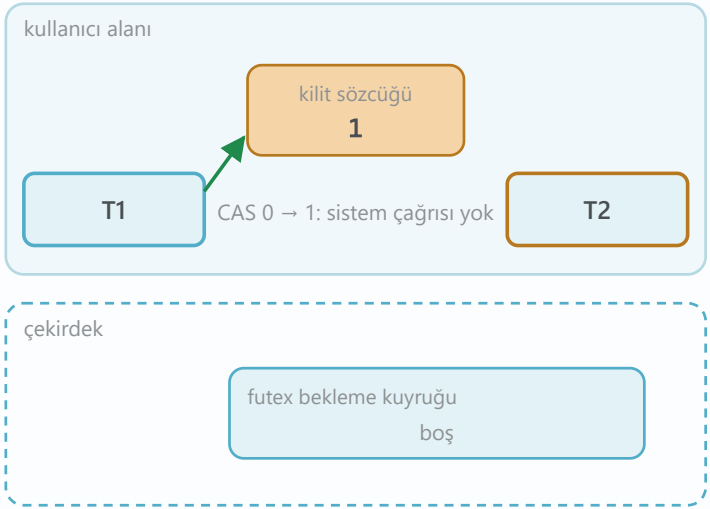
	Spinlock	**Mutex (uyuyan)**
Uygun	Çok çekirdek, KB iki bağlam değiştirmeden kısa	Uzun ya da bloke olabilen kritik bölge
Tek çekirdek	Anlamsız: tutan çalışamaz, bekleyen dilimini yakar	Doğru seçim
Örnek	Linux `spin_lock`, `pthread_spin_lock`	`pthread_mutex_t`, Java `synchronized`

Çoğu mutex gerçeklemesi önce kısa süre döner, sonra uyur.

Mutex Gerçeklemesi: yield ve futex

```
mutex_lock:
  TSL REGISTER,MUTEX | eskisini al, 1 yap
  CMP REGISTER,#0    | boş muydu?
  JZE ok             | evet: alındı
  CALL thread_yield  | hayır: CPU'yu ver
  JMP mutex_lock     | sonra yine dene
ok: RET
mutex_unlock:
  MOVE MUTEX,#0
  RET
```

- Tanenbaum Şekil 2-29: `enter_region`'dan farkı döngü yerine `thread_yield`; kullanıcı düzeyi iş parçacıklarında meşgul beklemeyi önler.
- Linux **futex** (fast userspace mutex): çekişme yoksa sistem çağrısı yapılmaz; glibc `pthread_mutex`, Java kilitleri ve Go bunun üstüne kuruludur.

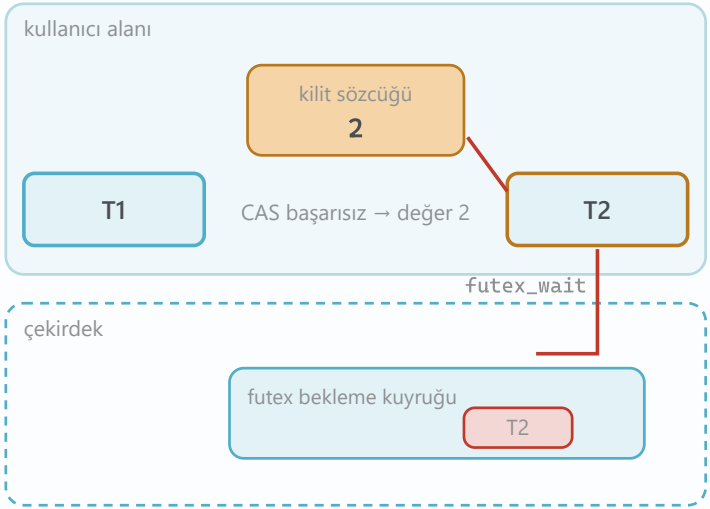


Çekişme yok: T1 sözcüğü tek CAS ile 0 → 1 yapar; çekirdeğe hiç girilmez.

Mutex Gerçeklemesi: yield ve futex

```
mutex_lock:
  TSL REGISTER,MUTEX | eskisini al, 1 yap
  CMP REGISTER,#0    | boş muydu?
  JZE ok             | evet: alındı
  CALL thread_yield  | hayır: CPU'yu ver
  JMP mutex_lock     | sonra yine dene
ok: RET
mutex_unlock:
  MOVE MUTEX,#0
  RET
```

- Tanenbaum Şekil 2-29: `enter_region`'dan farkı döngü yerine `thread_yield`; kullanıcı düzeyi iş parçacıklarında meşgul beklemeyi önler.
- Linux **futex** (fast userspace mutex): çekişme yoksa sistem çağrısı yapılmaz; glibc `pthread_mutex`, Java kilitleri ve Go bunun üstüne kuruludur.

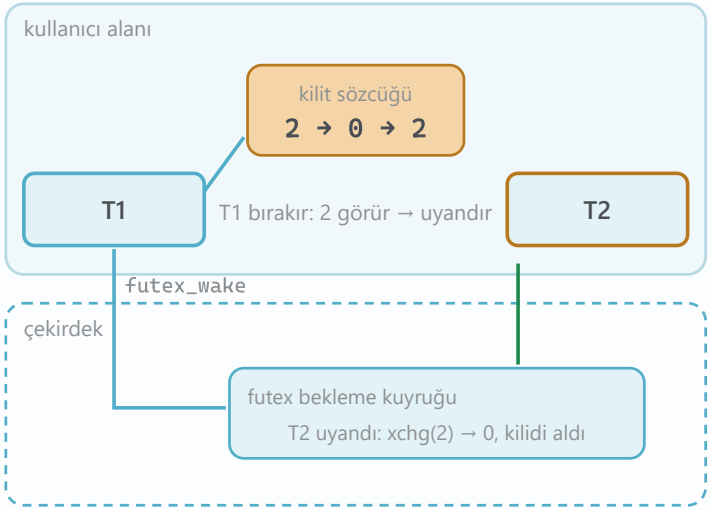


T2 kilidi dolu bulur, değeri 2 (“bekleyen var”) yapar ve `futex_wait` ile uyur.

Mutex Gerçeklemesi: yield ve futex

```
mutex_lock:
  TSL REGISTER,MUTEX | eskisini al, 1 yap
  CMP REGISTER,#0    | boş muydu?
  JZE ok             | evet: alındı
  CALL thread_yield  | hayır: CPU'yu ver
  JMP mutex_lock     | sonra yine dene
ok: RET
mutex_unlock:
  MOVE MUTEX,#0
  RET
```

- Tanenbaum Şekil 2-29: `enter_region`'dan farkı döngü yerine `thread_yield`; kullanıcı düzeyi iş parçacıklarında meşgul beklemeyi önler.
- Linux **futex** (fast userspace mutex): çekişme yoksa sistem çağrısı yapılmaz; glibc `pthread_mutex`, Java kilitleri ve Go bunun üstüne kuruludur.



T1 bırakırken eski değer 2 olduğu için sözcüğü 0 yapar ve `futex_wake` çağırır. T2 uyanır, sözcüğü temkinli olarak yine 2 yapar; eski değer 0 olduğu için kilidi almıştır.

Ölçeklenebilir Sayaç: Çekirdek Başına Sayım

```
1  typedef struct {
2    int global;      pthread_mutex_t gkilit;
3    int yerel[NCPU]; pthread_mutex_t ykilit[NCPU];
4    int S;                    /* aktarma eşiği */
5  } ycount_t;
6
7  void ekle(ycount_t *c, int cpu) {
8    pthread_mutex_lock(&c->ykilit[cpu]);
9    c->yerel[cpu]++;
10   if (c->yerel[cpu] >= c->S) {        /* aktar */
11     pthread_mutex_lock(&c->gkilit);
12     c->global += c->yerel[cpu];
13     pthread_mutex_unlock(&c->gkilit);
14     c->yerel[cpu] = 0;
15   }
16   pthread_mutex_unlock(&c->ykilit[cpu]);
17 }
18 int oku(ycount_t *c) {             /* yaklaşık */
19   pthread_mutex_lock(&c->gkilit);
20   int v = c->global;
21   pthread_mutex_unlock(&c->gkilit);
22   return v;
23 }
```

t	Y1	Y2	Y3	Y4	G	gerçek
0	0	0	0	0	0	0

Dört çekirdek, eşik **S = 4**. Her çekirdeğin kendi sayacı ve kilidi var; genel sayaç G yalnız arada bir güncellenir.

Adım 1 / 7

DEĞİŞKENLER

`S = 4` `G = 0` `gerçek = 0`

Tek kilitli sayaçta her artış aynı kilit için yarışır; çekirdek eklendikçe süre *uzar*. Büyük S: genel kilide seyrek gidilir, ölçeklenme iyi ama `oku()` daha geride. Linux'taki `percpu_counter` aynı fikirdir.

Ölçeklenebilir Sayaç: Çekirdek Başına Sayım

```
1  typedef struct {
2    int global;      pthread_mutex_t gkilit;
3    int yerel[NCPU]; pthread_mutex_t ykilit[NCPU];
4    int S;                     /* aktarma eşiği */
5  } ycount_t;
6
▶ void ekle(ycount_t *c, int cpu) {
▶   pthread_mutex_lock(&c->ykilit[cpu]);
▶   c->yerel[cpu]++;
10   if (c->yerel[cpu] >= c->S) {        /* aktar */
11     pthread_mutex_lock(&c->gkilit);
12     c->global += c->yerel[cpu];
13     pthread_mutex_unlock(&c->gkilit);
14     c->yerel[cpu] = 0;
15   }
16   pthread_mutex_unlock(&c->ykilit[cpu]);
17 }
18 int oku(ycount_t *c) {             /* yaklaşık */
19   pthread_mutex_lock(&c->gkilit);
20   int v = c->global;
21   pthread_mutex_unlock(&c->gkilit);
22   return v;
23 }
```

t	Y1	Y2	Y3	Y4	G	gerçek
0	0	0	0	0	0	0
1	1	0	1	0	0	2

Çekirdek 1 ve 3 aynı anda `ekle` çağırır. Kilitleri farklı olduğundan birbirlerini beklemezler.

Adım 2 / 7

DEĞİŞKENLER

`S = 4` `G = 0` `gerçek = 2`

Tek kilitli sayaçta her artış aynı kilit için yarışır; çekirdek eklendikçe süre *uzar*. Büyük S: genel kilide seyrek gidilir, ölçeklenme iyi ama `oku()` daha geride. Linux'taki `percpu_counter` aynı fikirdir.

Ölçeklenebilir Sayaç: Çekirdek Başına Sayım

```
1  typedef struct {
2    int global;      pthread_mutex_t gkilit;
3    int yerel[NCPU]; pthread_mutex_t ykilit[NCPU];
4    int S;                     /* aktarma eşiği */
5  } ycount_t;
6
7  void ekle(ycount_t *c, int cpu) {
▶    pthread_mutex_lock(&c->ykilit[cpu]);
▶    c->yerel[cpu]++;
10   if (c->yerel[cpu] >= c->S) {        /* aktar */
11     pthread_mutex_lock(&c->gkilit);
12     c->global += c->yerel[cpu];
13     pthread_mutex_unlock(&c->gkilit);
14     c->yerel[cpu] = 0;
15   }
16   pthread_mutex_unlock(&c->ykilit[cpu]);
17 }
18 int oku(ycount_t *c) {             /* yaklaşık */
19   pthread_mutex_lock(&c->gkilit);
20   int v = c->global;
21   pthread_mutex_unlock(&c->gkilit);
22   return v;
23 }
```

t	Y1	Y2	Y3	Y4	G	gerçek
0	0	0	0	0	0	0
1	1	0	1	0	0	2
2	2	1	2	0	0	5

Üç çekirdek daha artırır. Genel kilide hâlâ hiç dokunulmadı.

Adım 3 / 7

DEĞİŞKENLER

`S = 4` `G = 0` `gerçek = 5`

Tek kilitli sayaçta her artış aynı kilit için yarışır; çekirdek eklendikçe süre *uzar*. Büyük S: genel kilide seyrek gidilir, ölçeklenme iyi ama `oku()` daha geride. Linux'taki `percpu_counter` aynı fikirdir.

Ölçeklenebilir Sayaç: Çekirdek Başına Sayım

```
1  typedef struct {
2    int global;      pthread_mutex_t gkilit;
3    int yerel[NCPU]; pthread_mutex_t ykilit[NCPU];
4    int S;                     /* aktarma eşiği */
5  } ycount_t;
6
7  void ekle(ycount_t *c, int cpu) {
▶    pthread_mutex_lock(&c->ykilit[cpu]);
▶    c->yerel[cpu]++;
10   if (c->yerel[cpu] >= c->S) {        /* aktar */
11     pthread_mutex_lock(&c->gkilit);
12     c->global += c->yerel[cpu];
13     pthread_mutex_unlock(&c->gkilit);
14     c->yerel[cpu] = 0;
15   }
16   pthread_mutex_unlock(&c->ykilit[cpu]);
17 }
18 int oku(ycount_t *c) {             /* yaklaşık */
19   pthread_mutex_lock(&c->gkilit);
20   int v = c->global;
21   pthread_mutex_unlock(&c->gkilit);
22   return v;
23 }
```

t	Y1	Y2	Y3	Y4	G	gerçek
0	0	0	0	0	0	0
1	1	0	1	0	0	2
2	2	1	2	0	0	5
3	3	1	3	1	0	8

Gerçek değer 8 oldu ama G hâlâ 0: genel sayaç geride kalıyor.

Adım 4 / 7

DEĞİŞKENLER

`S = 4` `G = 0` `gerçek = 8`

Tek kilitli sayaçta her artış aynı kilit için yarışır; çekirdek eklendikçe süre *uzar*. Büyük S: genel kilide seyrek gidilir, ölçeklenme iyi ama `oku()` daha geride. Linux'taki `percpu_counter` aynı fikirdir.

Ölçeklenebilir Sayaç: Çekirdek Başına Sayım

```
1  typedef struct {
2    int global;      pthread_mutex_t gkilit;
3    int yerel[NCPU]; pthread_mutex_t ykilit[NCPU];
4    int S;                    /* aktarma eşiği */
5  } ycount_t;
6
7  void ekle(ycount_t *c, int cpu) {
8    pthread_mutex_lock(&c->ykilit[cpu]);
9    c->yerel[cpu]++;
▶    if (c->yerel[cpu] >= c->S) {        /* aktar */
▶      pthread_mutex_lock(&c->gkilit);
▶      c->global += c->yerel[cpu];
▶      pthread_mutex_unlock(&c->gkilit);
▶      c->yerel[cpu] = 0;
15   }
16   pthread_mutex_unlock(&c->ykilit[cpu]);
17 }
18 int oku(ycount_t *c) {             /* yaklaşık */
19   pthread_mutex_lock(&c->gkilit);
20   int v = c->global;
21   pthread_mutex_unlock(&c->gkilit);
22   return v;
23 }
```

t	Y1	Y2	Y3	Y4	G	gerçek
0	0	0	0	0	0	0
1	1	0	1	0	0	2
2	2	1	2	0	0	5
3	3	1	3	1	0	8
4	4→0	2	3	1	4	10

Y1 eşiğe ulaştı: çekirdek 1 genel kilidi alır, 4'ü G'ye ekler ve yerel sayacını sıfırlar.

Adım 5 / 7

DEĞİŞKENLER

`S = 4` `G = 4` `gerçek = 10`

Tek kilitli sayaçta her artış aynı kilit için yarışır; çekirdek eklendikçe süre *uzar*. Büyük S: genel kilide seyrek gidilir, ölçeklenme iyi ama `oku()` daha geride. Linux'taki `percpu_counter` aynı fikirdir.

Ölçeklenebilir Sayaç: Çekirdek Başına Sayım

```
1  typedef struct {
2    int global;      pthread_mutex_t gkilit;
3    int yerel[NCPU]; pthread_mutex_t ykilit[NCPU];
4    int S;                     /* aktarma eşiği */
5  } ycount_t;
6
7  void ekle(ycount_t *c, int cpu) {
8    pthread_mutex_lock(&c->ykilit[cpu]);
9    c->yerel[cpu]++;
▶    if (c->yerel[cpu] >= c->S) {        /* aktar */
▶      pthread_mutex_lock(&c->gkilit);
▶      c->global += c->yerel[cpu];
▶      pthread_mutex_unlock(&c->gkilit);
▶      c->yerel[cpu] = 0;
15   }
16   pthread_mutex_unlock(&c->ykilit[cpu]);
17 }
18 int oku(ycount_t *c) {             /* yaklaşık */
19   pthread_mutex_lock(&c->gkilit);
20   int v = c->global;
21   pthread_mutex_unlock(&c->gkilit);
22   return v;
23 }
```

t	Y1	Y2	Y3	Y4	G	gerçek
0	0	0	0	0	0	0
1	1	0	1	0	0	2
2	2	1	2	0	0	5
3	3	1	3	1	0	8
4	0	2	3	1	4	10
5	0	2	4→0	2	8	12

Y3 de eşiğe ulaşır ve aktarır: G = 8.

Adım 6 / 7

DEĞİŞKENLER

`S = 4` `G = 8` `gerçek = 12`

Tek kilitli sayaçta her artış aynı kilit için yarışır; çekirdek eklendikçe süre *uzar*. Büyük S: genel kilide seyrek gidilir, ölçeklenme iyi ama `oku()` daha geride. Linux'taki `percpu_counter` aynı fikirdir.

Ölçeklenebilir Sayaç: Çekirdek Başına Sayım

```
1  typedef struct {
2    int global;      pthread_mutex_t gkilit;
3    int yerel[NCPU]; pthread_mutex_t ykilit[NCPU];
4    int S;                     /* aktarma eşiği */
5  } ycount_t;
6
7  void ekle(ycount_t *c, int cpu) {
8    pthread_mutex_lock(&c->ykilit[cpu]);
9    c->yerel[cpu]++;
10   if (c->yerel[cpu] >= c->S) {        /* aktar */
11     pthread_mutex_lock(&c->gkilit);
12     c->global += c->yerel[cpu];
13     pthread_mutex_unlock(&c->gkilit);
14     c->yerel[cpu] = 0;
15   }
16   pthread_mutex_unlock(&c->ykilit[cpu]);
17 }
▶ int oku(ycount_t *c) {             /* yaklaşık */
▶   pthread_mutex_lock(&c->gkilit);
▶   int v = c->global;
▶   pthread_mutex_unlock(&c->gkilit);
▶   return v;
23 }
```

t	Y1	Y2	Y3	Y4	G	gerçek
0	0	0	0	0	0	0
1	1	0	1	0	0	2
2	2	1	2	0	0	5
3	3	1	3	1	0	8
4	0	2	3	1	4	10
5	0	2	0	2	8	12

oku() = 8, gerçek = 12: fark yerellerde bekliyor

`oku()` 8 döndürür. Gerçek değer 12'dir. En kötü durumda fark NCPU × (S − 1) = 4 × 3 = 12 olabilir.

Adım 7 / 7

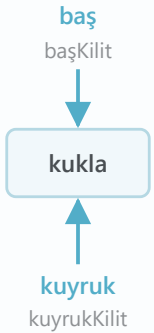
DEĞİŞKENLER

`S = 4` `G = 8` `gerçek = 12`

Tek kilitli sayaçta her artış aynı kilit için yarışır; çekirdek eklendikçe süre *uzar*. Büyük S: genel kilide seyrek gidilir, ölçeklenme iyi ama `oku()` daha geride. Linux'taki `percpu_counter` aynı fikirdir.

Eş Zamanlı Veri Yapılarında Kilit Düzeni

Yapı	Kilit düzeni	Sonuç
Bağlı liste	Tek mutex	Basit. `malloc` kilit dışında yapılmalı; her dönüş yolunda `unlock` unutulmamalı.
El ele kilitleme	Düğüm başına kilit: önce sonrakini al, sonra öncekini bırak	Gezinmeler paralel ilerler ama her düğümdeki kilit bedeli kazancı çoğu zaman yer.
Kuyruk	Baş ve kuyruk için ayrı kilit, kukla düğüm	Ekleme ile çıkarma aynı anda yapılabilir.
Hash tablosu	Kova başına kilitli liste	Farklı kovalara giden işlemler birbirini beklemez: çok iyi ölçeklenir.



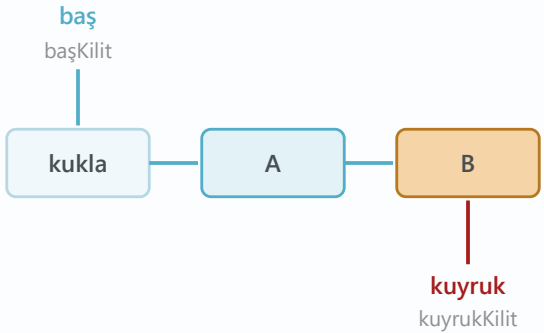
Başlangıçta tek bir **kukla** düğüm vardır; baş da kuyruk da onu gösterir.

Adım 1 / 3

Kod önce tek kilitle doğru yazılır. Ölçüm kilidin darboğaz olduğunu gösterirse kilit bölünür. Daha çok kilit her zaman daha hızlı demek değildir.

Eş Zamanlı Veri Yapılarında Kilit Düzeni

Yapı	Kilit düzeni	Sonuç
Bağlı liste	Tek mutex	Basit. `malloc` kilit dışında yapılmalı; her dönüş yolunda `unlock` unutulmamalı.
El ele kilitleme	Düğüm başına kilit: önce sonrakini al, sonra öncekini bırak	Gezinmeler paralel ilerler ama her düğümdeki kilit bedeli kazancı çoğu zaman yer.
Kuyruk	Baş ve kuyruk için ayrı kilit, kukla düğüm	Ekleme ile çıkarma aynı anda yapılabilir.
Hash tablosu	Kova başına kilitli liste	Farklı kovalara giden işlemler birbirini beklemez: çok iyi ölçeklenir.



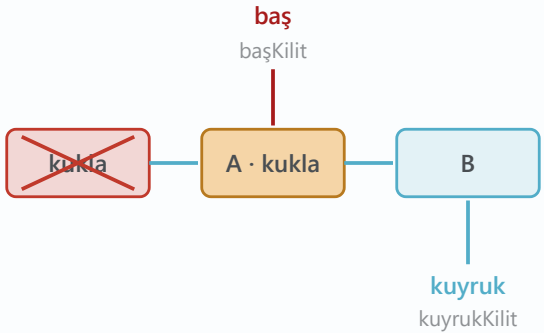
`ekle(A)` , `ekle(B)` : yalnız **kuyrukKilit** alınır, kuyruk ucu ilerler.

Adım 2 / 3

Kod önce tek kilitle doğru yazılır. Ölçüm kilidin darboğaz olduğunu gösterirse kilit bölünür. Daha çok kilit her zaman daha hızlı demek değildir.

Eş Zamanlı Veri Yapılarında Kilit Düzeni

Yapı	Kilit düzeni	Sonuç
Bağlı liste	Tek mutex	Basit. `malloc` kilit dışında yapılmalı; her dönüş yolunda `unlock` unutulmamalı.
El ele kilitleme	Düğüm başına kilit: önce sonrakini al, sonra öncekini bırak	Gezinmeler paralel ilerler ama her düğümdeki kilit bedeli kazancı çoğu zaman yer.
Kuyruk	Baş ve kuyruk için ayrı kilit, kukla düğüm	Ekleme ile çıkarma aynı anda yapılabilir.
Hash tablosu	Kova başına kilitli liste	Farklı kovalara giden işlemler birbirini beklemez: çok iyi ölçeklenir.



`çıkar()` : yalnız **başKilit**. Eski kukla serbest bırakılır, A'nın değeri döner ve A yeni kukla olur.

Adım 3 / 3

Kod önce tek kilitle doğru yazılır. Ölçüm kilidin darboğaz olduğunu gösterirse kilit bölünür. Daha çok kilit her zaman daha hızlı demek değildir.

Semafor

- Dijkstra (1965): yalnızca iki **atomik** işlemle erişilen bir tam sayı.
  - `wait(S)` = **P** = `down` : değer 0 ise **süreç bloke olur** (uyur); değilse 1 azaltılır ve devam edilir.
  - `signal(S)` = **V** = `up` : değer 1 artırılır; bekleyen varsa biri uyandırılır.
- Değer, **kullanılabilir kaynak sayısı** gibi düşünülebilir.
- Silberschatz gerçeklemesinde değer negatif olabilir: `−k`, **k sürecin beklediğini** gösterir.
- `wait` / `signal` 'ın kendisi kısa bir kritik bölgedir: tek çekirdekte kesme kapatılarak, çok çekirdekte spinlock ile korunur. Meşgul bekleme yok olmaz ama birkaç komuta iner.

```
typedef struct {
  int value;
  struct process *list;
} semaphore;
```

Değer 0 iken `down` çağıran süreç Dijkstra semaforunda ve gerçek sistemlerde **uyur**; meşgul bekleyen sürüm yalnızca kavramsal ilk tanımdır.



k = 2 bekliyor — list: uyuyan süreçler (FIFO)

Etkileşimli simülatör: Synchronization Lab (İngilizce): mutex, semafor, üretici–tüketici, okur-yazar ve yemek yiyen filozoflar senaryoları adım adım.

Semafor Bekleme Kuyruğu

```
1  void wait(semaphore *S) {      /* P / down */
2    S->value--;
3    if (S->value < 0) {
4      ekle(S->list, bu_surec);
5      sleep();                    /* bloke ol */
6    }
7  }
8  void signal(semaphore *S) {    /* V / up */
9    S->value++;
10   if (S->value <= 0) {
11     P = cikar(S->list);         /* FIFO */
12     wakeup(P);
13   }
14 }
```

S.value

1

S.list (bekleyenler)

boş

kritik bölge

Kaynak bir tane: `S` 1 ile başlar (ikili semafor, mutex gibi).

Adım 1 / 7

DEĞİŞKENLER

`S.value = 1`

Silberschatz 10e §6.6.2. Kuyruk sırası tanımda belirtilmez; FIFO kullanılmazsa açlık mümkündür.

Semafor Bekleme Kuyruğu

```
▶ void wait(semaphore *S) {      /* P / down */
▶   S->value--;
▶   if (S->value < 0) {
4     ekle(S->list, bu_surec);
5     sleep();                    /* bloke ol */
6   }
7 }
8 void signal(semaphore *S) {    /* V / up */
9   S->value++;
10  if (S->value <= 0) {
11    P = cikar(S->list);         /* FIFO */
12    wakeup(P);
13  }
14 }
```

S.value

0

S.list (bekleyenler)

boş

kritik bölge

T1

T1 `wait` : değer 0 olur, negatif değil → T1 devam eder.

Adım 2 / 7

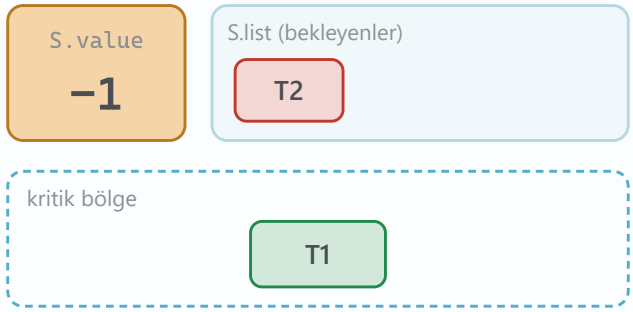
DEĞİŞKENLER

`S.value = 0`

Silberschatz 10e §6.6.2. Kuyruk sırası tanımda belirtilmez; FIFO kullanılmazsa açlık mümkündür.

Semafor Bekleme Kuyruğu

```
1  void wait(semaphore *S) {      /* P / down */
▶    S->value--;
▶    if (S->value < 0) {
▶      ekle(S->list, bu_surec);
▶      sleep();                    /* bloke ol */
6    }
7  }
8  void signal(semaphore *S) {    /* V / up */
9    S->value++;
10   if (S->value <= 0) {
11     P = cikar(S->list);         /* FIFO */
12     wakeup(P);
13   }
14 }
```



T2 `wait` : değer −1 → T2 kuyruğa eklenir ve uyur.

Adım 3 / 7

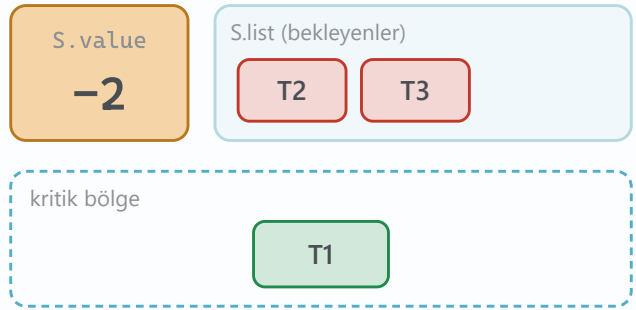
DEĞİŞKENLER

`S.value = −1`

Silberschatz 10e §6.6.2. Kuyruk sırası tanımda belirtilmez; FIFO kullanılmazsa açlık mümkündür.

Semafor Bekleme Kuyruğu

```
1  void wait(semaphore *S) {      /* P / down */
▶    S->value--;
▶    if (S->value < 0) {
▶      ekle(S->list, bu_surec);
▶      sleep();                    /* bloke ol */
6    }
7  }
8  void signal(semaphore *S) {    /* V / up */
9    S->value++;
10   if (S->value <= 0) {
11     P = cikar(S->list);         /* FIFO */
12     wakeup(P);
13   }
14 }
```



T3 de uyur. `−2` : iki süreç bekliyor.

Adım 4 / 7

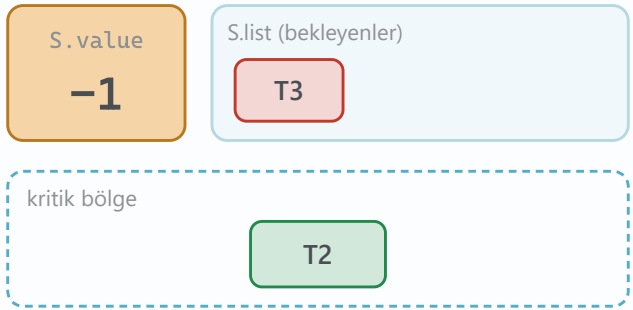
DEĞİŞKENLER

`S.value = −2`

Silberschatz 10e §6.6.2. Kuyruk sırası tanımda belirtilmez; FIFO kullanılmazsa açlık mümkündür.

Semafor Bekleme Kuyruğu

```
1  void wait(semaphore *S) {      /* P / down */
2    S->value--;
3    if (S->value < 0) {
4      ekle(S->list, bu_surec);
5      sleep();                    /* bloke ol */
6    }
7  }
8  void signal(semaphore *S) {    /* V / up */
▶    S->value++;
▶    if (S->value <= 0) {
▶      P = cikar(S->list);         /* FIFO */
▶      wakeup(P);
13   }
14 }
```



T1 `signal` : değer −1, hâlâ ≤ 0 → kuyruğun başındaki T2 uyandırılır ve girer.

Adım 5 / 7

DEĞİŞKENLER

`S.value = −1`

Silberschatz 10e §6.6.2. Kuyruk sırası tanımda belirtilmez; FIFO kullanılmazsa açlık mümkündür.

Semafor Bekleme Kuyruğu

```
1  void wait(semaphore *S) {      /* P / down */
2    S->value--;
3    if (S->value < 0) {
4      ekle(S->list, bu_surec);
5      sleep();                    /* bloke ol */
6    }
7  }
8  void signal(semaphore *S) {    /* V / up */
▶    S->value++;
▶    if (S->value <= 0) {
▶      P = cikar(S->list);        /* FIFO */
▶      wakeup(P);
13   }
14 }
```

S.value

0

S.list (bekleyenler)

boş

kritik bölge

T3

T2 `signal` : değer 0 → T3 uyandırılır.

Adım 6 / 7

DEĞİŞKENLER

`S.value = 0`

Silberschatz 10e §6.6.2. Kuyruk sırası tanımda belirtilmez; FIFO kullanılmazsa açlık mümkündür.

Semafor Bekleme Kuyruğu

```
1  void wait(semaphore *S) {      /* P / down */
2    S->value--;
3    if (S->value < 0) {
4      ekle(S->list, bu_surec);
5      sleep();                    /* bloke ol */
6    }
7  }
8  void signal(semaphore *S) {    /* V / up */
▶    S->value++;
▶    if (S->value <= 0) {
11     P = cikar(S->list);         /* FIFO */
12     wakeup(P);
13   }
14 }
```

S.value
1

S.list (bekleyenler)

boş

kritik bölge

T3 `signal` : değer 1, kimse beklemiyor. Kuyruk FIFO olduğu için bekleme sınırlıdır.

Adım 7 / 7

DEĞİŞKENLER

`S.value = 1`

Silberschatz 10e §6.6.2. Kuyruk sırası tanımda belirtilmez; FIFO kullanılmazsa açlık mümkündür.

Semafor Kullanım Kalıpları

İkili (mutex gibi)

Başlangıç **1**: `wait` … KB … `signal`.

Sayma

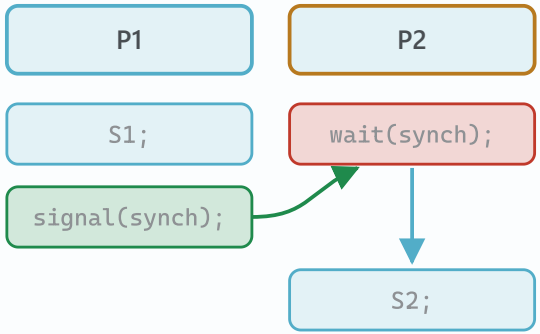
Başlangıç **N**: N özdeş kaynak (bağlantı havuzu, tampon yuvaları).

Sinyal / sıralama

Başlangıç **0**: bir olay gerçekleşene kadar beklemek (sağdaki şekil).

Yaygın hatalar

`signal(m)` … KB … `wait(m)`	Sıra ters: birden çok süreç KB'de
`wait(m)` … KB … `wait(m)`	Kendini bekler: kalıcı bloke
`wait` ya da `signal` unutulur	Dışlama bozulur ya da kilitlenme
İki semafor farklı sırayla alınır	Kilitlenme



synch = 0 ile başlar

Semafor güçlüdür ama *yapısal değildir*: doğruluk her `wait` / `signal` çiftine bağlıdır. Monitörler bu yüzden geliştirildi.

Semaforla Bariyer: Kod İzleme

```
1  #define N 3
▶  semaphore mutex = 1, turnike = 0;
▶  int sayac = 0;
4  void bariyer(void) {
5    down(&mutex);
6    sayac = sayac + 1;
7    if (sayac == N) up(&turnike); /* son */
8    up(&mutex);
9    down(&turnike);  /* turnikeden geç */
10   up(&turnike);    /* arkadakini geçir */
11 }
```

A hesaplıyor | B hesaplıyor | C hesaplıyor

mutex = 1 | sayac = 0

turnike = 0 | bekleyen: (yok)

Üç iş parçacığı fazı hesaplıyor. Turnike kapalı (0); `sayac` gelenleri sayar.

Adım 1 / 10

DEĞİŞKENLER

`mutex = 1` `turnike = 0` `sayac = 0` `kuyruk = []`

Tanenbaum semaforu: `up`, uyuyan varsa değeri artırmadan birini uyandırır. Tek kullanımlık bariyerdir; tekrar kullanılabilir sürüm iki turnike ister.

Semaforla Bariyer: Kod İzleme

```
1  #define N 3
2  semaphore mutex = 1, turnike = 0;
3  int sayac = 0;
4  void bariyer(void) {
▶    down(&mutex);
▶    sayac = sayac + 1;
7    if (sayac == N) up(&turnike); /* son */
8    up(&mutex);
9    down(&turnike);  /* turnikeden geç */
10   up(&turnike);    /* arkadakini geçir */
11 }
```

A aktif | B hesaplıyor | C hesaplıyor | mutex = 0 | sayac = 1

turnike = 0 | bekleyen: (yok)

A bitirdi: mutex'i alır, `sayac = 1`.

Adım 2 / 10

DEĞİŞKENLER

`mutex = 0` `turnike = 0` `sayac = 1` `kuyruk = []`

Tanenbaum semaforu: `up`, uyuyan varsa değeri artırmadan birini uyandırır. Tek kullanımlık bariyerdir; tekrar kullanılabilir sürüm iki turnike ister.

Semaforla Bariyer: Kod İzleme

```
1  #define N 3
2  semaphore mutex = 1, turnike = 0;
3  int sayac = 0;
4  void bariyer(void) {
5    down(&mutex);
6    sayac = sayac + 1;
7    if (sayac == N) up(&turnike); /* son */
▶    up(&mutex);
▶    down(&turnike);  /* turnikeden geç */
10   up(&turnike);    /* arkadakini geçir */
11 }
```

A — uyuyor | B — hesaplıyor | C — hesaplıyor | mutex = 1 | sayac = 1

turnike = 0 | bekleyen: A

1 ≠ 3: A mutex'i bırakır, `down(&turnike)`'ta **uyur**.

Adım 3 / 10

DEĞİŞKENLER

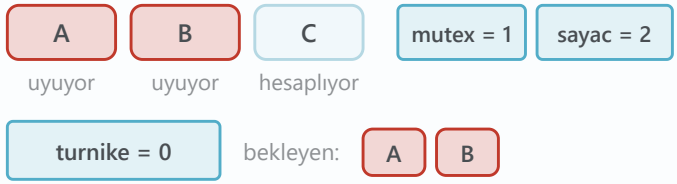
`mutex = 1` `turnike = 0` `sayac = 1`

`kuyruk = [A]`

Tanenbaum semaforu: `up`, uyuyan varsa değeri artırmadan birini uyandırır. Tek kullanımlık bariyerdir; tekrar kullanılabilir sürüm iki turnike ister.

Semaforla Bariyer: Kod İzleme

```
1  #define N 3
2  semaphore mutex = 1, turnike = 0;
3  int sayac = 0;
4  void bariyer(void) {
▶    down(&mutex);
6    sayac = sayac + 1;
7    if (sayac == N) up(&turnike); /* son */
▶    up(&mutex);
▶    down(&turnike);  /* turnikeden geç */
10   up(&turnike);    /* arkadakini geçir */
11 }
```



B de gelir: `sayac = 2`, turnikede uyur.

Adım 4 / 10

DEĞİŞKENLER

`mutex = 1` `turnike = 0` `sayac = 2`

`kuyruk = [A, B]`

Tanenbaum semaforu: `up`, uyuyan varsa değeri artırmadan birini uyandırır. Tek kullanımlık bariyerdir; tekrar kullanılabilir sürüm iki turnike ister.

Semaforla Bariyer: Kod İzleme

```
1  #define N 3
2  semaphore mutex = 1, turnike = 0;
3  int sayac = 0;
4  void bariyer(void) {
▶    down(&mutex);
▶    sayac = sayac + 1;
7    if (sayac == N) up(&turnike); /* son */
8    up(&mutex);
9    down(&turnike);  /* turnikeden geç */
10   up(&turnike);    /* arkadakini geçir */
11 }
```



En yavaş C gelir: `sayac = 3 = N`.

Adım 5 / 10

DEĞİŞKENLER

`mutex = 0` `turnike = 0` `sayac = 3` `kuyruk = [A, B]`

Tanenbaum semaforu: `up`, uyuyan varsa değeri artırmadan birini uyandırır. Tek kullanımlık bariyerdir; tekrar kullanılabilir sürüm iki turnike ister.

Semaforla Bariyer: Kod İzleme

```
1  #define N 3
2  semaphore mutex = 1, turnike = 0;
3  int sayac = 0;
4  void bariyer(void) {
5    down(&mutex);
6    sayac = sayac + 1;
▶    if (sayac == N) up(&turnike); /* son */
8    up(&mutex);
9    down(&turnike);  /* turnikeden geç */
10   up(&turnike);    /* arkadakini geçir */
11 }
```



Son gelen C `up(&turnike)` yapar: bekleyen olduğu için değer artmaz, **A uyandırılır**.

Adım 6 / 10

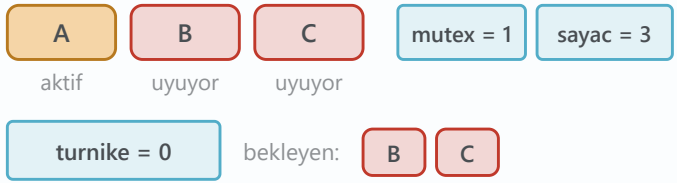
DEĞİŞKENLER

`mutex = 0` `turnike = 0` `sayac = 3` `kuyruk = [B]`

Tanenbaum semaforu: `up`, uyuyan varsa değeri artırmadan birini uyandırır. Tek kullanımlık bariyerdir; tekrar kullanılabilir sürüm iki turnike ister.

Semaforla Bariyer: Kod İzleme

```
1  #define N 3
2  semaphore mutex = 1, turnike = 0;
3  int sayac = 0;
4  void bariyer(void) {
5    down(&mutex);
6    sayac = sayac + 1;
7    if (sayac == N) up(&turnike); /* son */
▶    up(&mutex);
▶    down(&turnike);  /* turnikeden geç */
10   up(&turnike);    /* arkadakini geçir */
11 }
```



C mutex'i bırakır ve turnikede uyur: turnikeden ancak biri geçirirse geçebilir.

Adım 7 / 10

DEĞİŞKENLER

`mutex = 1` `turnike = 0` `sayac = 3`

`kuyruk = [B, C]`

Tanenbaum semaforu: `up`, uyuyan varsa değeri artırmadan birini uyandırır. Tek kullanımlık bariyerdir; tekrar kullanılabilir sürüm iki turnike ister.

Semaforla Bariyer: Kod İzleme

```
1  #define N 3
2  semaphore mutex = 1, turnike = 0;
3  int sayac = 0;
4  void bariyer(void) {
5    down(&mutex);
6    sayac = sayac + 1;
7    if (sayac == N) up(&turnike); /* son */
8    up(&mutex);
9    down(&turnike);  /* turnikeden geç */
▶    up(&turnike);    /* arkadakini geçir */
11 }
```

A geçti | B aktif | C uyuyor | mutex = 1 | sayac = 3

turnike = 0 | bekleyen: C

A turnikeden geçti; `up(&turnike)` ile arkasındaki **B**'yi uyandırır.

Adım 8 / 10

DEĞİŞKENLER

`mutex = 1` `turnike = 0` `sayac = 3` `kuyruk = [C]`

Tanenbaum semaforu: `up`, uyuyan varsa değeri artırmadan birini uyandırır. Tek kullanımlık bariyerdir; tekrar kullanılabilir sürüm iki turnike ister.

Semaforla Bariyer: Kod İzleme

```
1  #define N 3
2  semaphore mutex = 1, turnike = 0;
3  int sayac = 0;
4  void bariyer(void) {
5    down(&mutex);
6    sayac = sayac + 1;
7    if (sayac == N) up(&turnike); /* son */
8    up(&mutex);
9    down(&turnike);  /* turnikeden geç */
▶    up(&turnike);    /* arkadakini geçir */
11 }
```



B geçer ve C'yi uyandırır: turnike bir kişilik döner kapı gibi çalışır.

Adım 9 / 10

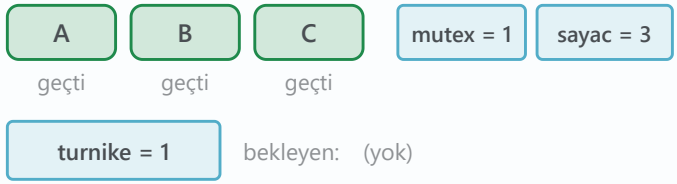
DEĞİŞKENLER

`mutex = 1` `turnike = 0` `sayac = 3` `kuyruk = []`

Tanenbaum semaforu: `up`, uyuyan varsa değeri artırmadan birini uyandırır. Tek kullanımlık bariyerdir; tekrar kullanılabilir sürüm iki turnike ister.

Semaforla Bariyer: Kod İzleme

```
1  #define N 3
2  semaphore mutex = 1, turnike = 0;
3  int sayac = 0;
4  void bariyer(void) {
5    down(&mutex);
6    sayac = sayac + 1;
7    if (sayac == N) up(&turnike); /* son */
8    up(&mutex);
9    down(&turnike);  /* turnikeden geç */
▶    up(&turnike);    /* arkadakini geçir */
11 }
```



C geçer; bekleyen kalmadığı için `turnike = 1` olur. Hepsi sonraki faza geçti; yeniden kullanım için turnike kapatılmalıdır.

Adım 10 / 10

DEĞİŞKENLER

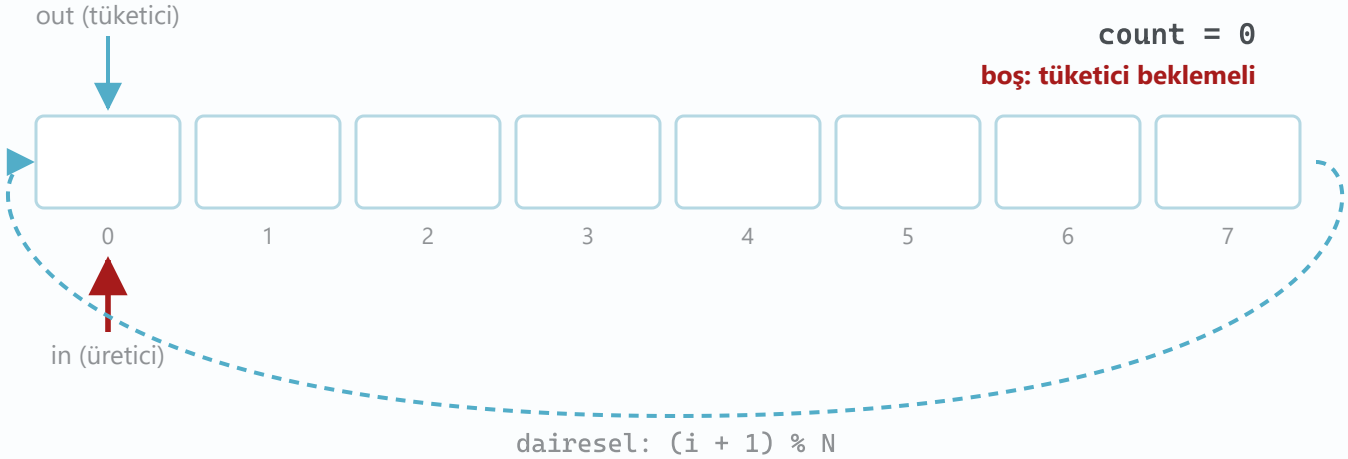
`mutex = 1` `turnike = 1` `sayac = 3` `kuyruk = []`

Tanenbaum semaforu: `up`, uyuyan varsa değeri artırmadan birini uyandırır. Tek kullanımlık bariyerdir; tekrar kullanılabilir sürüm iki turnike ister.

KLASİK PROBLEMLER

Üretici–Tüketici (Sınırlı Tampon) Problemi

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

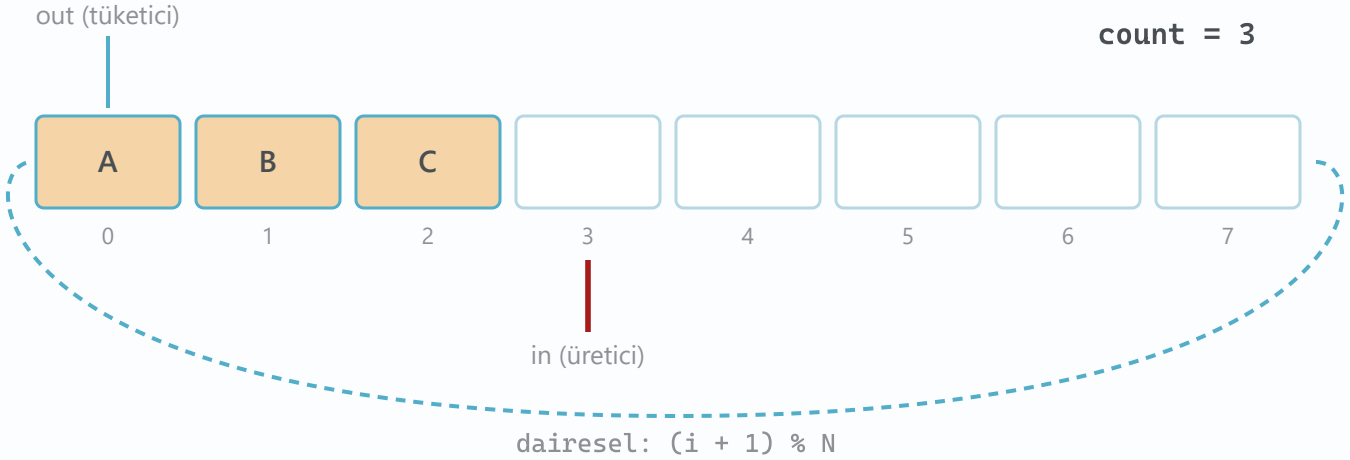


Tampon boş (count = 0): tüketici okuyamaz, beklemelidir.

KLASİK PROBLEMLER

Üretici–Tüketici (Sınırlı Tampon) Problemi

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

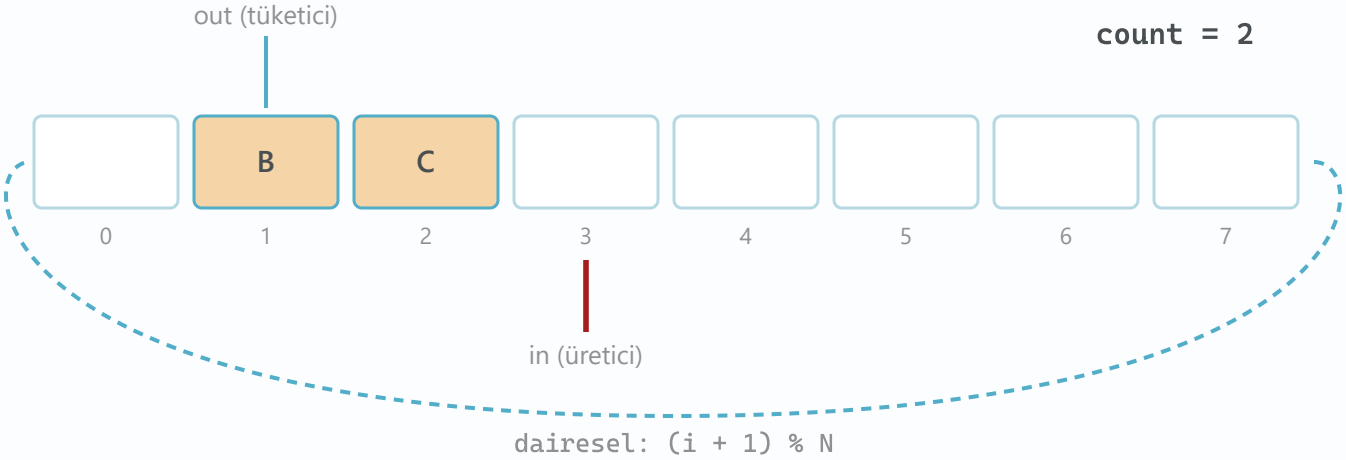


Üretici A, B, C ekler: her eklemede `in = (in + 1) % N`.

KLASİK PROBLEMLER

Üretici–Tüketici (Sınırlı Tampon) Problemi

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

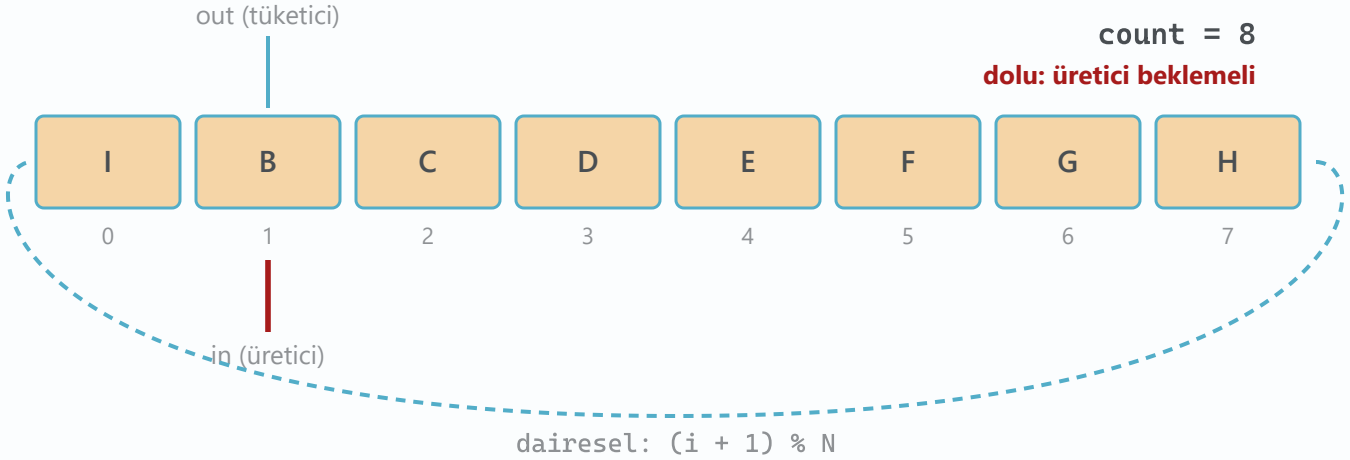


Tüketici A'yı alır: `out = 1`, count = 2.

KLASİK PROBLEMLER

Üretici–Tüketici (Sınırlı Tampon) Problemi

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

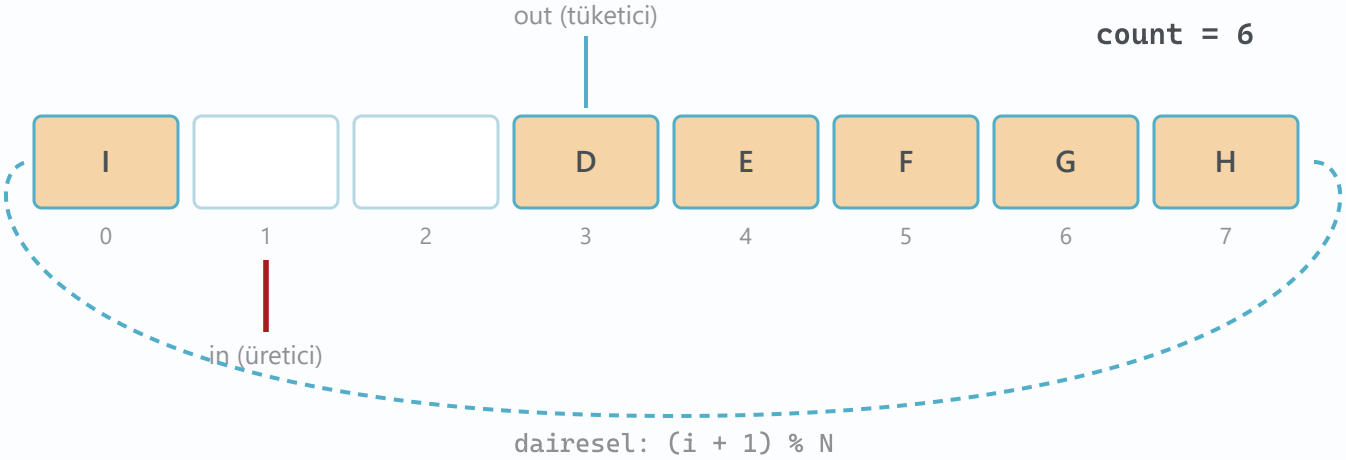


Üretici D...I ekler; `in` başa sarar (1). count = N = 8: tampon dolu, üretici beklemelidir.

KLASİK PROBLEMLER

Üretici–Tüketici (Sınırlı Tampon) Problemi

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



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

KLASİK PROBLEMLER · KODU ADIM ADIM ÇALIŞTIR

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) {
▶          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`



`count = 0`

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

Adım 1 / 7

Tanenbaum MOS Şekil 2-27. `sleep()` çağıranı bloke eder (meşgul bekleme yok), `wakeup(p)` uyuyan `p`'yi hazır yapar. Semafor uyandırmayı sayaç olarak sakladığı için sinyal kaybolmaz.

Ölümcül Yarış: Kayıp Uyandırma

```
1  #define N 100
2  int count = 0;
3  void producer(void) {
4      while (TRUE) {
           item = produce_item();
6          if (count == N) sleep();
           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`



`count = 0`

Üretici bir öğe üretir ve tampona koyar.

Adım 2 / 7

Tanenbaum MOS Şekil 2-27. `sleep()` çağıranı bloke eder (meşgul bekleme yok), `wakeup(p)` uyuyan `p`'yi hazır yapar. Semafor uyandırmayı sayaç olarak sakladığı için sinyal kaybolmaz.

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);
▶          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`



`count = 1`

`count` 1 olur.

Adım 3 / 7

Tanenbaum MOS Şekil 2-27. `sleep()` çağıranı bloke eder (meşgul bekleme yok), `wakeup(p)` uyuyan `p`'yi hazır yapar. Semafor uyandırmayı sayaç olarak sakladığı için sinyal kaybolmaz.

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;
▶          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`



`count = 1`

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

Adım 4 / 7

Tanenbaum MOS Şekil 2-27. `sleep()` çağıranı bloke eder (meşgul bekleme yok), `wakeup(p)` uyuyan `p`'yi hazır yapar. Semafor uyandırmayı sayaç olarak sakladığı için sinyal kaybolmaz.

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) {
▶          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`



`count = 1`

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

Adım 5 / 7

Tanenbaum MOS Şekil 2-27. `sleep()` çağıranı bloke eder (meşgul bekleme yok), `wakeup(p)` uyuyan `p` 'yi hazır yapar. Semafor uyandırmayı sayaç olarak sakladığı için sinyal kaybolmaz.

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);
▶          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`



`count = 100`

Üretici tamponu doldurur; tüketici uyuduğu için kimse okumuyor.

Adım 6 / 7

Tanenbaum MOS Şekil 2-27. `sleep()` çağıranı bloke eder (meşgul bekleme yok), `wakeup(p)` uyuyan `p`'yi hazır yapar. Semafor uyandırmayı sayaç olarak sakladığı için sinyal kaybolmaz.

KLASİK PROBLEMLER · KODU ADIM ADIM ÇALIŞTIR

Ölümcül Yarış: Kayıp Uyandırma

```
1  #define N 100
2  int count = 0;
3  void producer(void) {
4      while (TRUE) {
5          item = produce_item();
▶          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`



`count = 100`

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

Adım 7 / 7

Tanenbaum MOS Şekil 2-27. `sleep()` çağıranı bloke eder (meşgul bekleme yok), `wakeup(p)` uyuyan `p`'yi hazır yapar. Semafor uyandırmayı sayaç olarak sakladığı için sinyal kaybolmaz.

KLASİK PROBLEMLER

Sınırlı Tampon: Semaforlarla

```
#define N 100
semaphore mutex = 1;  /* tampona erişim     */
semaphore empty = N;  /* boş yuva sayısı    */
semaphore full  = 0;  /* dolu yuva sayısı   */

void producer(void) {
  while (true) {
    item = produce_item();
    down(&empty);    /* boş yuva bekle   */
    down(&mutex);    /* KB'ye gir        */
    insert_item(item);
    up(&mutex);      /* KB'den çık       */
    up(&full);       /* dolu sayısı + 1  */
  }
}
```

```
void consumer(void) {
  while (true) {
    down(&full);     /* dolu yuva bekle  */
    down(&mutex);
    item = remove_item();
    up(&mutex);
    up(&empty);      /* boş sayısı + 1   */
    consume_item(item);
  }
}
```

mutex = 1

Karşılıklı dışlama. (0 ile başlasaydı hiç kimse giremezdi.)

empty = N

Üretici tampon doluyken bloke olur.

full = 0

Tüketici tampon boşken bloke olur: *sıralama* (senkronizasyon) görevi.

Sınırlı Tampon Adım Adım

```
▶ semaphore mutex = 1, empty = 2, full = 0;
2
3  void uretici(void) {
4    while (true) {
5      item = uret();
6      wait(&empty);
7      wait(&mutex);
8      tampona_koy(item);
9      signal(&mutex);
10     signal(&full);
11   }
12 }
13 void tuketici(void) {
14   while (true) {
15     wait(&full);
16     wait(&mutex);
17     item = tampondan_al();
18     signal(&mutex);
19     signal(&empty);
20     tuket(item);
21   }
22 }
```

Ü — başlıyor

T — başlıyor

tampon (N = 2)



Tampon boş: 2 boş yuva, 0 dolu yuva.

Adım 1 / 8

DEĞİŞKENLER

`mutex = 1` `empty = 2` `full = 0`

Sınırlı Tampon Adım Adım

```
1  semaphore mutex = 1, empty = 2, full = 0;
2
3  void uretici(void) {
4    while (true) {
5      item = uret();
6      wait(&empty);
7      wait(&mutex);
8      tampona_koy(item);
9      signal(&mutex);
10     signal(&full);
11   }
12 }
13 void tuketici(void) {
14   while (true) {
▶      wait(&full);
16     wait(&mutex);
17     item = tampondan_al();
18     signal(&mutex);
19     signal(&empty);
20     tuket(item);
21   }
22 }
```

Ü — hazır

T — wait(full) → bloke

tampon (N = 2)



Tüketici önce çalışır: `full = 0` → **uyur**. Meşgul bekleme yok.

Adım 2 / 8

DEĞİŞKENLER

`mutex = 1` `empty = 2` `full = 0`

Sınırlı Tampon Adım Adım

```
1  semaphore mutex = 1, empty = 2, full = 0;
2
3  void uretici(void) {
4    while (true) {
5      item = uret();
▶      wait(&empty);
▶      wait(&mutex);
▶      tampona_koy(item);
9      signal(&mutex);
10     signal(&full);
11   }
12 }
13 void tuketici(void) {
14   while (true) {
15     wait(&full);
16     wait(&mutex);
17     item = tampondan_al();
18     signal(&mutex);
19     signal(&empty);
20     tuket(item);
21   }
22 }
```

Ü — A'yı koydu (KB)

T — bloke

tampon (N = 2)



Üretici: `empty` 2→1, `mutex` 1→0, A'yı koyar.

Adım 3 / 8

DEĞİŞKENLER

`mutex = 0` `empty = 1` `full = 0`

Sınırlı Tampon Adım Adım

```
1  semaphore mutex = 1, empty = 2, full = 0;
2
3  void uretici(void) {
4    while (true) {
5      item = uret();
6      wait(&empty);
7      wait(&mutex);
8      tampona_koy(item);
▶      signal(&mutex);
▶      signal(&full);
11   }
12 }
13 void tuketici(void) {
14   while (true) {
15     wait(&full);
16     wait(&mutex);
17     item = tampondan_al();
18     signal(&mutex);
19     signal(&empty);
20     tuket(item);
21   }
22 }
```

Ü signal(full)

T uyandı (hazır)

tampon (N = 2)



`signal(full)` bekleyen tüketiciyi uyandırır; uyanan süreç bu “1”i kullandığı için `full` 0 kalır.

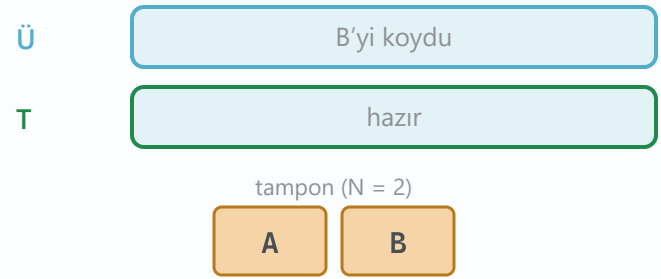
Adım 4 / 8

DEĞİŞKENLER

`mutex = 1` `empty = 1` `full = 0`

Sınırlı Tampon Adım Adım

```
1  semaphore mutex = 1, empty = 2, full = 0;
2
3  void uretici(void) {
4    while (true) {
5      item = uret();
▶      wait(&empty);
▶      wait(&mutex);
▶      tampona_koy(item);
▶      signal(&mutex);
▶      signal(&full);
11   }
12 }
13 void tuketici(void) {
14   while (true) {
15     wait(&full);
16     wait(&mutex);
17     item = tampondan_al();
18     signal(&mutex);
19     signal(&empty);
20     tuket(item);
21   }
22 }
```



Tüketici henüz çalışmadı; üretici B'yi de koyar: `empty = 0`, `full = 1`.

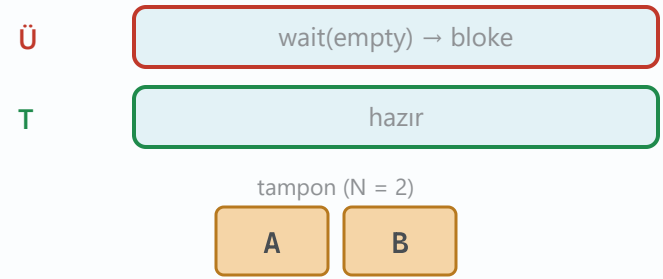
Adım 5 / 8

DEĞİŞKENLER

`mutex = 1` `empty = 0` `full = 1`

Sınırlı Tampon Adım Adım

```
1  semaphore mutex = 1, empty = 2, full = 0;
2
3  void uretici(void) {
4    while (true) {
5      item = uret();
▶      wait(&empty);
7      wait(&mutex);
8      tampona_koy(item);
9      signal(&mutex);
10     signal(&full);
11   }
12 }
13 void tuketici(void) {
14   while (true) {
15     wait(&full);
16     wait(&mutex);
17     item = tampondan_al();
18     signal(&mutex);
19     signal(&empty);
20     tuket(item);
21   }
22 }
```



C için boş yuva yok: üretici **uyur**. Tampon taşmaz.

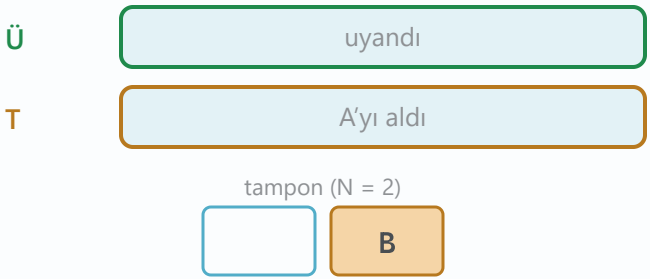
Adım 6 / 8

DEĞİŞKENLER

`mutex = 1` `empty = 0` `full = 1`

Sınırlı Tampon Adım Adım

```
1  semaphore mutex = 1, empty = 2, full = 0;
2
3  void uretici(void) {
4    while (true) {
5      item = uret();
6      wait(&empty);
7      wait(&mutex);
8      tampona_koy(item);
9      signal(&mutex);
10     signal(&full);
11   }
12 }
13 void tuketici(void) {
14   while (true) {
15     wait(&full);
▶      wait(&mutex);
▶      item = tampondan_al();
▶      signal(&mutex);
▶      signal(&empty);
20     tuket(item);
21   }
22 }
```



Tüketici A'yı alır; `signal(empty)` uyuyan üreticiyi uyandırır.

Adım 7 / 8

DEĞİŞKENLER

`mutex = 1` `empty = 0` `full = 1`

Sınırlı Tampon Adım Adım

```
1  semaphore mutex = 1, empty = 2, full = 0;
2
3  void uretici(void) {
4    while (true) {
5      item = uret();
6      wait(&empty);
▶      wait(&mutex);
▶      tampona_koy(item);
▶      signal(&mutex);
▶      signal(&full);
11   }
12 }
13 void tuketici(void) {
14   while (true) {
15     wait(&full);
16     wait(&mutex);
17     item = tampondan_al();
18     signal(&mutex);
19     signal(&empty);
20     tuket(item);
21   }
22 }
```

Ü — C'yi koydu

T — tuket(A)

tampon (N = 2)



Üretici C'yi boşalan yuvaya koyar. Her an: `empty + full` + yoldaki işlemler = N.

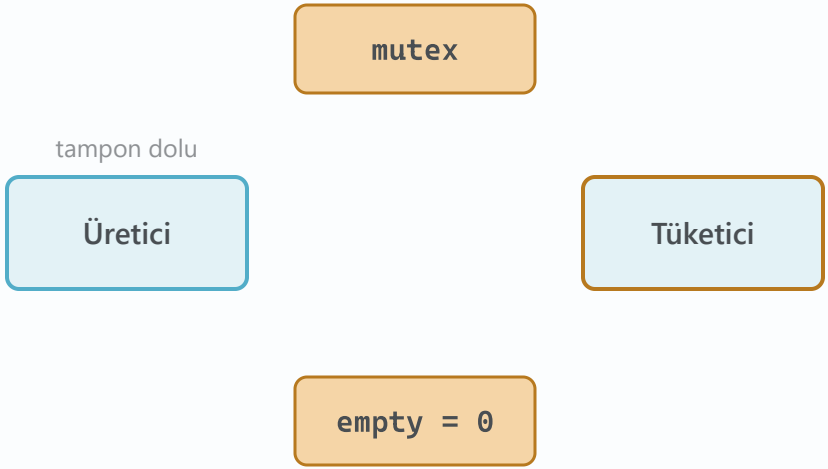
Adım 8 / 8

DEĞİŞKENLER

`mutex = 1` `empty = 0` `full = 2`

down Sırası Değişirse

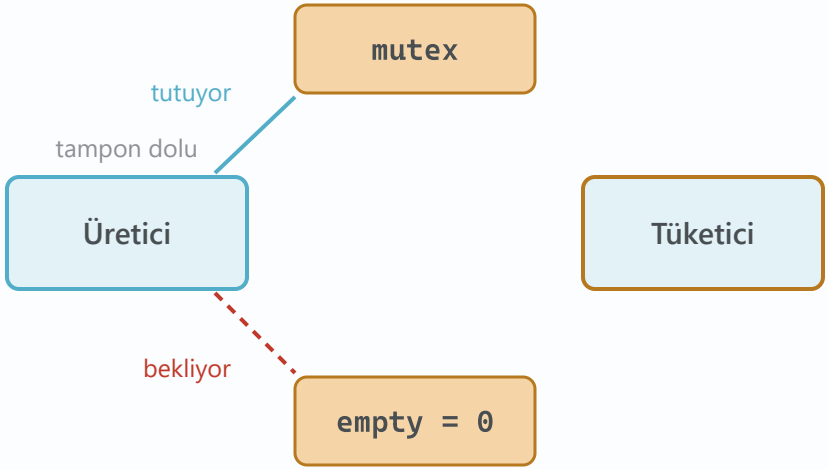
```
void producer(void) {
  while (true) {
    item = produce_item();
    down(&mutex);  /* ← önce */
    down(&empty);  /* ← sonra */
    insert_item(item);
    up(&mutex);
    up(&full);
  }
}
```



Üreticide iki `down`'ın yeri değiştirildi. Tampon tamamen dolu: `empty = 0`.

down Sırası Değişirse

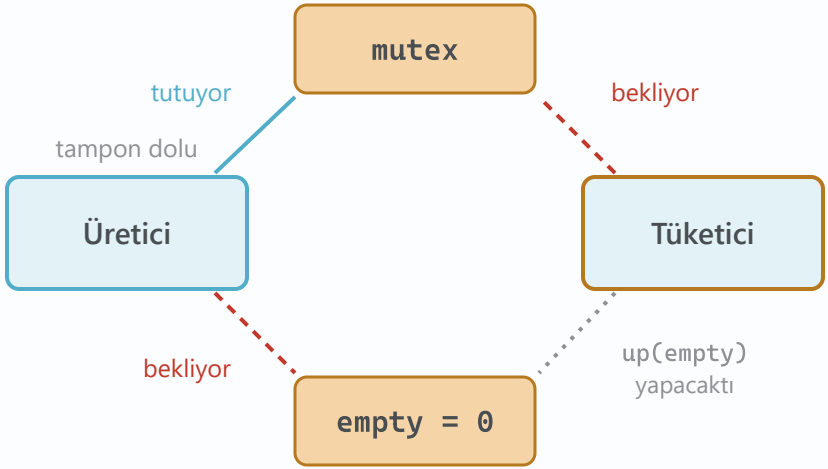
```
void producer(void) {
  while (true) {
    item = produce_item();
    down(&mutex);  /* ← önce */
    down(&empty);  /* ← sonra */
    insert_item(item);
    up(&mutex);
    up(&full);
  }
}
```



Üretici `mutex`'i alır, sonra `down(&empty)`'de uyur — **mutex elindeyken**.

down Sırası Değişirse

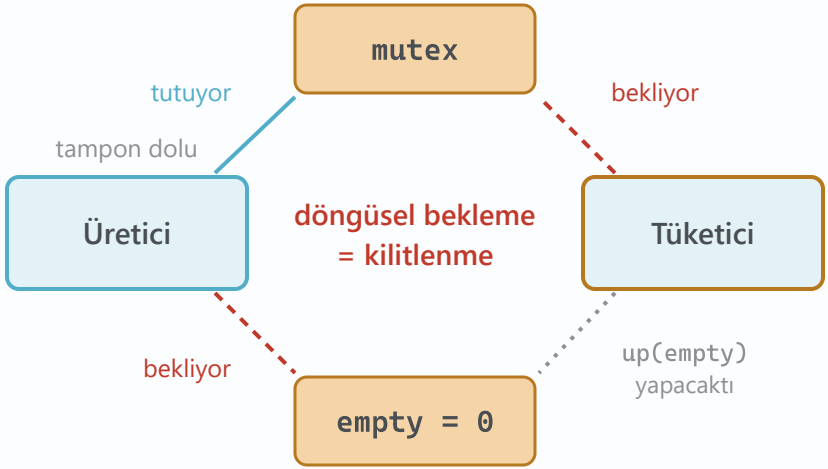
```
void producer(void) {
  while (true) {
    item = produce_item();
    down(&mutex);  /* ← önce */
    down(&empty);  /* ← sonra */
    insert_item(item);
    up(&mutex);
    up(&full);
  }
}
```



Tüketici `down(&full)`'ü geçer ama `down(&mutex)`'te uyur. `up(&empty)` yapacak olan oydu.

down Sırası Değişirse

```
void producer(void) {
  while (true) {
    item = produce_item();
    down(&mutex);   /* ← önce */
    down(&empty);   /* ← sonra */
    insert_item(item);
    up(&mutex);
    up(&full);
  }
}
```



Herkes ötekini bekliyor: **kilitlenme**. Kural: önce sayma semaforu, sonra mutex beklenir; mutex tutulurken uyunmaz.

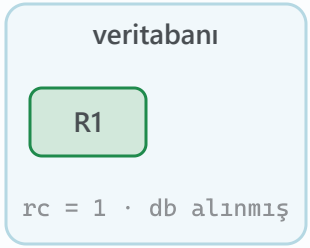
KLASİK PROBLEMLER

Okur-Yazar Problemi

```
semaphore mutex = 1;  /* rc'yi korur         */
semaphore db = 1;     /* veritabanını korur */
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);
    if (++rc == 1) down(&db);  /* ilk okur  */
    up(&mutex);
    read_data_base();
    down(&mutex);
    if (--rc == 0) up(&db);    /* son okur  */
    up(&mutex);
    use_data_read();
  }
}
```

ilk okur db'yi kilitler



Okurlar birbirini dışlamaz; yalnız **ilk** okur `db` 'yi alır.

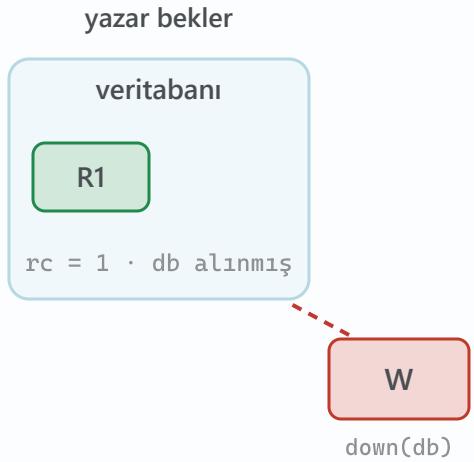
Adım 1 / 5

KLASİK PROBLEMLER

Okur-Yazar Problemi

```
semaphore mutex = 1;  /* rc'yi korur         */
semaphore db = 1;     /* veritabanını korur */
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);
    if (++rc == 1) down(&db);  /* ilk okur  */
    up(&mutex);
    read_data_base();
    down(&mutex);
    if (--rc == 0) up(&db);    /* son okur  */
    up(&mutex);
    use_data_read();
  }
}
```



Bir yazar gelir: `down(&db)`'de bekler.

Adım 2 / 5

Okur-Yazar Problemi

```
semaphore mutex = 1;  /* rc'yi korur         */
semaphore db = 1;     /* veritabanını korur */
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);
    if (++rc == 1) down(&db);  /* ilk okur  */
    up(&mutex);
    read_data_base();
    down(&mutex);
    if (--rc == 0) up(&db);    /* son okur  */
    up(&mutex);
    use_data_read();
  }
}
```

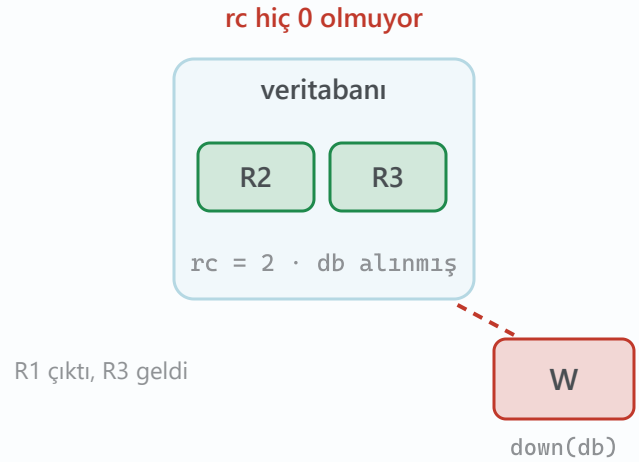


Yeni okur R2 `db`'ye dokunmadan girer: `rc = 2`.

Okur-Yazar Problemi

```
semaphore mutex = 1;  /* rc'yi korur         */
semaphore db = 1;     /* veritabanını korur */
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);
    if (++rc == 1) down(&db);  /* ilk okur  */
    up(&mutex);
    read_data_base();
    down(&mutex);
    if (--rc == 0) up(&db);    /* son okur  */
    up(&mutex);
    use_data_read();
  }
}
```



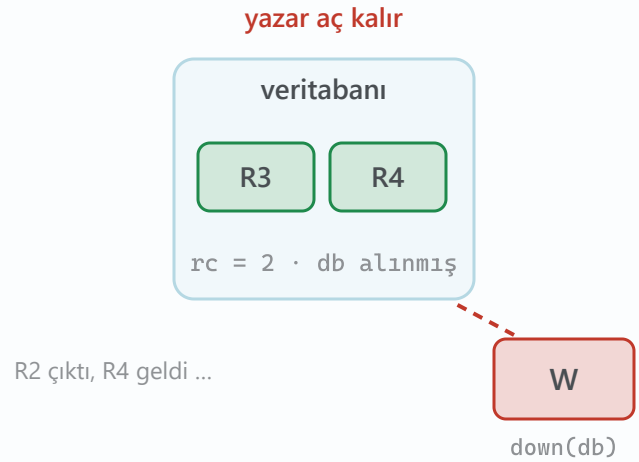
R1 çıkarken R3 gelir: `rc` 1'e inip yeniden 2 olur, hiç 0 olmaz.

KLASİK PROBLEMLER

Okur-Yazar Problemi

```
semaphore mutex = 1;  /* rc'yi korur         */
semaphore db = 1;     /* veritabanını korur */
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);
    if (++rc == 1) down(&db);  /* ilk okur  */
    up(&mutex);
    read_data_base();
    down(&mutex);
    if (--rc == 0) up(&db);    /* son okur  */
    up(&mutex);
    use_data_read();
  }
}
```



Okur akışı sürdükçe yazar **süresiz bekler**: bu çözüm okur önceliklidir ve **yazar açlığına** açıktır.

Okur-Yazar: Varyantlar ve API'ler

Varyant	Davranış
1. okur-yazar (okur öncelikli)	Yazar ancak hiç okur yokken girer; **yazarlar aç kalabilir**.
2. okur-yazar (yazar öncelikli)	Bekleyen yazar varsa yeni okur giremez; **okurlar aç kalabilir**.
Adil (turnike)	Gelen herkes önce aynı turnikeden geçer; yazar beklerken turnikeyi tutar.
`pthread_rwlock_t`	glibc varsayılanı okur öncelikli; yazar önceliği ayarlanabilir.
Java `ReentrantReadWriteLock`	`new ReentrantReadWriteLock(true)` adil (FIFO'ya yakın) sıralama.
Linux seqlock, RCU	Okurların hiç kilit almadığı, okuma ağırlıklı çekirdek yapıları.

```
/* Downey: açlıksız çözüm */
/* yazar */
wait(turnike);
wait(odaBos);
  yaz();
signal(turnike);
signal(odaBos);

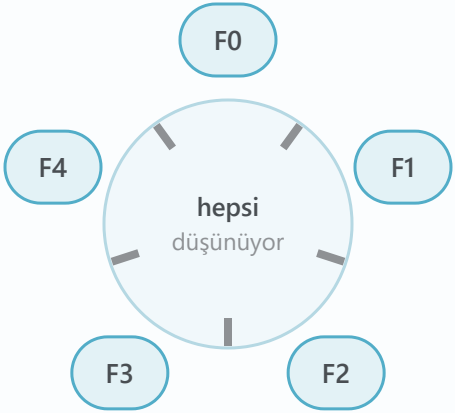
/* okur */
wait(turnike); signal(turnike);
/* ilk okur odaBos'u alır,
   son okur bırakır (rc ile) */
  oku();
```

Yemek Yiyen Filozoflar: Naif Çözüm

```
semaphore catal[5];        /* hepsi 1 */

void filozof(int i) {
  while (true) {
    dusun();
    wait(&catal[i]);              /* sol */
    wait(&catal[(i + 1) % 5]);    /* sağ */
    ye();
    signal(&catal[(i + 1) % 5]);
    signal(&catal[i]);
  }
}
```

- Dijkstra (1965): her filozof düşünür, acıkınca **iki** komşu çatalı alıp yer.
- Kısıtlı kaynakların süreçler arasında **kilitlenmesiz ve açlıksız** paylaştırılmasının modelidir.



Çatallar boşta, herkes düşünüyor.

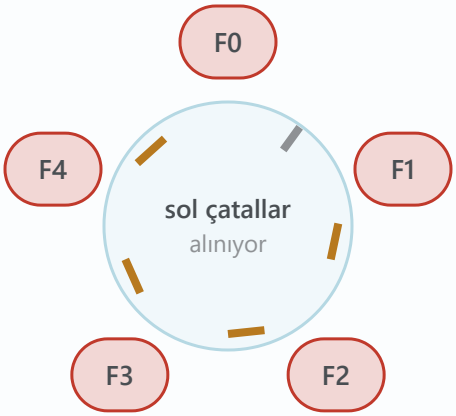
Adım 1 / 4

Yemek Yiyen Filozoflar: Naif Çözüm

```
semaphore catal[5];        /* hepsi 1 */

void filozof(int i) {
  while (true) {
    dusun();
    wait(&catal[i]);              /* sol */
    wait(&catal[(i + 1) % 5]);    /* sağ */
    ye();
    signal(&catal[(i + 1) % 5]);
    signal(&catal[i]);
  }
}
```

- Dijkstra (1965): her filozof düşünür, acıkınca **iki** komşu çatalı alıp yer.
- Kısıtlı kaynakların süreçler arasında **kilitlenmesiz ve açlıksız** paylaştırılmasının modelidir.



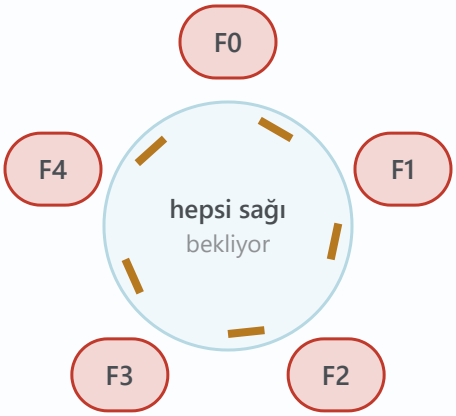
Hepsi aynı anda acıkır ve **sol** çatalını alır...

Yemek Yiyen Filozoflar: Naif Çözüm

```
semaphore catal[5];        /* hepsi 1 */

void filozof(int i) {
  while (true) {
    dusun();
    wait(&catal[i]);              /* sol */
    wait(&catal[(i + 1) % 5]);    /* sağ */
    ye();
    signal(&catal[(i + 1) % 5]);
    signal(&catal[i]);
  }
}
```

- Dijkstra (1965): her filozof düşünür, acıkınca **iki** komşu çatalı alıp yer.
- Kısıtlı kaynakların süreçler arasında **kilitlenmesiz ve açlıksız** paylaştırılmasının modelidir.



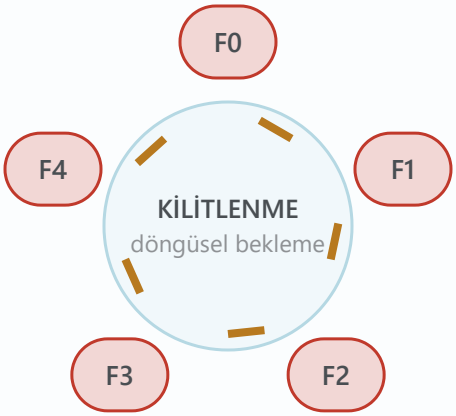
...beşinci de alır. Şimdi herkes **sağ** çatalı bekliyor.

Yemek Yiyen Filozoflar: Naif Çözüm

```
semaphore catal[5];        /* hepsi 1 */

void filozof(int i) {
  while (true) {
    dusun();
    wait(&catal[i]);              /* sol */
    wait(&catal[(i + 1) % 5]);    /* sağ */
    ye();
    signal(&catal[(i + 1) % 5]);
    signal(&catal[i]);
  }
}
```

- Dijkstra (1965): her filozof düşünür, acıkınca **iki** komşu çatalı alıp yer.
- Kısıtlı kaynakların süreçler arasında **kilitlenmesiz ve açlıksız** paylaştırılmasının modelidir.



Kimse bırakmıyor: **kilitlenme**. Her çatal semaforla korunsa bile çözüm yanlıştır.

KLASİK PROBLEMLER

Filozoflar: Çözüm Yolları

Masaya en fazla dört

Başlangıcı 4 olan bir `oda` semaforu: en az bir filozof iki çatala ulaşır. Döngüsel bekleme kırılır.

Kaynak sıralaması (asimetri)

Her filozof önce *numarası küçük* çatalı alır; F4 önce çatal 0'ı ister. Genel kural: kilitleri hep aynı global sırayla al.

İki çatalı birlikte al

Tanenbaum: `state[]` dizisi ve filozof başına semafor; yalnız iki komşu da yemiyorsa yemeye geç.

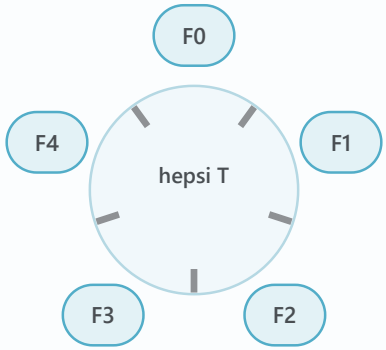
Monitör

Silberschatz: `pickup(i)` / `putdown(i)` ve her filozof için koşul değişkeni `self[i]`; aynı `test()` mantığı.

Kilitlenmesiz olmak **açlıksız** olmak demek değildir: iki komşusu sırayla yiyen bir filozof sonsuza dek bekleyebilir. Açlığı önlemek için ek kural (sıra numarası, yaşlandırma) gerekir.

Tanenbaum Çözümü Adım Adım

```
1  void take_forks(int i) {
2    down(&mutex);
3    state[i] = HUNGRY;
4    test(i);
5    up(&mutex);
6    down(&s[i]);    /* çatal yoksa bloke */
7  }
8  void put_forks(int i) {
9    down(&mutex);
10   state[i] = THINKING;
11   test(LEFT);     /* (i+4) % 5 */
12   test(RIGHT);    /* (i+1) % 5 */
13   up(&mutex);
14 }
15 void test(int i) {
16   if (state[i] == HUNGRY && state[LEFT] != EATING
17                          && state[RIGHT] != EATING) {
18     state[i] = EATING;
19     up(&s[i]);
20   }
21 }
```



Başlangıç: herkes `THINKING` (T). `s[i]` semaforları 0.

Adım 1 / 6

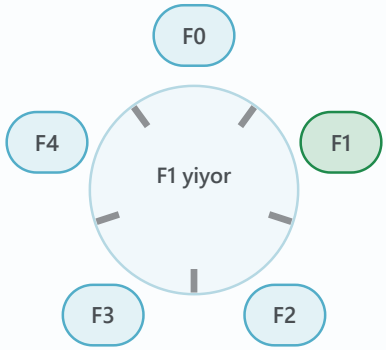
DEĞİŞKENLER

`state` = **TTTTT**

Tanenbaum MOS Şekil 2-47. Kilitlenme yoktur ve beş filozoftan en fazla ⌊5/2⌋ = 2'si aynı anda yer; ancak F1 ve F3 sırayla yemeye devam ederse F2 teorik olarak aç kalabilir.

Tanenbaum Çözümü Adım Adım

```
 1 void take_forks(int i) {
 2   down(&mutex);
 3   state[i] = HUNGRY;
 4   test(i);
 5   up(&mutex);
 6   down(&s[i]);    /* çatal yoksa bloke */
 7 }
 8 void put_forks(int i) {
 9   down(&mutex);
10   state[i] = THINKING;
11   test(LEFT);     /* (i+4) % 5 */
12   test(RIGHT);    /* (i+1) % 5 */
13   up(&mutex);
14 }
15 void test(int i) {
16   if (state[i] == HUNGRY && state[LEFT] != EATING
17                          && state[RIGHT] != EATING) {
18     state[i] = EATING;
19     up(&s[i]);
20   }
21 }
```



F1 acıkır; `test(1)` : komşular F0, F2 yemiyor → `EATING` , `up(&s[1])` ; ardından `down(&s[1])` hemen geçer.

Adım 2 / 6

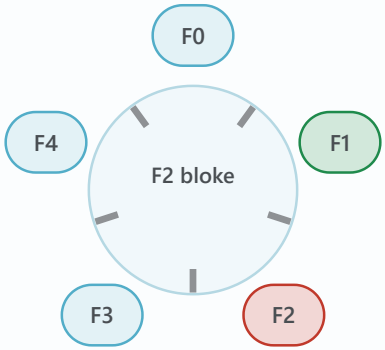
DEĞİŞKENLER

`state = TETTT` `s[1] = 0`

Tanenbaum MOS Şekil 2-47. Kilitlenme yoktur ve beş filozoftan en fazla ⌊5/2⌋ = 2'si aynı anda yer; ancak F1 ve F3 sırayla yemeye devam ederse F2 teorik olarak aç kalabilir.

Tanenbaum Çözümü Adım Adım

```
1  void take_forks(int i) {
2    down(&mutex);
▶    state[i] = HUNGRY;
▶    test(i);
5    up(&mutex);
▶    down(&s[i]);    /* çatal yoksa bloke */
7  }
8  void put_forks(int i) {
9    down(&mutex);
10   state[i] = THINKING;
11   test(LEFT);     /* (i+4) % 5 */
12   test(RIGHT);    /* (i+1) % 5 */
13   up(&mutex);
14 }
15 void test(int i) {
16   if (state[i] == HUNGRY && state[LEFT] != EATING
17                          && state[RIGHT] != EATING) {
18     state[i] = EATING;
19     up(&s[i]);
20   }
21 }
```



F2 acıkır; sol komşu F1 yiyor → test başarısız, `down(&s[2])` 'de uyur.

Adım 3 / 6

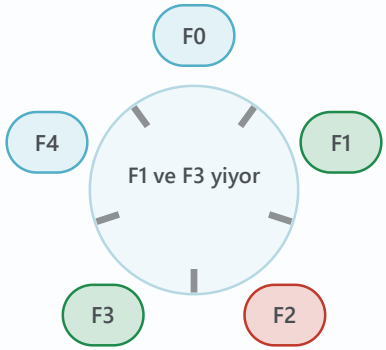
DEĞİŞKENLER

`state = TEHTT` `s[2] = 0`

Tanenbaum MOS Şekil 2-47. Kilitlenme yoktur ve beş filozoftan en fazla ⌊5/2⌋ = 2'si aynı anda yer; ancak F1 ve F3 sırayla yemeye devam ederse F2 teorik olarak aç kalabilir.

Tanenbaum Çözümü Adım Adım

```
 1 void take_forks(int i) {
 2   down(&mutex);
 3   state[i] = HUNGRY;
 4   test(i);
 5   up(&mutex);
 6   down(&s[i]);    /* çatal yoksa bloke */
 7 }
 8 void put_forks(int i) {
 9   down(&mutex);
10   state[i] = THINKING;
11   test(LEFT);     /* (i+4) % 5 */
12   test(RIGHT);    /* (i+1) % 5 */
13   up(&mutex);
14 }
15 void test(int i) {
16   if (state[i] == HUNGRY && state[LEFT] != EATING
17                          && state[RIGHT] != EATING) {
18     state[i] = EATING;
19     up(&s[i]);
20   }
21 }
```



F3 acıkır; F2 aç ama yemiyor, F4 düşünüyor → F3 yer. Aynı anda iki filozof yiyebilir.

Adım 4 / 6

DEĞİŞKENLER

`state = TEHET` `s[3] = 0`

Tanenbaum MOS Şekil 2-47. Kilitlenme yoktur ve beş filozoftan en fazla ⌊5/2⌋ = 2'si aynı anda yer; ancak F1 ve F3 sırayla yemeye devam ederse F2 teorik olarak aç kalabilir.

Tanenbaum Çözümü Adım Adım

```
1  void take_forks(int i) {
2    down(&mutex);
3    state[i] = HUNGRY;
4    test(i);
5    up(&mutex);
6    down(&s[i]);    /* çatal yoksa bloke */
7  }
8  void put_forks(int i) {
9    down(&mutex);
▶    state[i] = THINKING;
▶    test(LEFT);     /* (i+4) % 5 */
▶    test(RIGHT);    /* (i+1) % 5 */
13   up(&mutex);
14 }
15 void test(int i) {
16   if (state[i] == HUNGRY && state[LEFT] != EATING
17                          && state[RIGHT] != EATING) {
18     state[i] = EATING;
19     up(&s[i]);
20   }
21 }
```



F1 bırakır, komşularını test eder: `test(2)` başarısız çünkü sağdaki F3 hâlâ yiyor.

Adım 5 / 6

DEĞİŞKENLER

`state = TTHET`

Tanenbaum MOS Şekil 2-47. Kilitlenme yoktur ve beş filozoftan en fazla ⌊5/2⌋ = 2'si aynı anda yer; ancak F1 ve F3 sırayla yemeye devam ederse F2 teorik olarak aç kalabilir.

Tanenbaum Çözümü Adım Adım

```
1  void take_forks(int i) {
2    down(&mutex);
3    state[i] = HUNGRY;
4    test(i);
5    up(&mutex);
6    down(&s[i]);    /* çatal yoksa bloke */
7  }
8  void put_forks(int i) {
9    down(&mutex);
▶    state[i] = THINKING;
▶    test(LEFT);     /* (i+4) % 5 */
12   test(RIGHT);    /* (i+1) % 5 */
13   up(&mutex);
14 }
15 void test(int i) {
16   if (state[i] == HUNGRY && state[LEFT] != EATING
17                          && state[RIGHT] != EATING) {
▶      state[i] = EATING;
▶      up(&s[i]);
20   }
21 }
```



F3 bırakınca `test(2)` başarılı: F2 `EATING` olur ve `up(&s[2])` onu uyandırır.

Adım 6 / 6

DEĞİŞKENLER

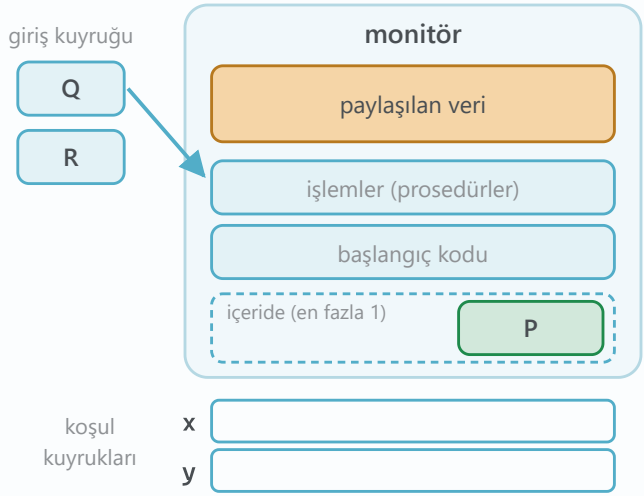
`state = TTETT` `s[2] = 0`

Tanenbaum MOS Şekil 2-47. Kilitlenme yoktur ve beş filozoftan en fazla ⌊5/2⌋ = 2'si aynı anda yer; ancak F1 ve F3 sırayla yemeye devam ederse F2 teorik olarak aç kalabilir.

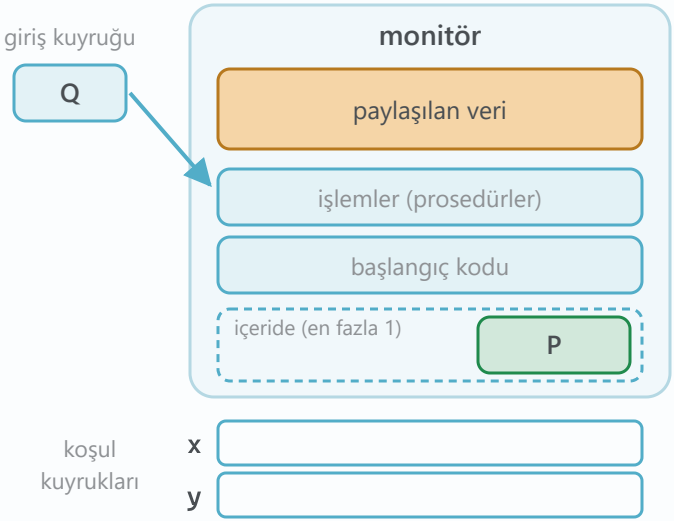
Monitör

- Hoare (1974) ve Brinch Hansen (1975): veri, prosedürler ve başlangıç kodunu bir arada tutan **dil düzeyinde** soyut veri tipi.
- Süreçler monitörün prosedürlerini çağırabilir ama iç verisine **doğrudan erişemez**.
- Aynı anda monitör içinde **yalnız bir süreç** etkin olabilir; giriş kilidini **derleyici** ekler, programcı unutamaz.
- Monitör dolu ise çağıran giriş kuyruğunda bekler. Bir koşulu beklemek için **koşul değişkenleri** kullanılır.
- Semaforda `down` sırasını karıştırmak kilitlenme yaratıyordu; monitör bu hata sınıfını azaltır.

```
monitor ornek
  integer i;
  condition c;
  procedure uretici(); … end;
  procedure tuketici(); … end;
end monitor;
```



Koşul Değişkenleri



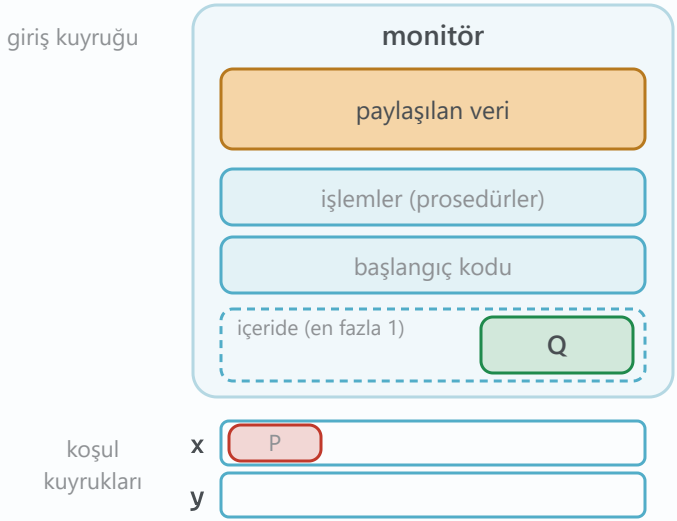
P monitörde; tampon boş olduğu için devam edemiyor. Q giriş kuyruğunda.

Adım 1 / 4

İşlem	Anlamı
`x.wait()`	Çağıranı askıya al, monitörü serbest bırak.
`x.signal()`	x'te bekleyen bir süreci (varsa) devam ettir.
`broadcast` / `notifyAll`	x'te bekleyen herkesi uyandır.

Koşul değişkeninin **değeri yoktur**; yalnızca bir bekleme kuyruğudur. Beklenen koşul (ör. `count > 0`) her zaman paylaşılan veride ayrıca tutulur.

Koşul Değişkenleri

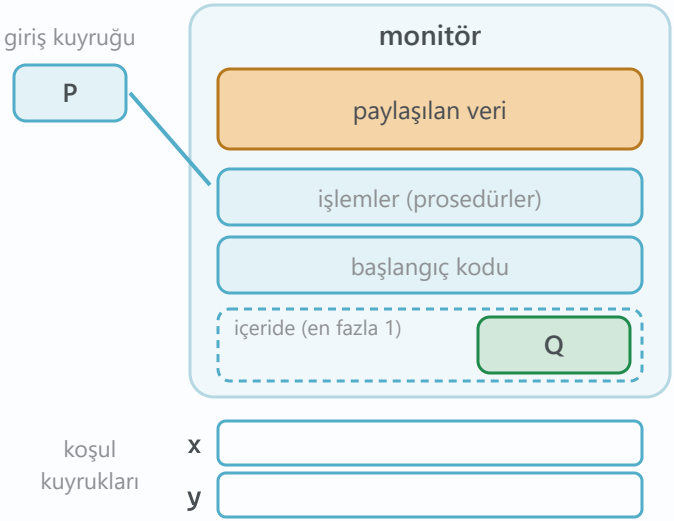


P `x.wait()` : **monitörü bırakır** ve x kuyruğunda uyur. Böylece Q girebilir.

İşlem	Anlamı
`x.wait()`	Çağıranı askıya al, monitörü serbest bırak.
`x.signal()`	x'te bekleyen bir süreci (varsa) devam ettir.
`broadcast` / `notifyAll`	x'te bekleyen herkesi uyandır.

Koşul değişkeninin **değeri yoktur**; yalnızca bir bekleme kuyruğudur. Beklenen koşul (ör. `count > 0`) her zaman paylaşılan veride ayrıca tutulur.

Koşul Değişkenleri

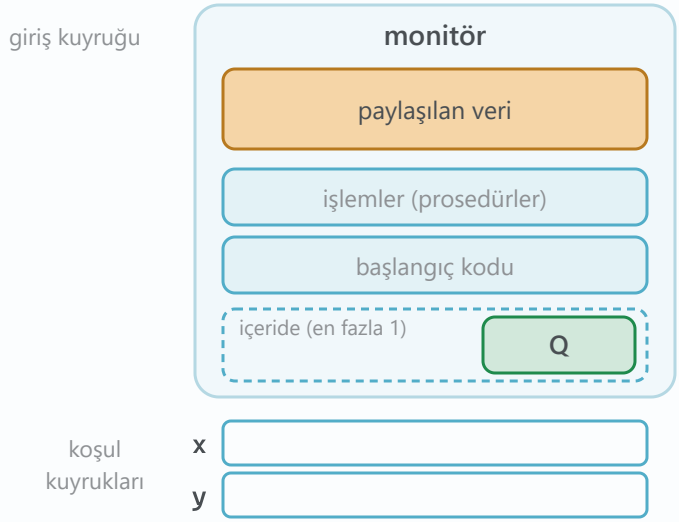


Q koşulu sağlar ve `x.signal()` çağırır: x kuyruğundan **bir** süreç uyanır. Kimin çalışacağı anlamsal bir seçimdir.

İşlem	Anlamı
`x.wait()`	Çağıranı askıya al, monitörü serbest bırak.
`x.signal()`	x'te bekleyen bir süreci (varsa) devam ettir.
`broadcast` / `notifyAll`	x'te bekleyen herkesi uyandır.

Koşul değişkeninin **değeri yoktur**; yalnızca bir bekleme kuyruğudur. Beklenen koşul (ör. `count > 0`) her zaman paylaşılan veride ayrıca tutulur.

Koşul Değişkenleri

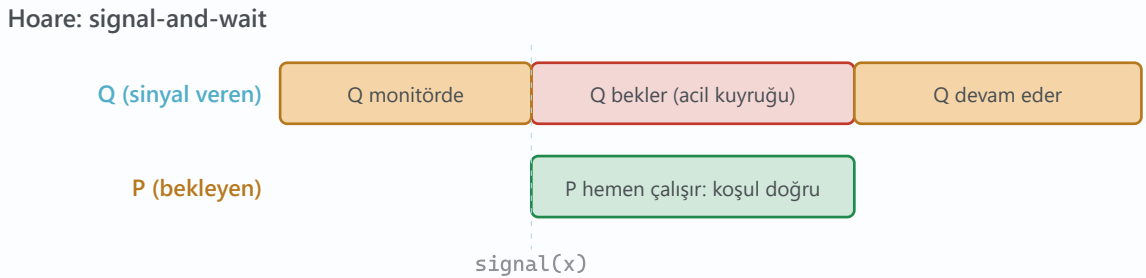


x kuyruğunda kimse yokken `x.signal()` **etkisizdir**, hiçbir iz bırakmaz. Semaforun `signal`'ı ise değeri artırır ve hatırlanır.

İşlem	Anlamı
`x.wait()`	Çağıranı askıya al, monitörü serbest bırak.
`x.signal()`	x'te bekleyen bir süreci (varsa) devam ettir.
`broadcast` / `notifyAll`	x'te bekleyen herkesi uyandır.

Koşul değişkeninin **değeri yoktur**; yalnızca bir bekleme kuyruğudur. Beklenen koşul (ör. `count > 0`) her zaman paylaşılan veride ayrıca tutulur.

Hoare ve Mesa Anlamı

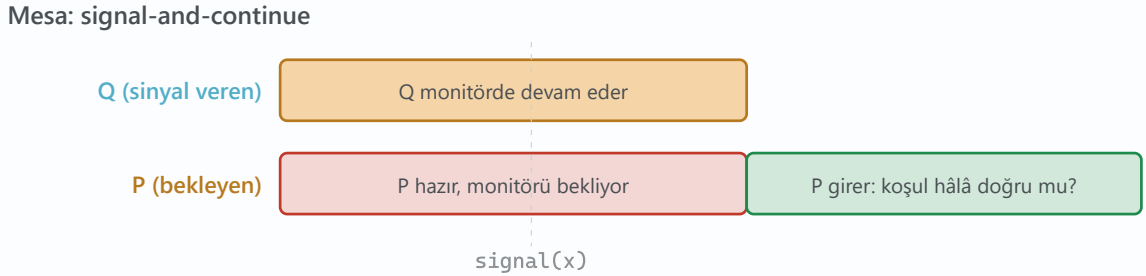


Signal-and-wait (Hoare): uyanan P **hemen** çalışır, Q bekler. P, koşulun doğru olduğundan emin olabilir: `if` yeterlidir.

Adım 1 / 2

Anlam	Kim hemen çalışır?	Kullananlar
Signal-and-wait (Hoare)	Uyandırılan	Ders kitabı monitörleri
Signal-and-exit (Brinch Hansen)	Uyandırılan; `signal` prosedürün son komutu olmalı	Concurrent Pascal
Signal-and-continue (Mesa)	Sinyal veren	Pthreads, Java, C#, Go `sync.Cond`, Linux çekirdeği

Hoare ve Mesa Anlamı



Signal-and-continue (Mesa, 1980): Q devam eder; P yalnızca "hazır" olur. P çalışana kadar başkası koşulu bozabilir: **yeniden kontrol şart**.

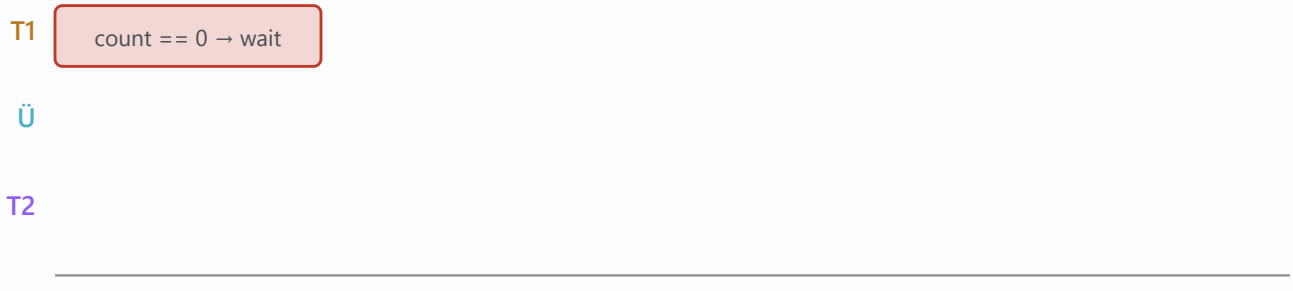
Adım 2 / 2

Anlam	Kim hemen çalışır?	Kullananlar
Signal-and-wait (Hoare)	Uyandırılan	Ders kitabı monitörleri
Signal-and-exit (Brinch Hansen)	Uyandırılan; `signal` prosedürün son komutu olmalı	Concurrent Pascal
Signal-and-continue (Mesa)	Sinyal veren	Pthreads, Java, C#, Go `sync.Cond`, Linux çekirdeği

Neden if Değil while?

```
/* YANLIŞ (Mesa'da) */
lock(&m);
if (count == 0)
  cond_wait(&dolu, &m);
x = al();        /* boş olabilir */
unlock(&m);
```

```
/* DOĞRU */
lock(&m);
while (count == 0)
  cond_wait(&dolu, &m);
x = al();        /* count > 0 kesin */
unlock(&m);
```



Tampon boş: tüketici T1 koşul değişkeninde bekliyor.

Neden if Değil while?

```
/* YANLIŞ (Mesa'da) */
lock(&m);
if (count == 0)
  cond_wait(&dolu, &m);
x = al();        /* boş olabilir */
unlock(&m);
```

```
/* DOĞRU */
lock(&m);
while (count == 0)
  cond_wait(&dolu, &m);
x = al();        /* count > 0 kesin */
unlock(&m);
```

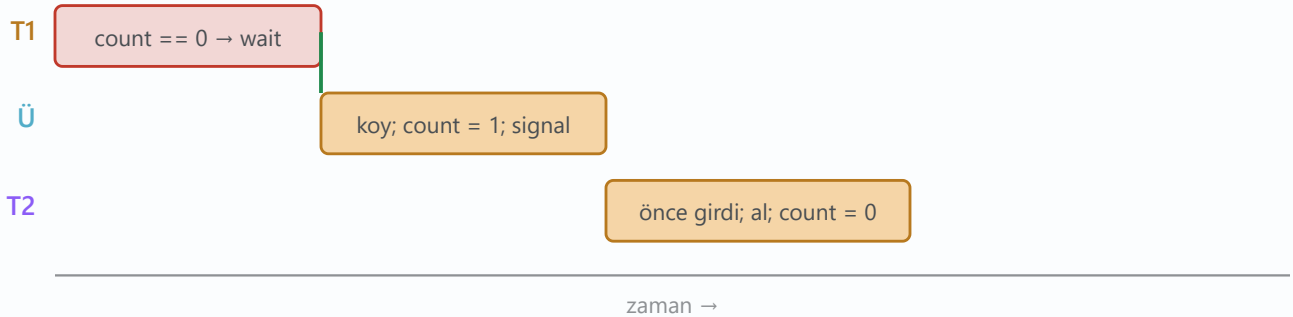


Üretici bir öğe koyar ve `signal` yapar. T1 uyanır ama önce kilidi geri alması gerekir.

Neden if Değil while?

```
/* YANLIŞ (Mesa'da) */
lock(&m);
if (count == 0)
  cond_wait(&dolu, &m);
x = al();        /* boş olabilir */
unlock(&m);
```

```
/* DOĞRU */
lock(&m);
while (count == 0)
  cond_wait(&dolu, &m);
x = al();        /* count > 0 kesin */
unlock(&m);
```

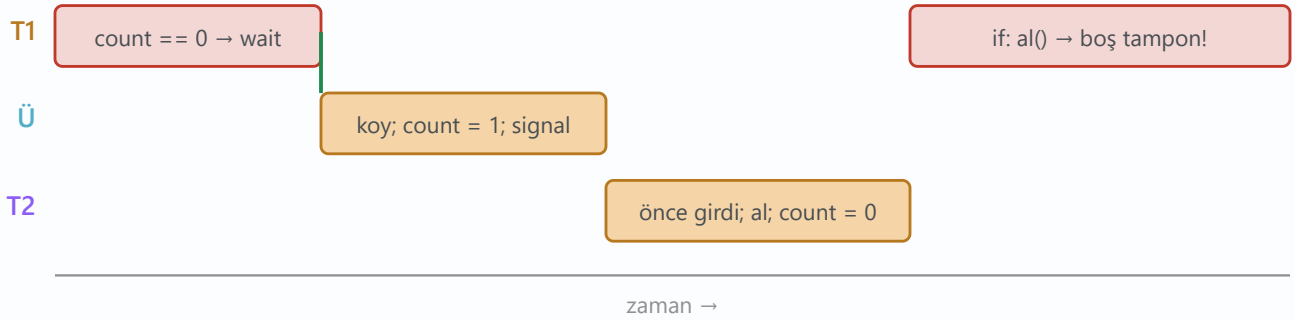


Bu arada yeni gelen T2 kilidi **önce** kapar (barging) ve öğeyi alır: `count = 0`.

Neden if Değil while?

```
/* YANLIŞ (Mesa'da) */
lock(&m);
if (count == 0)
  cond_wait(&dolu, &m);
x = al();        /* boş olabilir */
unlock(&m);
```

```
/* DOĞRU */
lock(&m);
while (count == 0)
  cond_wait(&dolu, &m);
x = al();        /* count > 0 kesin */
unlock(&m);
```

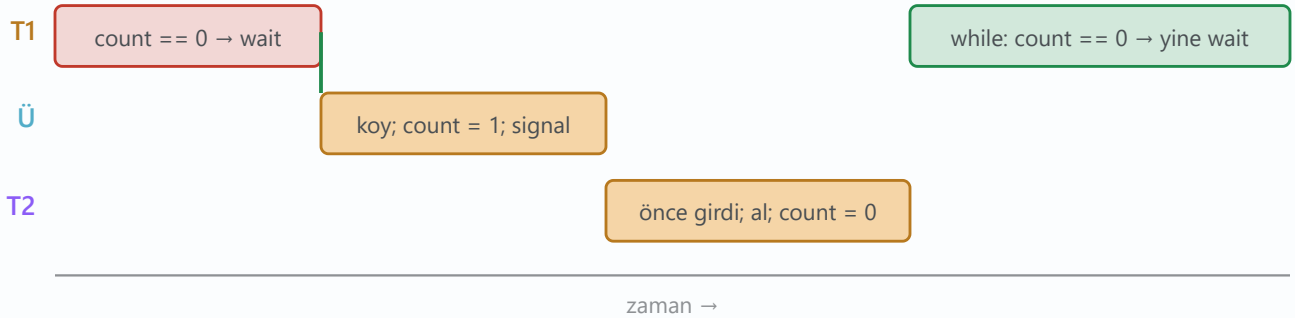


T1 sonunda çalışır. `if` ile koşulu yeniden sınamadan boş tampondan okur: **hata**.

Neden if Değil while?

```
/* YANLIŞ (Mesa'da) */
lock(&m);
if (count == 0)
  cond_wait(&dolu, &m);
x = al();        /* boş olabilir */
unlock(&m);
```

```
/* DOĞRU */
lock(&m);
while (count == 0)
  cond_wait(&dolu, &m);
x = al();        /* count > 0 kesin */
unlock(&m);
```



`while` ile T1 koşulu yeniden sınar ve tekrar bekler. POSIX ayrıca **sahte uyanmalara** (spurious wakeup) izin verir: `while` her durumda zorunludur.

Tek Koşul Değişkeni Yetmez

```
/* YANLIŞ: tek koşul c */
T: while (count == 0) wait(&c, &m);
   x = al();  signal(&c);
Ü: while (count == 1) wait(&c, &m);
   koy(x);    signal(&c);
```

```
/* DOĞRU: iki koşul */
T: while (count == 0) wait(&dolu, &m);
   x = al();  signal(&bos);
Ü: while (count == 1) wait(&bos, &m);
   koy(x);    signal(&dolu);
```



Tampon (boyut 1) boş: iki tüketici de aynı koşul değişkeni `c` 'de uyar.

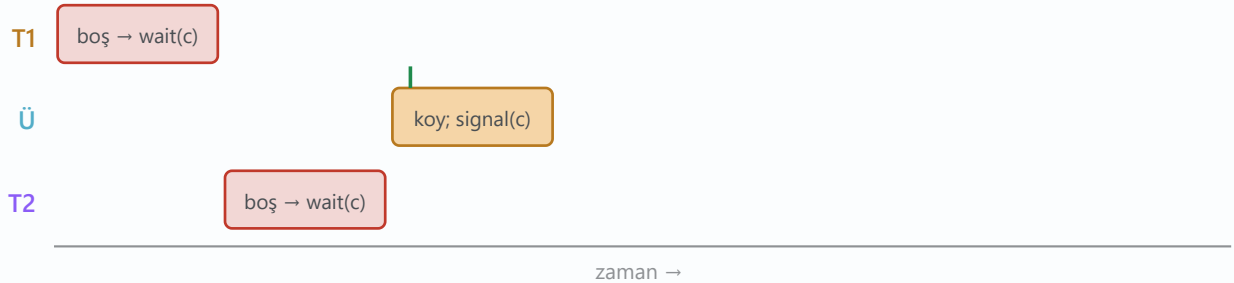
Adım 1 / 5

İki sürümde de kod `m` kilidi altında ve `while` ile çalışır. Hata, tüketicilerle üreticinin **aynı** kuyrukta beklemesidir: tüketici yalnız üreticiyi, üretici yalnız tüketiciyi uyandırmalıdır.

Tek Koşul Değişkeni Yetmez

```
/* YANLIŞ: tek koşul c */
T: while (count == 0) wait(&c, &m);
   x = al();  signal(&c);
Ü: while (count == 1) wait(&c, &m);
   koy(x);    signal(&c);
```

```
/* DOĞRU: iki koşul */
T: while (count == 0) wait(&dolu, &m);
   x = al();  signal(&bos);
Ü: while (count == 1) wait(&bos, &m);
   koy(x);    signal(&dolu);
```



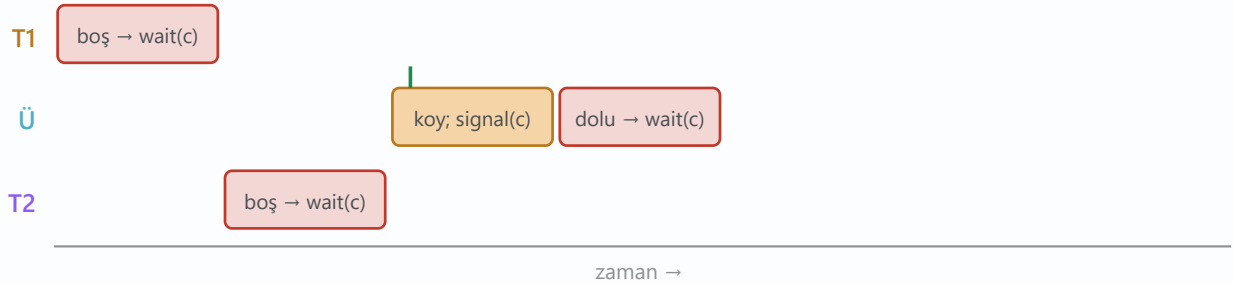
Üretici bir öğe koyar ve `signal(c)` ile T1'i uyandırır.

İki sürümde de kod `m` kilidi altında ve `while` ile çalışır. Hata, tüketicilerle üreticinin **aynı** kuyrukta beklemesidir: tüketici yalnız üreticiyi, üretici yalnız tüketiciyi uyandırmalıdır.

Tek Koşul Değişkeni Yetmez

```
/* YANLIŞ: tek koşul c */
T: while (count == 0) wait(&c, &m);
   x = al();  signal(&c);
Ü: while (count == 1) wait(&c, &m);
   koy(x);    signal(&c);
```

```
/* DOĞRU: iki koşul */
T: while (count == 0) wait(&dolu, &m);
   x = al();  signal(&bos);
Ü: while (count == 1) wait(&bos, &m);
   koy(x);    signal(&dolu);
```



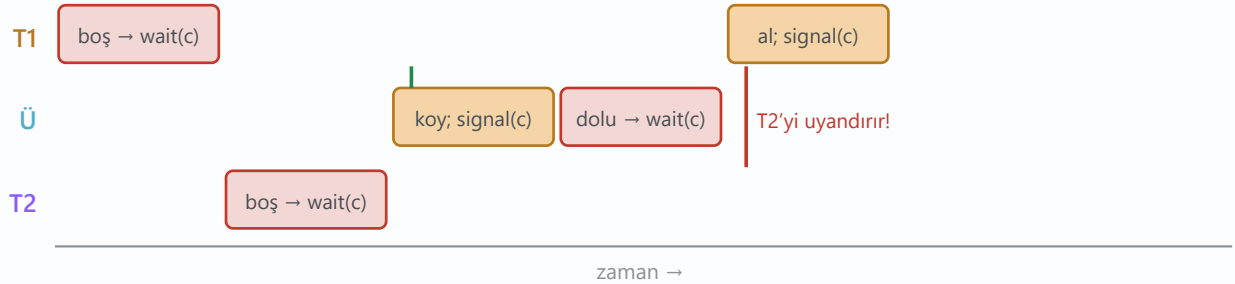
Üretici yeniden dener; tampon dolu olduğu için o da `c` 'de uyur. Artık `c` 'de T2 ve Ü bekliyor.

İki sürümde de kod `m` kilidi altında ve `while` ile çalışır. Hata, tüketicilerle üreticinin **aynı** kuyrukta beklemesidir: tüketici yalnız üreticiyi, üretici yalnız tüketiciyi uyandırmalıdır.

Tek Koşul Değişkeni Yetmez

```
/* YANLIŞ: tek koşul c */
T: while (count == 0) wait(&c, &m);
   x = al();  signal(&c);
Ü: while (count == 1) wait(&c, &m);
   koy(x);    signal(&c);
```

```
/* DOĞRU: iki koşul */
T: while (count == 0) wait(&dolu, &m);
   x = al();  signal(&bos);
Ü: while (count == 1) wait(&bos, &m);
   koy(x);    signal(&dolu);
```



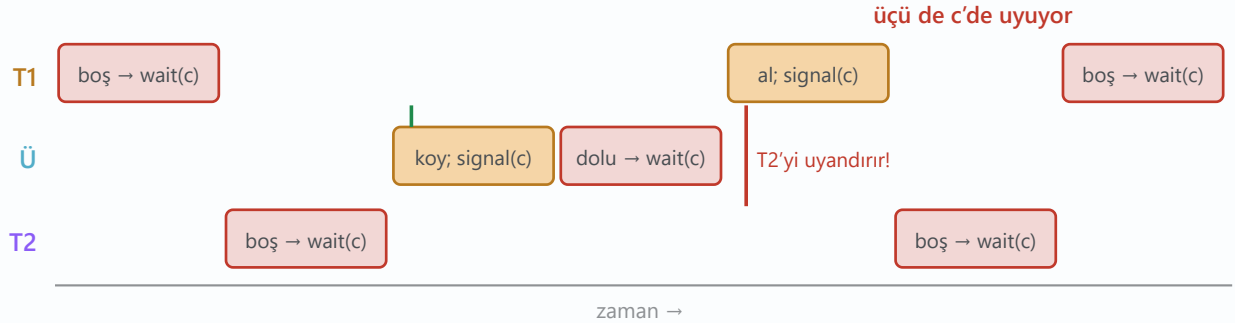
T1 öğeyi alır ve `signal(c)` yapar. Hangi bekleyenin uyanacağı belli değildir: bu kez **T2** uyanır.

İki sürümde de kod `m` kilidi altında ve `while` ile çalışır. Hata, tüketicilerle üreticinin **aynı** kuyrukta beklemesidir: tüketici yalnız üreticiyi, üretici yalnız tüketiciyi uyandırmalıdır.

Tek Koşul Değişkeni Yetmez

```
/* YANLIŞ: tek koşul c */
T: while (count == 0) wait(&c, &m);
   x = al();  signal(&c);
Ü: while (count == 1) wait(&c, &m);
   koy(x);    signal(&c);
```

```
/* DOĞRU: iki koşul */
T: while (count == 0) wait(&dolu, &m);
   x = al();  signal(&bos);
Ü: while (count == 1) wait(&bos, &m);
   koy(x);    signal(&dolu);
```



T2 tamponu boş bulur ve uyur; T1 de bir sonraki turda uyur. Üreticiyi uyandıracak kimse kalmadı: program durur.

İki sürümde de kod `m` kilidi altında ve `while` ile çalışır. Hata, tüketicilerle üreticinin **aynı** kuyrukta beklemesidir: tüketici yalnız üreticiyi, üretici yalnız tüketiciyi uyandırmalıdır.

Kimi Uyandırmalı? signal ve broadcast

```
int bos = 0;            /* boş bayt */

void bellek_al(int n) {
  lock(&m);
  while (bos < n)
    wait(&c, &m);
  bos -= n;  /* ... ayır */
  unlock(&m);
}
void bellek_ver(int n) {
  lock(&m);
  bos += n;
  broadcast(&c); /* signal yetmez */
  unlock(&m);
}
```

bos = 0 başlangıç

Ta 100 bayt ister bekliyor

Tb 10 bayt ister bekliyor

İkisi de c'de bekliyor.

c kuyruğu: Ta, Tb

Bellek tükenmiş: Ta 100, Tb 10 bayt ister; ikisi de `while` döngüsünde uyur.

Adım 1 / 3

Kapsayan koşul: hangi bekleyenin ilerleyebileceği bilinmiyorsa `pthread_cond_broadcast` (Java: `notifyAll`) kullanılır. Bekleyenler tek bir koşula ayrılabiliyorsa `signal` daha ucuzdur. Her iki durumda da bekleme `while` içinde olmalıdır.

Kimi Uyandırmalı? signal ve broadcast

```
int bos = 0;            /* boş bayt */

void bellek_al(int n) {
  lock(&m);
  while (bos < n)
    wait(&c, &m);
  bos -= n;  /* ... ayır */
  unlock(&m);
}
void bellek_ver(int n) {
  lock(&m);
  bos += n;
  broadcast(&c); /* signal yetmez */
  unlock(&m);
}
```

bos = 50 free(50) + signal

Ta 100 bayt ister
uyandı: 50 < 100 → yine bekler

Tb 10 bayt ister
uyandırılmadı, bekliyor

Tb ilerleyebilirdi ama uyuyor.

c kuyruğu: Ta, Tb

Tc 50 bayt bırakır. `signal` yalnız birini uyandırır; bu Ta olursa yine uyur ve Tb'nin hakkı boşa gider.

Kapsayan koşul: hangi bekleyenin ilerleyebileceği bilinmiyorsa `pthread_cond_broadcast` (Java: `notifyAll`) kullanılır. Bekleyenler tek bir koşula ayrılabiliyorsa `signal` daha ucuzdur. Her iki durumda da bekleme `while` içinde olmalıdır.

Kimi Uyandırmalı? signal ve broadcast

```
int bos = 0;            /* boş bayt */

void bellek_al(int n) {
  lock(&m);
  while (bos < n)
    wait(&c, &m);
  bos -= n;  /* ... ayır */
  unlock(&m);
}
void bellek_ver(int n) {
  lock(&m);
  bos += n;
  broadcast(&c); /* signal yetmez */
  unlock(&m);
}
```

bos = 40 free(50) + broadcast

Ta 100 bayt ister
uyandı: 40 < 100 → yine bekler

Tb 10 bayt ister
uyandı: 50 ≥ 10 → alır

Tb ilerledi; Ta boşuna uyandı.

c kuyruğu: Ta

`broadcast` herkesi uyandırır: her biri koşulunu yeniden sınar. Bedeli gereksiz uyanmalardır.

Kapsayan koşul: hangi bekleyenin ilerleyebileceği bilinmiyorsa `pthread_cond_broadcast` (Java: `notifyAll`) kullanılır. Bekleyenler tek bir koşula ayrılabiliyorsa `signal` daha ucuzdur. Her iki durumda da bekleme `while` içinde olmalıdır.

Monitörle Üretici–Tüketici

```
monitor ProducerConsumer
  condition full, empty;
  integer count;

  procedure insert(item: integer);
  begin
    if count = N then wait(full);
    insert_item(item);
    count := count + 1;
    if count = 1 then signal(empty)
  end;

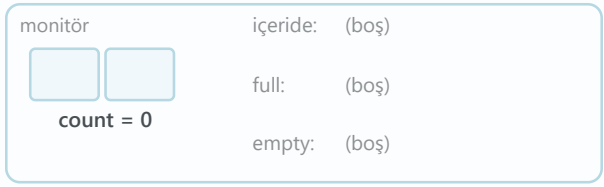
  function remove: integer;
  begin
    if count = 0 then wait(empty);
    remove := remove_item;
    count := count - 1;
    if count = N - 1 then signal(full)
  end;

  count := 0;
end monitor;
```

- Tanenbaum Şekil 2-34 (Pidgin Pascal). `count`'a erişim monitör tarafından zaten karşılıklı dışlanır.
- Tampon **doluysa üretici** `wait(full)` ile bloke olur ve monitörü bırakır; tüketici girip bir öğe alınca `signal(full)` ile üreticiyi uyandırır.
- Üreticiyi bloke eden `wait`'tir; `signal` yalnızca uyandırır.
- Burada `if` yeterlidir, çünkü Tanenbaum Brinch Hansen anlamını (signal son komut) varsayar. Pthreads/Java'da aynı kod `while` ile yazılmalıdır.
- Kayıp uyandırma olmaz: test ile `wait` aynı monitör içinde, bölünmeden yapılır.

Monitörle Üretici–Tüketici: Kod İzleme

```
1  monitor ProducerConsumer      { N = 2 }
2    condition full, empty; integer count;
3    procedure insert(item: integer);
4    begin
5      if count = N then wait(full);
6      insert_item(item); count := count + 1;
7      if count = 1 then signal(empty)
8    end;
9    function remove: integer;
10   begin
11     if count = 0 then wait(empty);
12     remove := remove_item; count := count - 1;
13     if count = N - 1 then signal(full)
14   end;
```



Tampon boş (`count = 0`), N = 2. Üretici Ü ve tüketici T monitöre sırayla girer.

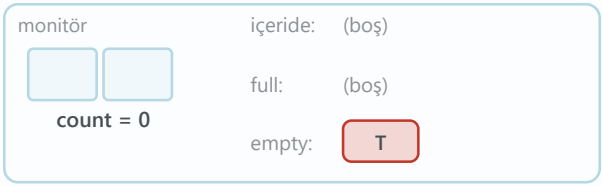
Adım 1 / 10

DEĞİŞKENLER

`count = 0` `tampon = []` `full = []` `empty = []` `T aldı = –`

Monitörle Üretici–Tüketici: Kod İzleme

```
1  monitor ProducerConsumer      { N = 2 }
2    condition full, empty; integer count;
3    procedure insert(item: integer);
4    begin
5      if count = N then wait(full);
6      insert_item(item); count := count + 1;
7      if count = 1 then signal(empty)
8    end;
9    function remove: integer;
10   begin
▶      if count = 0 then wait(empty);
12     remove := remove_item; count := count - 1;
13     if count = N - 1 then signal(full)
14   end;
```



T `remove` çağırır: count = 0 → `wait(empty)`. T uyur ve **monitörü bırakır**.

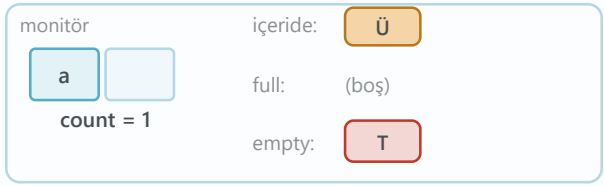
Adım 2 / 10

DEĞİŞKENLER

`count = 0` `tampon = []` `full = []` `empty = [T]` `T aldı = –`

Monitörle Üretici–Tüketici: Kod İzleme

```
1  monitor ProducerConsumer      { N = 2 }
2    condition full, empty; integer count;
3    procedure insert(item: integer);
4    begin
5      if count = N then wait(full);
6      insert_item(item); count := count + 1;
7      if count = 1 then signal(empty)
8    end;
9    function remove: integer;
10   begin
11     if count = 0 then wait(empty);
12     remove := remove_item; count := count - 1;
13     if count = N - 1 then signal(full)
14   end;
```



Ü `insert(a)` : tampon dolu değil, öğeyi koyar; count = 1.

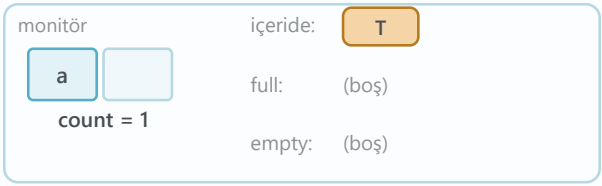
Adım 3 / 10

DEĞİŞKENLER

`count = 1` `tampon = [a]` `full = []` `empty = [T]` `T aldı = –`

Monitörle Üretici–Tüketici: Kod İzleme

```
1  monitor ProducerConsumer      { N = 2 }
2    condition full, empty; integer count;
3    procedure insert(item: integer);
4    begin
5      if count = N then wait(full);
6      insert_item(item); count := count + 1;
▶      if count = 1 then signal(empty)
8    end;
9    function remove: integer;
10   begin
11     if count = 0 then wait(empty);
12     remove := remove_item; count := count - 1;
13     if count = N - 1 then signal(full)
14   end;
```



count = 1 → `signal(empty)` : T uyanır. `signal` son komut olduğu için Ü hemen çıkar, monitörde iki süreç olmaz.

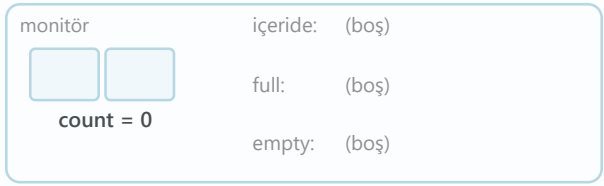
Adım 4 / 10

DEĞİŞKENLER

`count = 1` `tampon = [a]` `full = []` `empty = []` `T aldı = –`

Monitörle Üretici–Tüketici: Kod İzleme

```
1  monitor ProducerConsumer      { N = 2 }
2    condition full, empty; integer count;
3    procedure insert(item: integer);
4    begin
5      if count = N then wait(full);
6      insert_item(item); count := count + 1;
7      if count = 1 then signal(empty)
8    end;
9    function remove: integer;
10   begin
11     if count = 0 then wait(empty);
▶      remove := remove_item; count := count - 1;
▶      if count = N - 1 then signal(full)
14   end;
```



T kaldığı yerden sürer: `a`'yı alır, count = 0. 0 ≠ N − 1, sinyal yok.

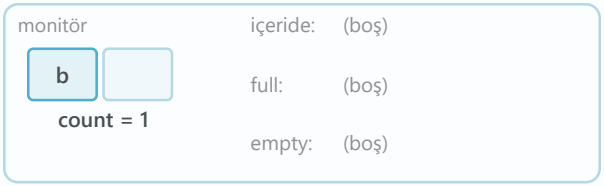
Adım 5 / 10

DEĞİŞKENLER

`count = 0` `tampon = []` `full = []` `empty = []` `T aldı = a`

Monitörle Üretici–Tüketici: Kod İzleme

```
1  monitor ProducerConsumer      { N = 2 }
2    condition full, empty; integer count;
3    procedure insert(item: integer);
4    begin
5      if count = N then wait(full);
▶      insert_item(item); count := count + 1;
▶      if count = 1 then signal(empty)
8    end;
9    function remove: integer;
10   begin
11     if count = 0 then wait(empty);
12     remove := remove_item; count := count - 1;
13     if count = N - 1 then signal(full)
14   end;
```



Ü `insert(b)` : count = 1 → `signal(empty)` , ama bekleyen yok: sinyal **kaybolur**, zararsızdır.

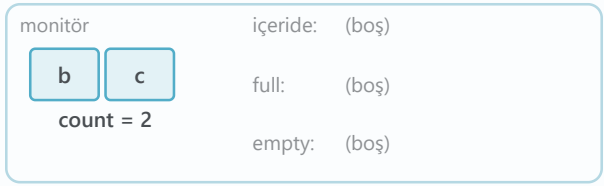
Adım 6 / 10

DEĞİŞKENLER

`count = 1` `tampon = [b]` `full = []` `empty = []` `T aldı = a`

Monitörle Üretici–Tüketici: Kod İzleme

```
1  monitor ProducerConsumer      { N = 2 }
2    condition full, empty; integer count;
3    procedure insert(item: integer);
4    begin
5      if count = N then wait(full);
▶      insert_item(item); count := count + 1;
7      if count = 1 then signal(empty)
8    end;
9    function remove: integer;
10   begin
11     if count = 0 then wait(empty);
12     remove := remove_item; count := count - 1;
13     if count = N - 1 then signal(full)
14   end;
```



Ü `insert(c)` : count = 2. Tampon doldu.

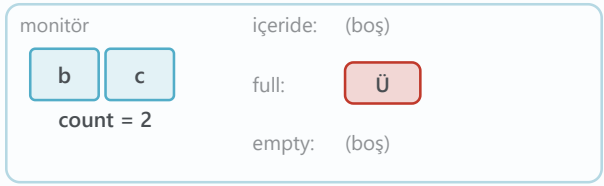
Adım 7 / 10

DEĞİŞKENLER

`count = 2` `tampon = [b c]` `full = []` `empty = []` `T aldı = a`

Monitörle Üretici–Tüketici: Kod İzleme

```
1  monitor ProducerConsumer      { N = 2 }
2    condition full, empty; integer count;
3    procedure insert(item: integer);
4    begin
▶      if count = N then wait(full);
6      insert_item(item); count := count + 1;
7      if count = 1 then signal(empty)
8    end;
9    function remove: integer;
10   begin
11     if count = 0 then wait(empty);
12     remove := remove_item; count := count - 1;
13     if count = N - 1 then signal(full)
14   end;
```



Ü `insert(d)` : count = N → `wait(full)` .
Ü uyur, monitör boşalır.

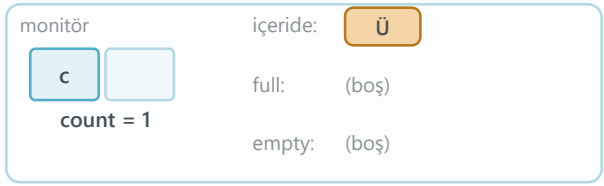
Adım 8 / 10

DEĞİŞKENLER

`count = 2` `tampon = [b c]`
`full = [Ü]` `empty = []`
`T aldı = a`

Monitörle Üretici–Tüketici: Kod İzleme

```
1  monitor ProducerConsumer      { N = 2 }
2    condition full, empty; integer count;
3    procedure insert(item: integer);
4    begin
5      if count = N then wait(full);
6      insert_item(item); count := count + 1;
7      if count = 1 then signal(empty)
8    end;
9    function remove: integer;
10   begin
11     if count = 0 then wait(empty);
▶      remove := remove_item; count := count - 1;
▶      if count = N - 1 then signal(full)
14   end;
```



T `remove` : `b` 'yi alır, count = 1 = N − 1 → `signal(full)` Ü'yü uyandırır.

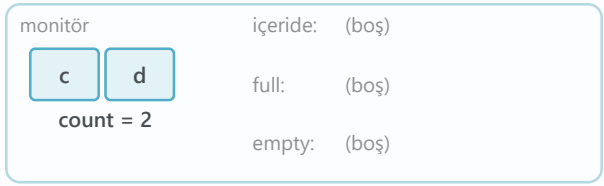
Adım 9 / 10

DEĞİŞKENLER

`count = 1` `tampon = [c]` `full = []` `empty = []` `T aldı = a b`

Monitörle Üretici–Tüketici: Kod İzleme

```
1  monitor ProducerConsumer      { N = 2 }
2    condition full, empty; integer count;
3    procedure insert(item: integer);
4    begin
5      if count = N then wait(full);
▶      insert_item(item); count := count + 1;
7      if count = 1 then signal(empty)
8    end;
9    function remove: integer;
10   begin
11     if count = 0 then wait(empty);
12     remove := remove_item; count := count - 1;
13     if count = N - 1 then signal(full)
14   end;
```



Ü `wait`'ten döner ve `d`'yi koyar: count = 2. Ne veri kayboldu ne süreç sonsuza dek uyudu.

Adım 10 / 10

DEĞİŞKENLER

`count = 2` `tampon = [c d]` `full = []` `empty = []` `T aldı = a b`

POSIX VE JAVA

Pthreads: Mutex ve Koşul Değişkeni

Mutex çağrısı	Tanım	Koşul çağrısı	Tanım
`pthread_mutex_init`	Mutex oluşturur	`pthread_cond_init`	Koşul değişkeni oluşturur
`pthread_mutex_destroy`	Mutex'i yok eder	`pthread_cond_destroy`	Koşul değişkenini yok eder
`pthread_mutex_lock`	Kilidi alır ya da bloke olur	`pthread_cond_wait`	Mutex'i **atomik olarak bırakıp** uyur; dönerken yeniden alır
`pthread_mutex_trylock`	Alır ya da hemen `EBUSY` döner	`pthread_cond_signal`	Bekleyenlerden **en az birini** uyandırır
`pthread_mutex_unlock`	Kilidi bırakır (yalnız sahibi)	`pthread_cond_broadcast`	Bekleyen **tümünü** uyandırır

Kural

`cond_wait` her zaman mutex tutulurken ve bir `while` döngüsü içinde çağrılır.

Mutex türleri

`NORMAL`, `ERRORCHECK`, `RECURSIVE`; öncelik kalıtımı için `PTHREAD_PRIO_INHERIT`.

Ek API'ler

`pthread_rwlock_*`, `pthread_spin_*`, `pthread_barrier_*`, C11 `mtx_t` / `cnd_t`.

Statik başlatma: `PTHREAD_MUTEX_INITIALIZER`, `PTHREAD_COND_INITIALIZER`. `cond_signal` yalnız o koşulda *bekleyenleri* etkiler; bekleyen yoksa sinyal kaybolur.

Pthreads Üretici–Tüketici Adım Adım

```
 1 pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
 2 pthread_cond_t condc = PTHREAD_COND_INITIALIZER,
 3                condp = PTHREAD_COND_INITIALIZER;
 4 int buffer = 0;                     /* 0 = boş */
 5
 6 void *producer(void *p) {
 7   for (int i = 1; i <= MAX; i++) {  /* MAX = 2 */
 8     pthread_mutex_lock(&m);
 9     while (buffer != 0) pthread_cond_wait(&condp, &m);
10     buffer = i;
11     pthread_cond_signal(&condc);
12     pthread_mutex_unlock(&m);
13   } return NULL;
14 }
15 void *consumer(void *p) {
16   for (int i = 1; i <= MAX; i++) {
▶      pthread_mutex_lock(&m);
18     while (buffer == 0) pthread_cond_wait(&condc, &m);
19     printf("%d\n", buffer); buffer = 0;
20     pthread_cond_signal(&condp);
21     pthread_mutex_unlock(&m);
22   } return NULL;
23 }
```

Tanenbaum & Bos, Şekil 2-32'den uyarlanmıştır.

Ü — başlamadı

T — lock(m)

mutex m: T

Tüketici önce çalışır ve mutex'i alır.

Adım 1 / 8

DEĞİŞKENLER

`buffer = 0`

ÇIKTI

Pthreads Üretici–Tüketici Adım Adım

```
 1 pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
 2 pthread_cond_t condc = PTHREAD_COND_INITIALIZER,
 3                condp = PTHREAD_COND_INITIALIZER;
 4 int buffer = 0;                     /* 0 = boş */
 5
 6 void *producer(void *p) {
 7   for (int i = 1; i <= MAX; i++) {  /* MAX = 2 */
 8     pthread_mutex_lock(&m);
 9     while (buffer != 0) pthread_cond_wait(&condp, &m);
10     buffer = i;
11     pthread_cond_signal(&condc);
12     pthread_mutex_unlock(&m);
13   } return NULL;
14 }
15 void *consumer(void *p) {
16   for (int i = 1; i <= MAX; i++) {
17     pthread_mutex_lock(&m);
▶    while (buffer == 0) pthread_cond_wait(&condc, &m);
19     printf("%d\n", buffer); buffer = 0;
20     pthread_cond_signal(&condp);
21     pthread_mutex_unlock(&m);
22   } return NULL;
23 }
```

Tanenbaum & Bos, Şekil 2-32'den uyarlanmıştır.

Ü — hazır

T — cond_wait(condc) → uyur

mutex m: boş

Tampon boş: `cond_wait` mutex'i **atomik olarak bırakır** ve uyur. Kayıp uyandırma penceresi yoktur.

Adım 2 / 8

DEĞİŞKENLER

`buffer = 0`

ÇIKTI

Pthreads Üretici–Tüketici Adım Adım

```
1  pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
2  pthread_cond_t condc = PTHREAD_COND_INITIALIZER,
3                 condp = PTHREAD_COND_INITIALIZER;
4  int buffer = 0;                    /* 0 = boş */
5
6  void *producer(void *p) {
7    for (int i = 1; i <= MAX; i++) {  /* MAX = 2 */
▶      pthread_mutex_lock(&m);
▶      while (buffer != 0) pthread_cond_wait(&condp, &m);
▶      buffer = i;
11     pthread_cond_signal(&condc);
12     pthread_mutex_unlock(&m);
13   } return NULL;
14 }
15 void *consumer(void *p) {
16   for (int i = 1; i <= MAX; i++) {
17     pthread_mutex_lock(&m);
18     while (buffer == 0) pthread_cond_wait(&condc, &m);
19     printf("%d\n", buffer); buffer = 0;
20     pthread_cond_signal(&condp);
21     pthread_mutex_unlock(&m);
22   } return NULL;
23 }
```

Tanenbaum & Bos, Şekil 2-32'den uyarlanmıştır.

Ü — lock, buffer = 1

T — uyuyor

mutex m: Ü

Üretici mutex'i alır; tampon boş olduğu için döngüyü atlar ve 1 yazar.

Adım 3 / 8

DEĞİŞKENLER

`buffer = 1`

ÇIKTI

Pthreads Üretici–Tüketici Adım Adım

```
1  pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
2  pthread_cond_t condc = PTHREAD_COND_INITIALIZER,
3                 condp = PTHREAD_COND_INITIALIZER;
4  int buffer = 0;                     /* 0 = boş */
5
6  void *producer(void *p) {
7    for (int i = 1; i <= MAX; i++) {  /* MAX = 2 */
8      pthread_mutex_lock(&m);
9      while (buffer != 0) pthread_cond_wait(&condp, &m);
10     buffer = i;
▶      pthread_cond_signal(&condc);
▶      pthread_mutex_unlock(&m);
13   } return NULL;
14 }
15 void *consumer(void *p) {
16   for (int i = 1; i <= MAX; i++) {
17     pthread_mutex_lock(&m);
18     while (buffer == 0) pthread_cond_wait(&condc, &m);
19     printf("%d\n", buffer); buffer = 0;
20     pthread_cond_signal(&condp);
21     pthread_mutex_unlock(&m);
22   } return NULL;
23 }
```

Tanenbaum & Bos, Şekil 2-32'den uyarlanmıştır.

Ü — signal(condc), unlock

T — uyandı, mutex bekliyor

mutex m: boş

`signal` tüketiciyi uyandırır ama tüketici **mutex'i geri almadan** dönemez (Mesa).

Adım 4 / 8

DEĞİŞKENLER

`buffer = 1`

ÇIKTI

Pthreads Üretici–Tüketici Adım Adım

```
 1 pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
 2 pthread_cond_t condc = PTHREAD_COND_INITIALIZER,
 3                condp = PTHREAD_COND_INITIALIZER;
 4 int buffer = 0;                     /* 0 = boş */
 5
 6 void *producer(void *p) {
 7   for (int i = 1; i <= MAX; i++) {  /* MAX = 2 */
 8     pthread_mutex_lock(&m);
 9     while (buffer != 0) pthread_cond_wait(&condp, &m);
10     buffer = i;
11     pthread_cond_signal(&condc);
12     pthread_mutex_unlock(&m);
13   } return NULL;
14 }
15 void *consumer(void *p) {
16   for (int i = 1; i <= MAX; i++) {
17     pthread_mutex_lock(&m);
18     while (buffer == 0) pthread_cond_wait(&condc, &m);
19     printf("%d\n", buffer); buffer = 0;
20     pthread_cond_signal(&condp);
21     pthread_mutex_unlock(&m);
22   } return NULL;
23 }
```

Tanenbaum & Bos, Şekil 2-32'den uyarlanmıştır.

Ü — i = 2: cond_wait(condp)

T — mutex bekliyor

mutex m: Ü

Üretici yarışı kazanıp mutex'i yeniden almış olabilir: tampon dolu → bekler ve mutex'i bırakır.

Adım 5 / 8

DEĞİŞKENLER

`buffer = 1`

ÇIKTI

Pthreads Üretici–Tüketici Adım Adım

```
 1 pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
 2 pthread_cond_t condc = PTHREAD_COND_INITIALIZER,
 3                condp = PTHREAD_COND_INITIALIZER;
 4 int buffer = 0;                    /* 0 = boş */
 5
 6 void *producer(void *p) {
 7   for (int i = 1; i <= MAX; i++) {  /* MAX = 2 */
 8     pthread_mutex_lock(&m);
 9     while (buffer != 0) pthread_cond_wait(&condp, &m);
10     buffer = i;
11     pthread_cond_signal(&condc);
12     pthread_mutex_unlock(&m);
13   } return NULL;
14 }
15 void *consumer(void *p) {
16   for (int i = 1; i <= MAX; i++) {
17     pthread_mutex_lock(&m);
▶    while (buffer == 0) pthread_cond_wait(&condc, &m);
▶    printf("%d\n", buffer); buffer = 0;
20     pthread_cond_signal(&condp);
21     pthread_mutex_unlock(&m);
22   } return NULL;
23 }
```

Tanenbaum & Bos, Şekil 2-32'den uyarlanmıştır.

Ü — uyuyor

T — while geçti, yazdı, buffer = 0

mutex m: T

Tüketici mutex'i alır, `while` koşulunu yeniden sınar (1 ≠ 0) ve öğeyi tüketir.

Adım 6 / 8

DEĞİŞKENLER

`buffer = 0`

ÇIKTI

```
1
```

Pthreads Üretici–Tüketici Adım Adım

```
 1 pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
 2 pthread_cond_t condc = PTHREAD_COND_INITIALIZER,
 3                condp = PTHREAD_COND_INITIALIZER;
 4 int buffer = 0;                    /* 0 = boş */
 5
 6 void *producer(void *p) {
 7   for (int i = 1; i <= MAX; i++) {  /* MAX = 2 */
 8     pthread_mutex_lock(&m);
▶      while (buffer != 0) pthread_cond_wait(&condp, &m);
▶      buffer = i;
11     pthread_cond_signal(&condc);
12     pthread_mutex_unlock(&m);
13   } return NULL;
14 }
15 void *consumer(void *p) {
16   for (int i = 1; i <= MAX; i++) {
17     pthread_mutex_lock(&m);
18     while (buffer == 0) pthread_cond_wait(&condc, &m);
19     printf("%d\n", buffer); buffer = 0;
▶      pthread_cond_signal(&condp);
▶      pthread_mutex_unlock(&m);
22   } return NULL;
23 }
```

Tanenbaum & Bos, Şekil 2-32'den uyarlanmıştır.

Ü — uyandı, buffer = 2

T — signal(condp), unlock

mutex m: Ü

Tüketici `condp`'ye sinyal verir; üretici uyanır, tamponu boş bulur ve 2 yazar.

Adım 7 / 8

DEĞİŞKENLER

`buffer = 2`

ÇIKTI

```
1
```

Pthreads Üretici–Tüketici Adım Adım

```
 1 pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
 2 pthread_cond_t condc = PTHREAD_COND_INITIALIZER,
 3                condp = PTHREAD_COND_INITIALIZER;
 4 int buffer = 0;                     /* 0 = boş */
 5
 6 void *producer(void *p) {
 7   for (int i = 1; i <= MAX; i++) {  /* MAX = 2 */
 8     pthread_mutex_lock(&m);
 9     while (buffer != 0) pthread_cond_wait(&condp, &m);
10     buffer = i;
11     pthread_cond_signal(&condc);
12     pthread_mutex_unlock(&m);
13   } return NULL;
14 }
15 void *consumer(void *p) {
16   for (int i = 1; i <= MAX; i++) {
▶      pthread_mutex_lock(&m);
▶      while (buffer == 0) pthread_cond_wait(&condc, &m);
▶      printf("%d\n", buffer); buffer = 0;
20     pthread_cond_signal(&condp);
21     pthread_mutex_unlock(&m);
22   } return NULL;
23 }
```

Tanenbaum & Bos, Şekil 2-32'den uyarlanmıştır.

Ü — bitti

T — 2'yi yazdı

mutex m: T

İkinci tur: tüketici 2'yi alır. Hangi sıra olursa olsun çıktı aynıdır.

Adım 8 / 8

DEĞİŞKENLER

`buffer = 0`

ÇIKTI

```
1
2
```

POSIX Semaforları

```c
#include <semaphore.h>

/* adsız: aynı sürecin iş parçacıkları */
sem_t bos, dolu;
sem_init(&bos,  0, N);   /* 0: süreç içi */
sem_init(&dolu, 0, 0);
sem_wait(&bos);   /* … koy … */   sem_post(&dolu);
sem_destroy(&bos);

/* adlandırılmış: ilgisiz süreçler arası */
sem_t *s = sem_open("/os06", O_CREAT, 0644, 1);
sem_wait(s);  /* KB */  sem_post(s);
sem_close(s);
sem_unlink("/os06");
```

`sem_wait`	P / down: 0 ise bloke
`sem_trywait`, `sem_timedwait`	Beklemeden ya da süre sınırıyla dene
`sem_post`	V / up; sinyal işleyicisinden çağrılabilir
`sem_getvalue`	Anlık değer (yalnız bilgi amaçlı)

- `sem_init`'in ikinci argümanı 1 ise semafor süreçler arası paylaşımlı bellekte olmalıdır.
- macOS adsız semaforları desteklemez (`sem_init` kullanımdan kalktı); adlandırılmış semafor ya da `dispatch_semaphore` kullanılır.
- Java: `java.util.concurrent.Semaphore` — `acquire()`, `release()`, isteğe bağlı adil mod.

Java: synchronized, wait ve notify

```
public class TekYuva {
  private int yuva = 0;             /* 0 = boş */

  synchronized void koy(int x) throws InterruptedException {
    while (yuva != 0) wait();       /* kilidi bırakır */
    yuva = x;
    notifyAll();
  }
  synchronized int al() throws InterruptedException {
    while (yuva == 0) wait();
    int x = yuva;
    yuva = 0;
    notifyAll();
    return x;
  }
  /* main: üretici 10, 20, 30, 40 koyar;
     main dört kez al() ile okuyup yazdırır */
}
```

- Her Java nesnesi bir **monitördür**: içsel kilit + tek bir bekleme kümesi (wait set).
- `synchronized` metot/blok kilidi alır, çıkışta (istisnada bile) bırakır; kilit **yeniden girilebilir**.
- `wait()` kilidi bırakıp uyur; `notify()` birini, `notifyAll()` hepsini uyandırır. Anlam **Mesa**'dır: `while` şart.
- Üretici ve tüketici aynı bekleme kümesini paylaştığı için `notifyAll` daha güvenlidir.

ÇIKTI (JDK 21, HER ÇALIŞTIRMADA AYNI)

```
alındı 10
alındı 20
alındı 30
alındı 40
```

Java: ReentrantLock ve Condition

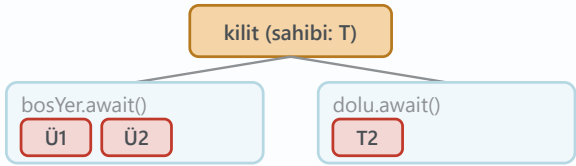
```
class Tampon {
  private final int[] buf = new int[2];
  private int bas = 0, son = 0, adet = 0;
  private final ReentrantLock kilit = new ReentrantLock();
  private final Condition bosYer = kilit.newCondition();
  private final Condition dolu   = kilit.newCondition();

  void koy(int x) throws InterruptedException {
    kilit.lock();
    try {
      while (adet == buf.length) bosYer.await();
      buf[son] = x; son = (son + 1) % buf.length; adet++;
      dolu.signal();
    } finally { kilit.unlock(); }
  }
  /* al(): while (adet == 0) dolu.await(); …
           bosYer.signal();                    */
}
/* üretici 1..5 koyar; main 5 kez al() */
```

- `java.util.concurrent.locks` : bir kilide **birden çok koşul** bağlanabilir; `signal` doğru grubu uyandırır.
- `tryLock()` , süre sınırlı ve kesilebilir kilitleme, isteğe bağlı **adil** mod.
- `unlock()` mutlaka `finally` içinde.
- Java 21 sanal iş parçacıkları `synchronized` içinde bloke olunca taşıyıcı iş parçacığını "sabitliyordu"; JDK 24 (JEP 491) bunu giderdi.

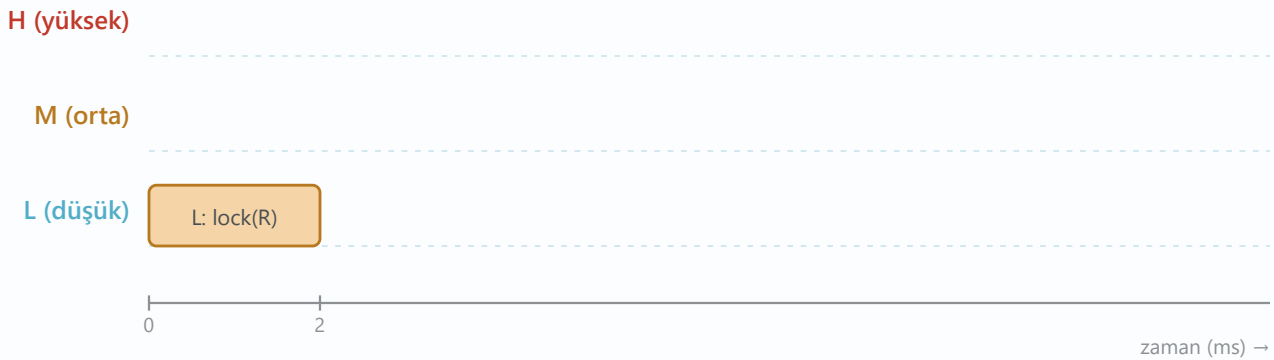
ÇIKTI (JDK 21, ÜÇ ÇALIŞTIRMA)

```
1 2 3 4 5 | toplam = 15
1 2 3 4 5 | toplam = 15
1 2 3 4 5 | toplam = 15
```



CANLILIK

Öncelik Terslemesi

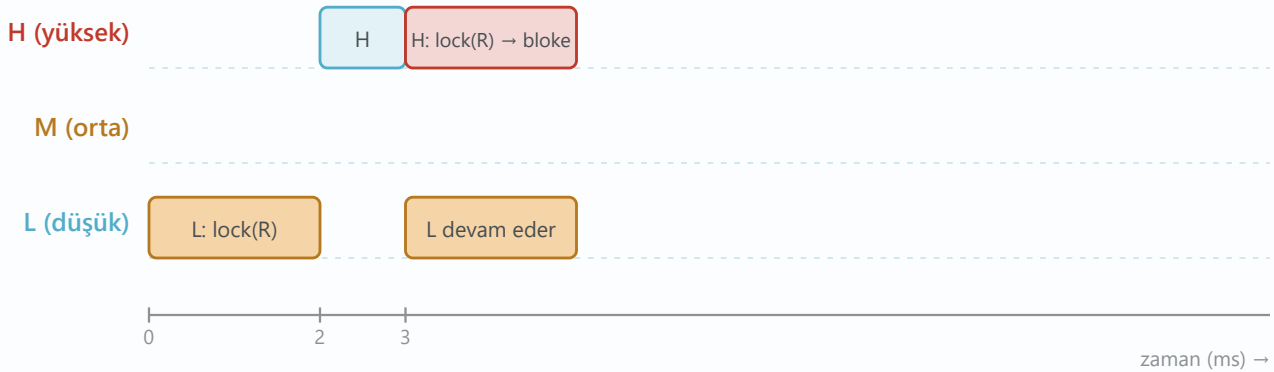


t = 0: düşük öncelikli L, paylaşılan kaynak R'nin kilidini alır.

Adım 1 / 4

Öncelik terslemesi (priority inversion) bir çözüm değil, **sorundur**; çözüm öncelik **kalıtımı** (inheritance) ya da öncelik **tavanıdır** (ceiling). Spinlock'la meşgul bekleyen H, tek çekirdekte L'yi hiç çalıştırmaz: sistem kilitlenir.

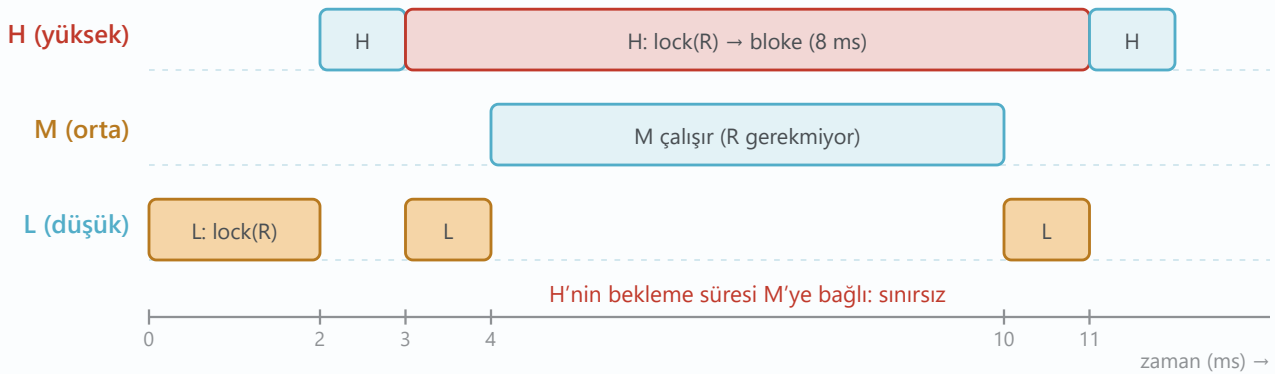
Öncelik Terslemesi



t = 2: yüksek öncelikli H gelir, L'yi keser; t = 3'te R'yi ister ve **bloke olur**. Kısa bir bekleme beklenir: L çıkınca H girer.

Öncelik terslemesi (priority inversion) bir çözüm değil, **sorundur**; çözüm öncelik **kalıtımı** (inheritance) ya da öncelik **tavanıdır** (ceiling). Spinlock'la meşgul bekleyen H, tek çekirdekte L'yi hiç çalıştırmaz: sistem kilitlenir.

Öncelik Terslemesi

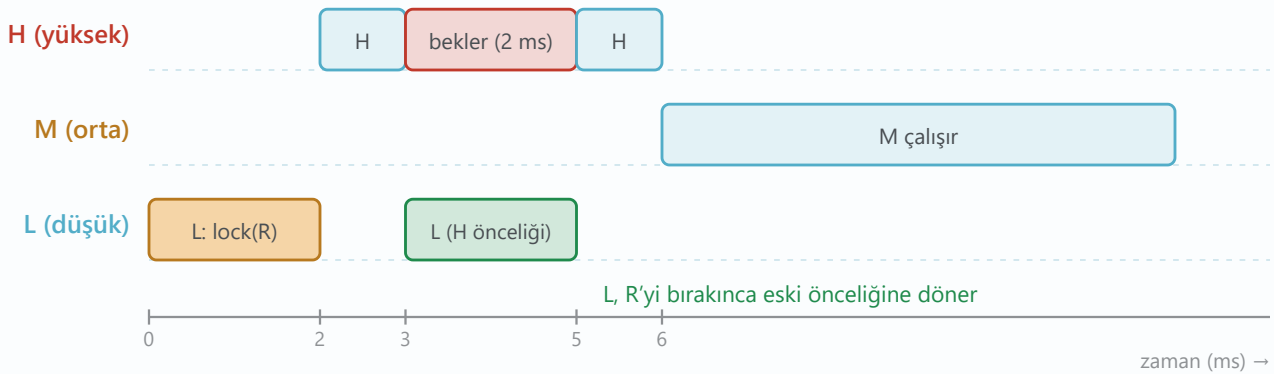


Ama R ile ilgisi olmayan orta öncelikli M, L'yi keser ve uzun süre çalışır. H dolaylı olarak **M'yi bekler**: **öncelik terslemesi**. M t = 4–10 arasında çalışırsa L, R'yi ancak t = 11'de bırakır: H **8 ms** bekler ve M uzadıkça bu süre sınırsız uzar.

Öncelik terslemesi (priority inversion) bir çözüm değil, **sorundur**; çözüm öncelik **kalıtımı** (inheritance) ya da öncelik **tavanıdır** (ceiling). Spinlock'la meşgul bekleyen H, tek çekirdekte L'yi hiç çalıştırmaz: sistem kilitlenir.

CANLILIK

Öncelik Terslemesi



Öncelik kalıtımı: H, R'yi beklerken L geçici olarak H'nin önceliğini alır; M araya giremez. L çıkar, H hemen çalışır: aynı örnekte H yalnız **2 ms** bekler.

Adım 4 / 4

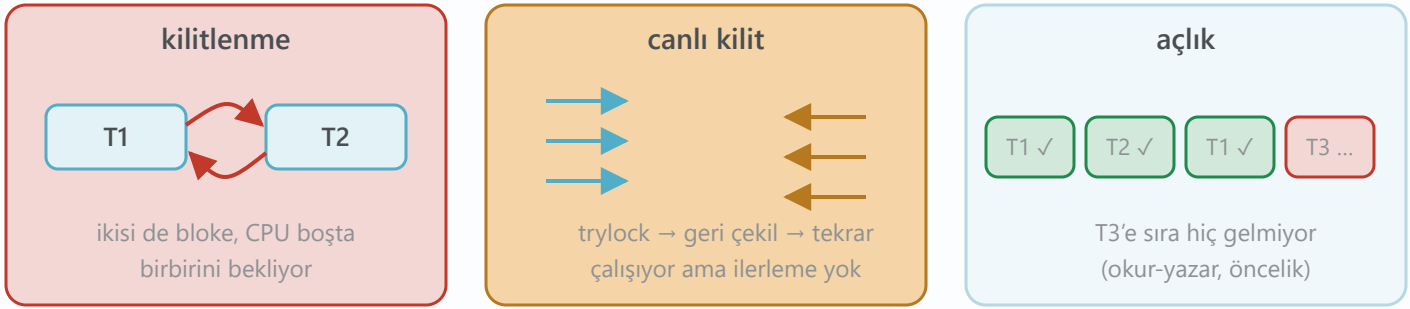
Öncelik terslemesi (priority inversion) bir çözüm değil, **sorundur**; çözüm öncelik **kalıtımı** (inheritance) ya da öncelik **tavanıdır** (ceiling). Spinlock'la meşgul bekleyen H, tek çekirdekte L'yi hiç çalıştırmaz: sistem kilitlenir.

Mars Pathfinder (1997)

- Temmuz 1997: Mars yüzeyindeki Pathfinder, VxWorks gerçek zamanlı işletim sisteminde çalışırken **tekrar tekrar kendini sıfırladı**.
- Düşük öncelikli meteoroloji görevi veri yolu mutex'ini tutarken yüksek öncelikli veri yolu yönetim görevi bloke oldu; orta öncelikli iletişim görevi araya girdi.
- Yönetim görevi zamanında çalışamayınca **bekçi zamanlayıcı** "sistem asılı kaldı" deyip tam sıfırlama yaptı.
- JPL mühendisleri hatayı yerdeki kopyada üretti; mutex'in **öncelik kalıtımı** seçeneği uzaktan yüklenen bir yamayla açıldı.
- Bugün: `PTHREAD_PRIO_INHERIT`, Linux PI-futex ve `rt_mutex`; Linux 6.12'de ana dala giren PREEMPT_RT bu mekanizmaya dayanır.



Canlılık Sorunları



Kilitlenme (deadlock)

Bir küme sürecin her biri, kümedeki başka birinin tuttuğu kaynağı bekler. Dört gerekli koşul aynı anda sağlanınca oluşur.

Canlı kilit (livelock)

Süreçler bloke değildir; birbirine tepki verip durmadan aynı adımları tekrarlar. Çare: rastgele geri çekilme (ör. Ethernet).

Açlık (starvation)

Bir süreç sürekli başkalarına yol verir. Çare: FIFO kuyruk, adil kilit, yaşlandırma. Sınırlı bekleme koşulu bunu dışlar.

Kilitsiz (Lock-free) Yapılara Giriş

```
void push(node_t *n) {
  node_t *eski = atomic_load(&top);
  do {
    n->next = eski;
  } while (!atomic_compare_exchange_weak(
              &top, &eski, n));
}  /* başarısızsa eski = güncel top */
```

Bloke eden	Kilidi tutan durursa herkes durur
Kilitsiz	Sistem bütün olarak her zaman ilerler
Beklemesiz	Her iş parçacığı sınırlı adımda biter



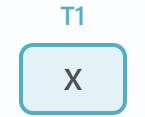
Yığın: `top → A → B`. Kilit yerine tek bir atomik işaretçi.

Kilitsiz (Lock-free) Yapılara Giriş

```
void push(node_t *n) {
  node_t *eski = atomic_load(&top);
  do {
    n->next = eski;
  } while (!atomic_compare_exchange_weak(
              &top, &eski, n));
}  /* başarısızsa eski = güncel top */
```

Bloke eden	Kilidi tutan durursa herkes durur
Kilitsiz	Sistem bütün olarak her zaman ilerler
Beklemesiz	Her iş parçacığı sınırlı adımda biter





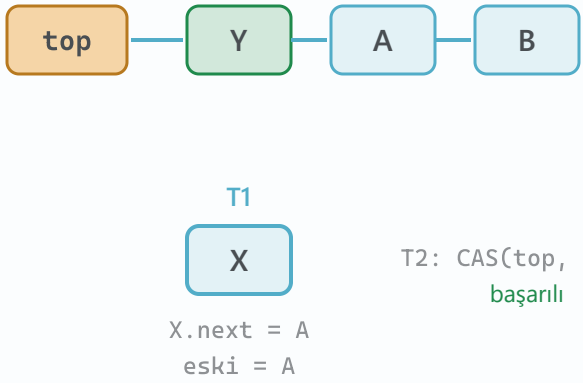
`X.next = A`
`eski = A`

T1, X'i eklemeye hazırlanır: `eski = A`, `X.next = A`.

Kilitsiz (Lock-free) Yapılara Giriş

```
void push(node_t *n) {
  node_t *eski = atomic_load(&top);
  do {
    n->next = eski;
  } while (!atomic_compare_exchange_weak(
              &top, &eski, n));
}  /* başarısızsa eski = güncel top */
```

Bloke eden	Kilidi tutan durursa herkes durur
Kilitsiz	Sistem bütün olarak her zaman ilerler
Beklemesiz	Her iş parçacığı sınırlı adımda biter

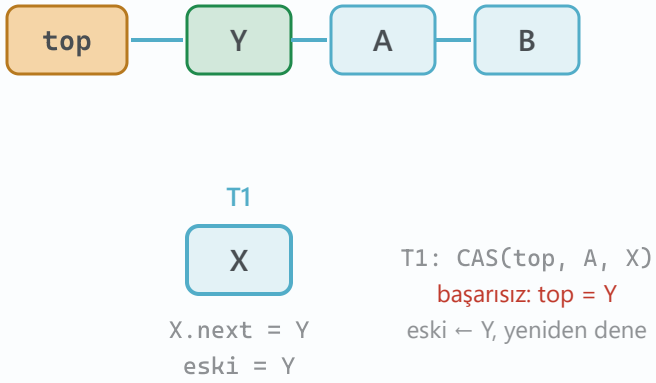


Bu arada T2, Y'yi ekler: `CAS(top, A, Y)` başarılı.

Kilitsiz (Lock-free) Yapılara Giriş

```
void push(node_t *n) {
  node_t *eski = atomic_load(&top);
  do {
    n->next = eski;
  } while (!atomic_compare_exchange_weak(
              &top, &eski, n));
}  /* başarısızsa eski = güncel top */
```

Bloke eden	Kilidi tutan durursa herkes durur
Kilitsiz	Sistem bütün olarak her zaman ilerler
Beklemesiz	Her iş parçacığı sınırlı adımda biter



T1'in CAS'ı başarısız olur (top artık Y). C11 `compare_exchange` güncel değeri `eski`'ye yazar; T1 `X.next = Y` ile yeniden dener.

Kilitsiz (Lock-free) Yapılara Giriş

```
void push(node_t *n) {
  node_t *eski = atomic_load(&top);
  do {
    n->next = eski;
  } while (!atomic_compare_exchange_weak(
              &top, &eski, n));
}  /* başarısızsa eski = güncel top */
```

Bloke eden	Kilidi tutan durursa herkes durur
Kilitsiz	Sistem bütün olarak her zaman ilerler
Beklemesiz	Her iş parçacığı sınırlı adımda biter



T1: CAS(top, Y, X) başarılı

kilit yok; bir iş parçacığı askıda kalsa da diğerleri ilerlemeye devam eder

İkinci CAS başarılı. **ABA sorunu**: `pop`'ta top A→B→A değişirse CAS bunu fark etmez; sürüm sayacı ya da güvenli bellek geri kazanımı (hazard pointer, epoch) gerekir.

Özet

Araç	Ne sağlar?	Ne zaman?
Peterson, katı sıra	Yalnız okuma/yazma ile dışlama (SC varsayımıyla)	Kavramsal; gerçek kodda kullanılmaz
Bellek bariyeri	Okuma/yazma sırası ve görünürlük	Kilit ve atomik gerçeklemelerinin içinde
TAS, XCHG, CAS	Atomik okuma-değiştirme-yazma	Kilitlerin ve kilitsiz yapıların temeli
Atomik değişken	Tek değişkenin bölünmez güncellemesi	Sayaç, bayrak, referans sayımı
Spinlock	Dışlama, uyumadan	Çok çekirdek, çok kısa KB, çekirdek içi
Mutex	Dışlama, bekleyen uyur	Genel amaçlı kritik bölgeler
Semafor	Sayma, dışlama ve sıralama	N özdeş kaynak, olay bildirimi
Monitör + koşul değişkeni	Yapısal dışlama + koşul bekleme	Karmaşık koşullar (Java, pthreads)
Okur-yazar kilidi	Çoklu okur, tek yazar	Okuma ağırlıklı veriler

Her araç *karşılıklı dışlama*yı sağlar; *ilerleme* ve *sınırlı bekleme* doğru kullanım ve adil kuyruklarla gelir. Öncelik terslemesi ve kilitlenme ayrıca düşünülmelidir.

Alıştırmalar

Ebeveyn `bekle()`, çocuk iş parçacığı işi bitince `bitir()` çağırır.

```
/* (a) durum değişkeni yok */
void bitir() { lock(&m);
  signal(&c); unlock(&m); }
void bekle() { lock(&m);
  wait(&c, &m); unlock(&m); }
```

```
/* (b) kilit yok, if var */
void bitir() { bitti = 1;
  signal(&c); }
void bekle() {
  if (bitti == 0) wait(&c); }
```

1. (a)'da çocuk, ebeveyn `bekle()`'ye girmeden önce bitirirse ne olur?
2. (b)'de ebeveynin sonsuza dek uyuduğu bir sıralama yazın.
3. Yalnız bir mutex ve bir koşul değişkeniyle `sem_bekle` / `sem_ver` gerçekleştirin.
4. 8 çekirdekli yaklaşık sayaçta S = 1024 ise `oku()` gerçek değerden en fazla ne kadar geride kalır? S = 1 olsaydı?

▶ **Cevaplar**

ÖZET

Kontrol Soruları (1/2)

1. Kritik bölge çözümünün üç koşulunu yazın. Katı sırayla değişim hangisini ihlal eder?
2. Peterson algoritması x86 üzerinde neden yanlış çalışabilir? Nasıl düzeltilir?
3. `test_and_set` ile kurulan basit spinlock hangi koşulu garanti etmez?
4. CAS ile artırmada iki iş parçacığı aynı değeri okursa ne olur? Son değer neden doğrudur?

▶ **Cevaplar**

ÖZET

Kontrol Soruları (2/2)

5. Başlangıç değeri 2 olan bir semaforda üç süreç art arda `wait` yaparsa değer ve kuyruk ne olur (Silberschatz gerçeklemesi)?
6. Sınırlı tampon çözümünde üreticideki iki `down`'ın yeri değişirse ne olur?
7. Mesa anlamında koşul değişkeni beklemesi neden `while` içinde yapılmalıdır?
8. Öncelik terslemesini tanımlayın; öncelik kalıtımı onu nasıl çözer?

▶ **Cevaplar**

ÖZET

Kaynaklar

- A. Silberschatz, P. B. Galvin, G. Gagne, *Operating System Concepts*, 10. baskı, Wiley, 2018 — Bölüm 6: Synchronization Tools, Bölüm 7: Synchronization Examples.
- A. S. Tanenbaum, H. Bos, *Modern Operating Systems*, 4. baskı (2014) / 5. baskı (2023), Pearson — §2.3 Interprocess Communication, §2.5 Classical IPC Problems.
- R. H. Arpaci-Dusseau, A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces* (OSTEP), ostep.org — "Locks", "Condition Variables", "Semaphores", "Common Concurrency Problems".
- A. B. Downey, *The Little Book of Semaphores*, 2. baskı, Green Tea Press (ücretsiz) — okur-yazar ve filozof çözümleri.
- POSIX.1-2017 (IEEE Std 1003.1): `pthread_mutex_lock`, `pthread_cond_wait`, `sem_overview(7)`; `futex(2)` Linux kılavuz sayfası.
- B. Goetz vd., *Java Concurrency in Practice*, 2006; Java SE 21 API: `java.util.concurrent`; JEP 491.
- G. L. Peterson, "Myths About the Mutual Exclusion Problem", *IPL*, 1981; M. Jones, "What really happened on Mars?", 1997.

Son

Bölüm 6: Senkronizasyon

Sonraki bölüm: Kilitlenme · Terim sözlüğü