Bölüm 7: Kilitlenme

İşletim Sistemleri

Kaynaklar | Dört Coffman koşulu | Kaynak tahsis çizgesi | Bekleme çizgesi | Devekuşu algoritması | Tespit algoritması | Kurtarma | Güvenli durumlar | Bankacı algoritması | Önleme ve kilit sıralaması | 2PL, canlı kilit, açlık | Java, jstack, lockdep

Sercan KÜLCÜ · Giresun Üniversitesi · Bilgisayar Mühendisliği

© Sercan Külcü. Tüm hakları saklıdır. · Sunumun etkileşimli HTML sürümü: https://sercankulcu.github.io/files/os/slides/Bolum_07_Kilitlenme.html

GİRİŞ

Öğrenme Çıktıları

Tanım ve kaynaklar
Kilitlenmeyi tanımlamak; kesilebilir ve kesilemez kaynakları, istek–kullanım–bırakma döngüsünü ayırt etmek.

Dört koşul
Coffman koşullarını (karşılıklı dışlama, tut ve bekle, kesme yok, döngüsel bekleme) bir örnek üzerinde göstermek.

Modelleme
Kaynak tahsis ve bekleme çizgelerini çizmek; döngünün ne zaman kilitlenme anlamına geldiğini söylemek.

Tespit ve kurtarma
Matris tabanlı tespit algoritmasını elle çalıştırmak; kesme, geri sarma ve sonlandırma ile kurtarmayı karşılaştırmak.

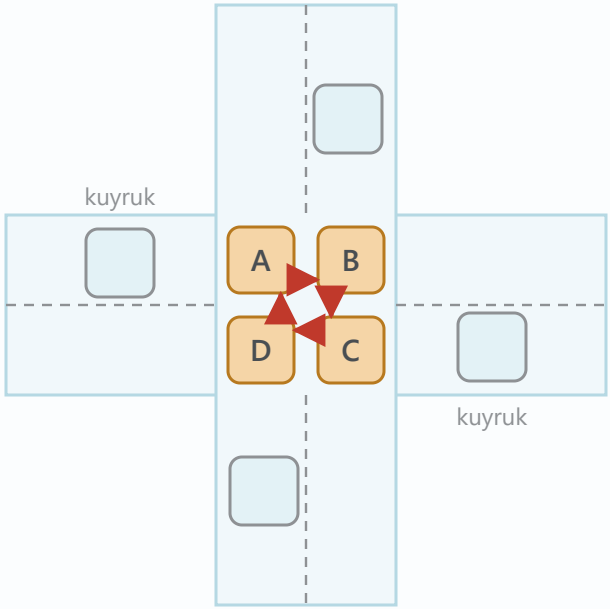
Kaçınma
Güvenli ve güvensiz durumları ayırt etmek; Bankacı algoritmasıyla bir isteğin verilip verilmeyeceğine karar vermek.

Önleme ve pratik
Kilit sıralaması, iki aşamalı kilitleme, canlı kilit ve açlığı tanımak; Java'da kilitlenmeyi `jstack` ile bulmak.

GİRİŞ

Kilitlenme Nedir?

- **Tanım (Tanenbaum):** bir süreç kümesindeki *her* süreç, yalnızca kümedeki *başka* bir sürecin tetikleyebileceği bir olayı bekliyorsa küme **kilitlenmiştir** (deadlock).
- Beklenen olay çoğunlukla bir **kaynağın serbest bırakılmasıdır**; hiçbiri çalışmadığından hiçbiri bırakamaz.
- Örnek: S1, K1'i tutup K2'yi bekler; S2, K2'yi tutup K1'i bekler.
- Gündelik örnek: kavşakta dört araç birbirinin önünü keser (*gridlock*); tek hatlı rayda karşılıklı iki tren.
- Kilitlenen süreçler **CPU harcamaz** (bloke olmuşlardır) ama tuttukları kaynaklar sonsuza dek kullanılamaz.



Konu yalnızca işletim sistemi değil: veritabanı işlemleri, çok iş parçacıklı uygulamalar ve dağıtık sistemler de kilitlenir.

GİRİŞ

Kaynaklar: Kesilebilir ve Kesilemez

Kesilebilir (preemptable)

Sahibinden *zarar vermeden* geri alınabilir. Örnek: CPU (bağlam kaydedilir), ana bellek (sayfa diske yazılır). Kilitlenme, kaynağı yeniden dağıtarak çözülebilir.

Kesilemez (nonpreemptable)

Geri almak işlemi bozar. Örnek: yazılmakta olan Blu-ray/yazıcı işi, bir `mutex`, veritabanı kaydı kilidi. Kilitlenmeler **bu tür** kaynaklarla ilgilidir.

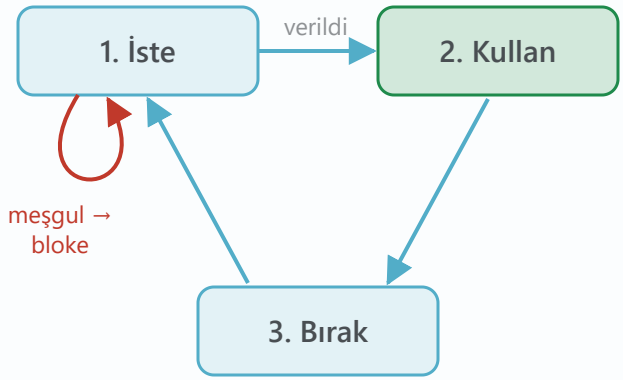
Kaynak	Örnek sayısı	Tür	Not
CPU çekirdeği	çok	kesilebilir	zamanlayıcı kesmesiyle el değiştirir
Bellek sayfası	çok	kesilebilir	takas/sayfalama ile geri alınır
Mutex, semafor, dosya kilidi	1	kesilemez	yalnızca sahibi bırakır
Veritabanı satır kilidi	1	kesilemez*	*işlem geri sarılırsa (abort) bırakılır

Her kaynak türü R_i'nin W_i tane özdeş örneği (instance) olabilir; bir istek o türün *herhangi* bir örneğiyle karşılanabiliyorsa örnekler özdeştir.

Kaynak Kullanım Döngüsü

- **İste (request):** kaynak boştaysa verilir; değilse süreç **bloke olur** (ya da hata kodu alıp sonra yeniden dener).
- **Kullan (use):** süreç kaynak üzerinde işlem yapar.
- **Bırak (release):** kaynak serbest kalır, bekleyenlerden biri uyandırılır.
- İşletim sistemi bir **sistem tablosunda** hangi kaynağın kime verildiğini ve kimin beklediğini tutar.

İste	Bırak
`open()`	`close()`
`malloc()` / `mmap()`	`free()` / `munmap()`
`pthread_mutex_lock()`	`pthread_mutex_unlock()`
`sem_wait()` (down)	`sem_post()` (up)



GİRİŞ

Aynı Sıra ve Ters Sıra

Her kaynak bir ikili semaforla korunur: erişmeden önce `down`, iş bitince `up`. Tek süreç ya da tek kaynak varken sorun yoktur.

Kilitlenmez: ikisi de önce 1, sonra 2

```
void process_A(void) {
  down(&resource_1);
  down(&resource_2);
  use_both_resources();
  up(&resource_2);
  up(&resource_1);
}
void process_B(void) {
  down(&resource_1);
  down(&resource_2);
  use_both_resources();
  up(&resource_2);
  up(&resource_1);
}
```

Kilitlenebilir: B önce 2, sonra 1

```
void process_A(void) {
  down(&resource_1);
  down(&resource_2);
  use_both_resources();
  up(&resource_2);
  up(&resource_1);
}
void process_B(void) {
  down(&resource_2);  /* ters sıra */
  down(&resource_1);
  use_both_resources();
  up(&resource_1);
  up(&resource_2);
}
```

İki süreç kaynakları **aynı sırada alıyorsa**, bırakma sırası farklı olsa bile kod kilitlenmez. Kilitlenme **edinme (down) sırası** ters olduğunda mümkündür (Tanenbaum Şekil 6-2b).

Ters Sırada Edinim: Adım Adım

```
▶ semaphore r1 = 1, r2 = 1;
2 void A(void) {
3   down(&r1);
4   down(&r2);
5   use_both();
6   up(&r2); up(&r1);
7 }
8 void B(void) {
9   down(&r2);
10  down(&r1);
11  use_both();
12  up(&r1); up(&r2);
13 }
```



İki ikili semafor 1 ile başlar: iki kaynak da boş.

Adım 1 / 6

DEĞİŞKENLER

`r1 = 1` `r2 = 1` `A = hazır` `B = hazır`

Kilitlenme her çalıştırmada oluşmaz: A iki `down`'u B başlamadan bitirirse sorun çıkmaz. Bu yüzden testte nadiren görülür, üretimde ortaya çıkar.

Ters Sırada Edinim: Adım Adım

```
1  semaphore r1 = 1, r2 = 1;
2  void A(void) {
▶    down(&r1);
4    down(&r2);
5    use_both();
6    up(&r2); up(&r1);
7  }
8  void B(void) {
9    down(&r2);
10   down(&r1);
11   use_both();
12   up(&r1); up(&r2);
13 }
```



A `down(&r1)` yapar ve r1'i alır (r1 → A kenarı).

Adım 2 / 6

DEĞİŞKENLER

`r1 = 0` `r2 = 1` `A = r1 tutuyor` `B = hazır`

Kilitlenme her çalıştırmada oluşmaz: A iki `down`'u B başlamadan bitirirse sorun çıkmaz. Bu yüzden testte nadiren görülür, üretimde ortaya çıkar.

Ters Sırada Edinim: Adım Adım

```
1  semaphore r1 = 1, r2 = 1;
2  void A(void) {
3    down(&r1);
4    down(&r2);
5    use_both();
6    up(&r2); up(&r1);
7  }
8  void B(void) {
▶    down(&r2);
10   down(&r1);
11   use_both();
12   up(&r1); up(&r2);
13 }
```



Çizelgeleyici B'ye geçer; B `down(&r2)` ile r2'yi alır.

Adım 3 / 6

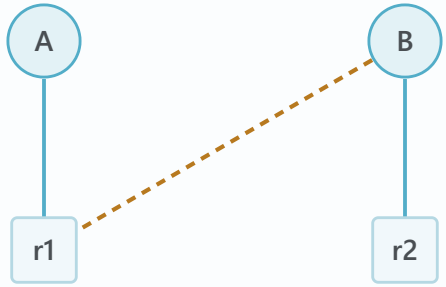
DEĞİŞKENLER

`r1 = 0` `r2 = 0` `A = r1 tutuyor` `B = r2 tutuyor`

Kilitlenme her çalıştırmada oluşmaz: A iki `down`'u B başlamadan bitirirse sorun çıkmaz. Bu yüzden testte nadiren görülür, üretimde ortaya çıkar.

Ters Sırada Edinim: Adım Adım

```
1  semaphore r1 = 1, r2 = 1;
2  void A(void) {
3    down(&r1);
4    down(&r2);
5    use_both();
6    up(&r2); up(&r1);
7  }
8  void B(void) {
9    down(&r2);
▶    down(&r1);
11   use_both();
12   up(&r1); up(&r2);
13 }
```



B r1'i ister; r1 A'da olduğu için B **bloke olur** (kesikli istek kenarı).

Adım 4 / 6

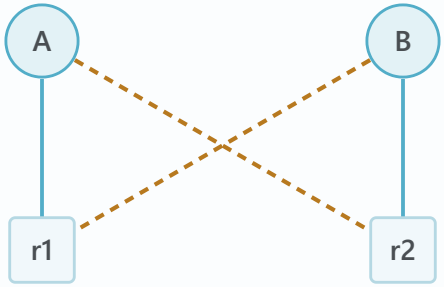
DEĞİŞKENLER

`r1 = 0` `r2 = 0` `A = r1 tutuyor` `B = bloke (r1)`

Kilitlenme her çalıştırmada oluşmaz: A iki `down`'u B başlamadan bitirirse sorun çıkmaz. Bu yüzden testte nadiren görülür, üretimde ortaya çıkar.

Ters Sırada Edinim: Adım Adım

```
1  semaphore r1 = 1, r2 = 1;
2  void A(void) {
3    down(&r1);
▶    down(&r2);
5    use_both();
6    up(&r2); up(&r1);
7  }
8  void B(void) {
9    down(&r2);
10   down(&r1);
11   use_both();
12   up(&r1); up(&r2);
13 }
```



A r2'yi ister; r2 B'de, A da bloke olur.

Adım 5 / 6

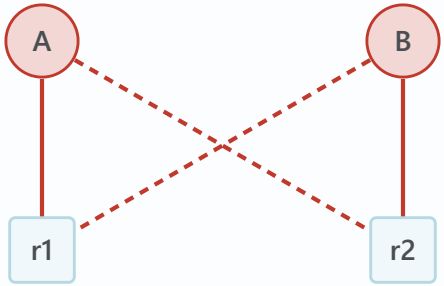
DEĞİŞKENLER

`r1 = 0` `r2 = 0` `A = bloke (r2)` `B = bloke (r1)`

Kilitlenme her çalıştırmada oluşmaz: A iki `down`'u B başlamadan bitirirse sorun çıkmaz. Bu yüzden testte nadiren görülür, üretimde ortaya çıkar.

Ters Sırada Edinim: Adım Adım

```
1  semaphore r1 = 1, r2 = 1;
2  void A(void) {
3    down(&r1);
▶    down(&r2);
5    use_both();
6    up(&r2); up(&r1);
7  }
8  void B(void) {
9    down(&r2);
▶    down(&r1);
11   use_both();
12   up(&r1); up(&r2);
13 }
```



döngü: A → r2 → B → r1 → A

Çizgede döngü oluştu. `up` satırlarına hiçbiri ulaşamaz: **kilitlenme**.

Adım 6 / 6

DEĞİŞKENLER

`r1 = 0` `r2 = 0` `A = bloke (r2)` `B = bloke (r1)`

Kilitlenme her çalıştırmada oluşmaz: A iki `down`'u B başlamadan bitirirse sorun çıkmaz. Bu yüzden testte nadiren görülür, üretimde ortaya çıkar.

KİLİTLENME KOŞULLARI

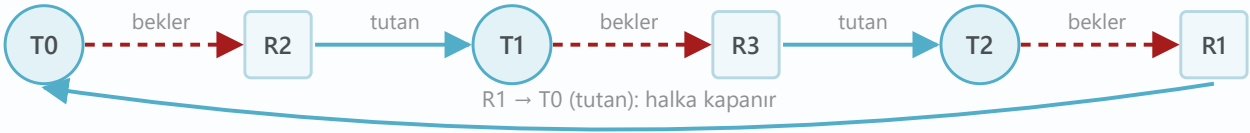
Kilitlenmenin Dört Koşulu (Coffman, 1971)

1. Karşılıklı dışlama
Kaynak ya tam olarak bir sürece atanmıştır ya da boştur; aynı anda paylaşılamaz.

2. Tut ve bekle
Bir süreç en az bir kaynağı *tutarken* başkasının tuttuğu yeni kaynakları ister ve bekler.

3. Kesme yok (no preemption)
Verilmiş bir kaynak zorla geri alınamaz; yalnızca tutan süreç kendi isteğiyle bırakır.

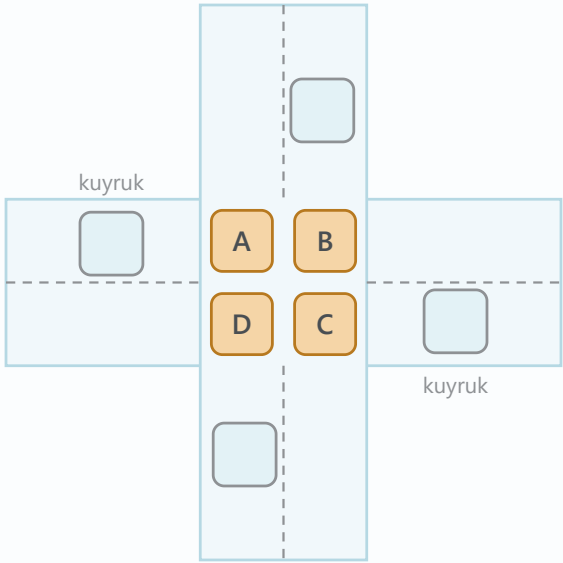
4. Döngüsel bekleme
T_0, T_1'in; T_1, T_2'nin; …; T_n, T_0'ın tuttuğu kaynağı bekler.



Dört koşul **birlikte gereklidir**: biri bile yoksa kilitlenme olamaz. Bu, "önleme" stratejisinin temelidir. İlk üçü sistemin *politikasıdır*; dördüncüsü o anki *durumdur*. Her türden tek örnek varsa dört koşul aynı zamanda **yeterlidir**.

Dört Koşul Bir Arada

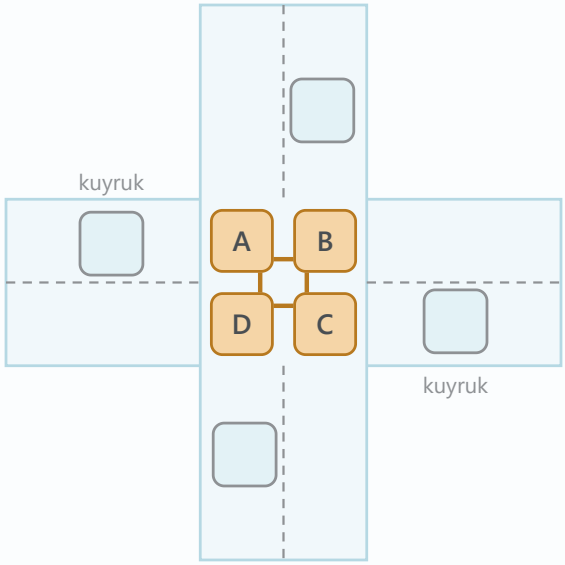
- Kavşağın dört karesi = dört **kesilemez** kaynak; araçlar = süreçler.
- Her adım bir koşulu işaretler.
- Herhangi bir koşulu kırmak çözümdür:
  - trafik polisi bir aracı geri çektirir (*kesme*),
  - "kavşak boşalmadan girme" kuralı (*tut ve bekle* yok),
  - trafik ışığı (sıralama, *döngü* yok).



Karşılıklı dışlama: bir kavşak karesinde aynı anda yalnızca bir araç bulunabilir.

Dört Koşul Bir Arada

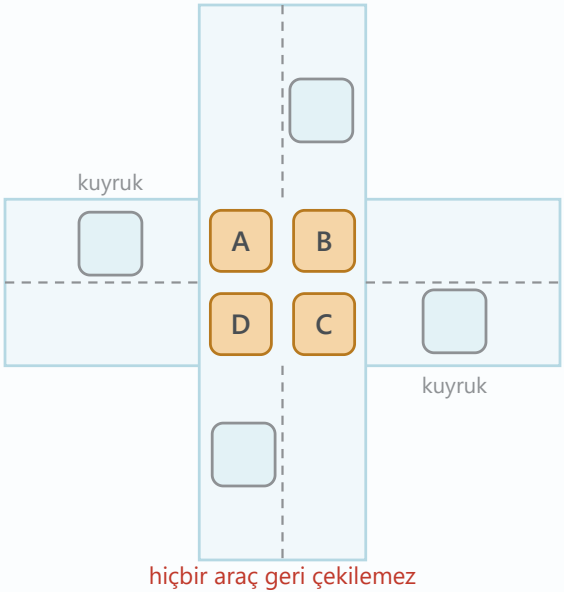
- Kavşağın dört karesi = dört **kesilemez** kaynak; araçlar = süreçler.
- Her adım bir koşulu işaretler.
- Herhangi bir koşulu kırmak çözümdür:
  - trafik polisi bir aracı geri çektirir (*kesme*),
  - "kavşak boşalmadan girme" kuralı (*tut ve bekle* yok),
  - trafik ışığı (sıralama, *döngü* yok).



Tut ve bekle: her araç bulunduğu kareyi tutarken önündeki kareyi bekliyor (turuncu oklar).

Dört Koşul Bir Arada

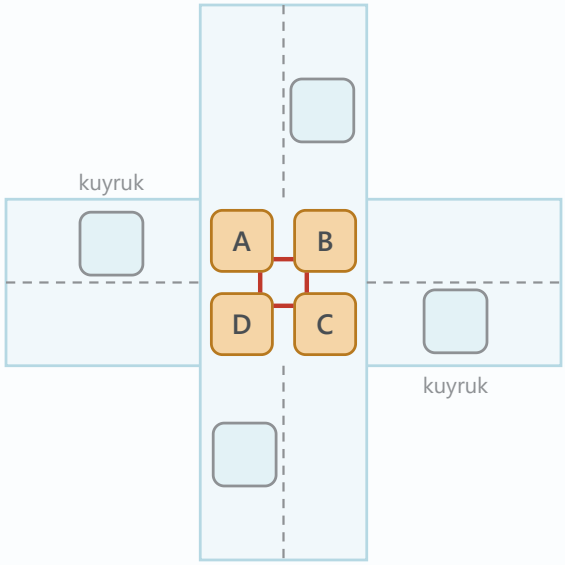
- Kavşağın dört karesi = dört **kesilemez** kaynak; araçlar = süreçler.
- Her adım bir koşulu işaretler.
- Herhangi bir koşulu kırmak çözümdür:
  - trafik polisi bir aracı geri çektirir (*kesme*),
  - "kavşak boşalmadan girme" kuralı (*tut ve bekle* yok),
  - trafik ışığı (sıralama, *döngü* yok).



Kesme yok: arkada kuyruk var; hiçbir araç kareyi boşaltmak için geri çekilemez.

Dört Koşul Bir Arada

- Kavşağın dört karesi = dört **kesilemez** kaynak; araçlar = süreçler.
- Her adım bir koşulu işaretler.
- Herhangi bir koşulu kırmak çözümdür:
  - trafik polisi bir aracı geri çektirir (*kesme*),
  - "kavşak boşalmadan girme" kuralı (*tut ve bekle* yok),
  - trafik ışığı (sıralama, *döngü* yok).



Döngüsel bekleme: A → B → C → D → A. Dört koşul birlikte: trafik kilitlendi.

MODELLEME

Kaynak Tahsis Çizgesi

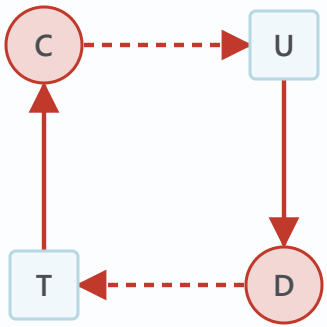
- Holt (1972): iki tür düğüm; **süreçler daire**, **kaynaklar kare**. Çok örnekli kaynakta karenin içinde her örnek için bir nokta.
- **Atama kenarı** R → P: R, P'ye verilmiş. **İstek kenarı** P → R: P, R'yi bekliyor (bu sunumda kesikli turuncu).
- İstek karşılandığında istek kenarı atama kenarına dönüşür; bırakıldığında silinir.



(a) A, R'yi tutuyor



(b) B, S'yi istiyor



(c) Kilitlenme: C → U → D → T → C

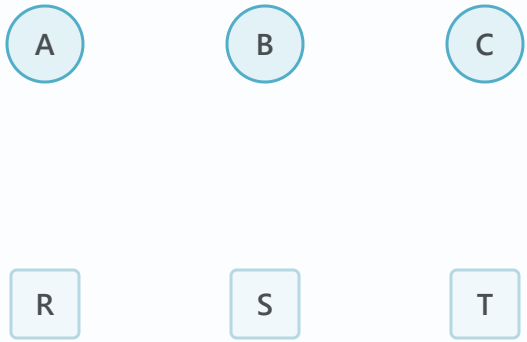
Tanenbaum MOS Şekil 6-3. Silberschatz aynı modeli "resource-allocation graph" olarak T (thread) düğümleriyle çizer.

MODELLEME

Kilitlenme Nasıl Oluşur?

```
A          B          C
R iste     S iste     T iste
S iste     T iste     R iste
R bırak    S bırak    T bırak
S bırak    T bırak    R bırak
```

İşletim sistemi istekleri **geldiği sırada** karşılarsa: R, S, T verilir, sonra her süreç bir sonrakinin kaynağını ister.



Başlangıç: üç kaynak da boş. Sıralı (round-robin) çizelgeleme varsayılıyor.

Adım 1 / 7

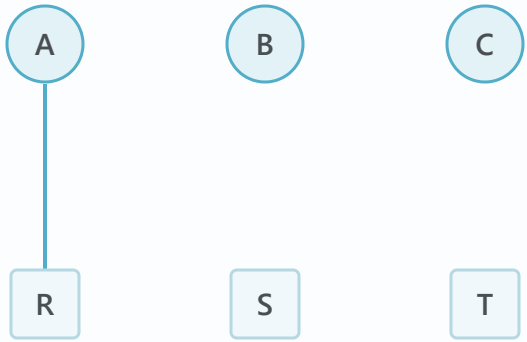
Tanenbaum MOS Şekil 6-4 (a)–(j). Gerçekte çizelgeleyici başka bir sıra da seçebilirdi; kilitlenme sıralamaya bağlıdır.

MODELLEME

Kilitlenme Nasıl Oluşur?

```
A          B          C
R iste     S iste     T iste
S iste     T iste     R iste
R bırak    S bırak    T bırak
S bırak    T bırak    R bırak
```

İşletim sistemi istekleri **geldiği sırada** karşılarsa: R, S, T verilir, sonra her süreç bir sonrakinin kaynağını ister.



1. A, R'yi ister ve alır.

Adım 2 / 7

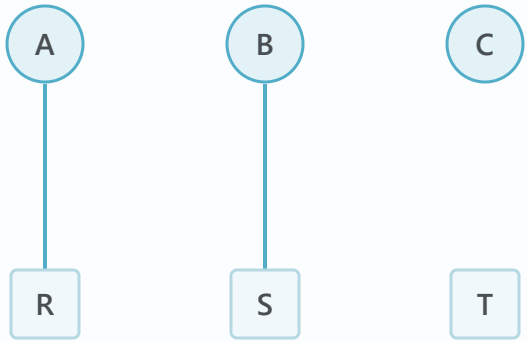
Tanenbaum MOS Şekil 6-4 (a)–(j). Gerçekte çizelgeleyici başka bir sıra da seçebilirdi; kilitlenme sıralamaya bağlıdır.

MODELLEME

Kilitlenme Nasıl Oluşur?

```
A          B          C
R iste     S iste     T iste
S iste     T iste     R iste
R bırak    S bırak    T bırak
S bırak    T bırak    R bırak
```

İşletim sistemi istekleri **geldiği sırada** karşılarsa: R, S, T verilir, sonra her süreç bir sonrakinin kaynağını ister.



2. B, S'yi ister ve alır.

Adım 3 / 7

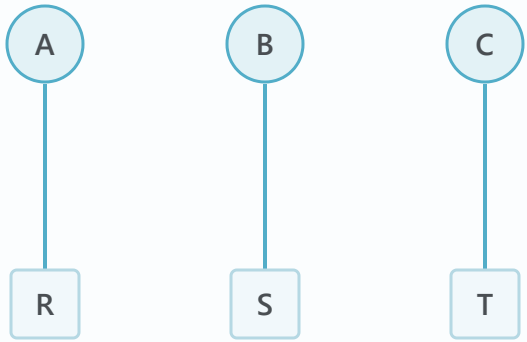
Tanenbaum MOS Şekil 6-4 (a)–(j). Gerçekte çizelgeleyici başka bir sıra da seçebilirdi; kilitlenme sıralamaya bağlıdır.

MODELLEME

Kilitlenme Nasıl Oluşur?

```
A          B          C
R iste     S iste     T iste
S iste     T iste     R iste
R bırak    S bırak    T bırak
S bırak    T bırak    R bırak
```

İşletim sistemi istekleri **geldiği sırada** karşılarsa: R, S, T verilir, sonra her süreç bir sonrakinin kaynağını ister.



3. C, T'yi ister ve alır.

Adım 4 / 7

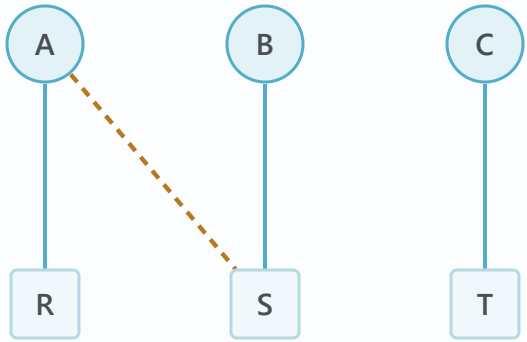
Tanenbaum MOS Şekil 6-4 (a)–(j). Gerçekte çizelgeleyici başka bir sıra da seçebilirdi; kilitlenme sıralamaya bağlıdır.

MODELLEME

Kilitlenme Nasıl Oluşur?

```
A          B          C
R iste     S iste     T iste
S iste     T iste     R iste
R bırak    S bırak    T bırak
S bırak    T bırak    R bırak
```

İşletim sistemi istekleri **geldiği sırada** karşılarsa: R, S, T verilir, sonra her süreç bir sonrakinin kaynağını ister.



4. A, S'yi ister: S, B'de → A bloke.

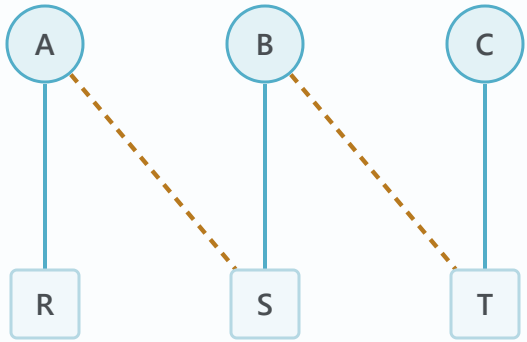
Adım 5 / 7

Tanenbaum MOS Şekil 6-4 (a)–(j). Gerçekte çizelgeleyici başka bir sıra da seçebilirdi; kilitlenme sıralamaya bağlıdır.

Kilitlenme Nasıl Oluşur?

```
A          B          C
R iste     S iste     T iste
S iste     T iste     R iste
R bırak    S bırak    T bırak
S bırak    T bırak    R bırak
```

İşletim sistemi istekleri **geldiği sırada** karşılarsa: R, S, T verilir, sonra her süreç bir sonrakinin kaynağını ister.



5. B, T'yi ister: T, C'de → B bloke.

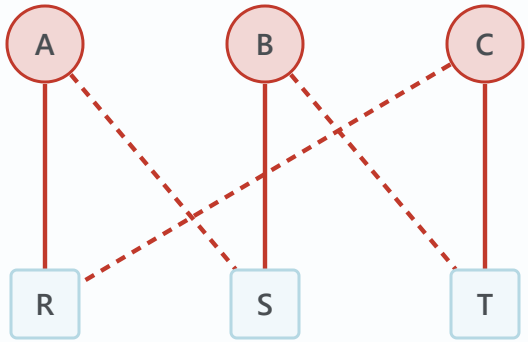
Tanenbaum MOS Şekil 6-4 (a)–(j). Gerçekte çizelgeleyici başka bir sıra da seçebilirdi; kilitlenme sıralamaya bağlıdır.

MODELLEME

Kilitlenme Nasıl Oluşur?

```
A          B          C
R iste     S iste     T iste
S iste     T iste     R iste
R bırak    S bırak    T bırak
S bırak    T bırak    R bırak
```

İşletim sistemi istekleri **geldiği sırada** karşılarsa: R, S, T verilir, sonra her süreç bir sonrakinin kaynağını ister.



döngü A→S→B→T→C→R→A

6. C, R'yi ister: R, A'da → C bloke. Döngü kapandı: **kilitlenme**.

Adım 7 / 7

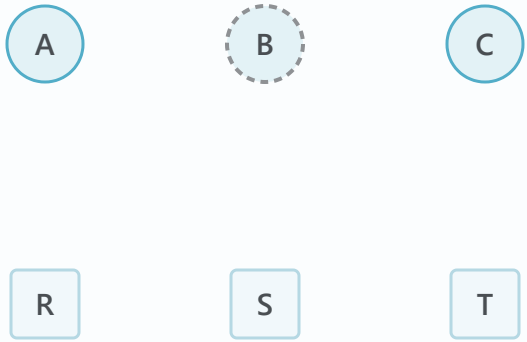
Tanenbaum MOS Şekil 6-4 (a)–(j). Gerçekte çizelgeleyici başka bir sıra da seçebilirdi; kilitlenme sıralamaya bağlıdır.

MODELLEME

Kilitlenme Nasıl Önlenebilir?

```
A          B          C
R iste     S iste     T iste
S iste     T iste     R iste
R bırak    S bırak    T bırak
S bırak    T bırak    R bırak
```

İS, bir isteği vermenin kilitlenmeye yol açacağını **önceden bilseydi**, B'yi bekleterek tehlikeli sırayı engelleyebilirdi.



İS, B'nin isteklerini bir süre **askıya alır** (B'yi çizelgelemez). Sadece A ve C çalışır.

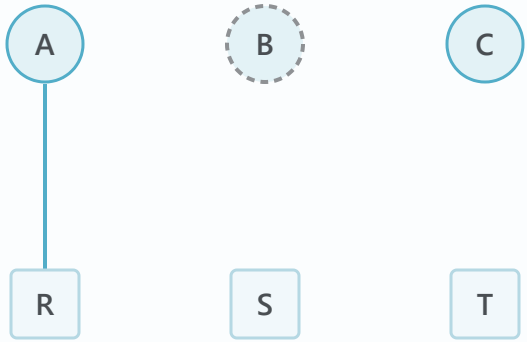
Adım 1 / 7

Tanenbaum MOS Şekil 6-4 (k)–(q). "Hangi isteği ertelemeli?" sorusu, ilerideki **kaçınma** (Bankacı algoritması) konusunun özüdür.

Kilitlenme Nasıl Önlenebilir?

```
A          B          C
R iste     S iste     T iste
S iste     T iste     R iste
R bırak    S bırak    T bırak
S bırak    T bırak    R bırak
```

İS, bir isteği vermenin kilitlenmeye yol açacağını **önceden bilseydi**, B'yi bekleterek tehlikeli sırayı engelleyebilirdi.



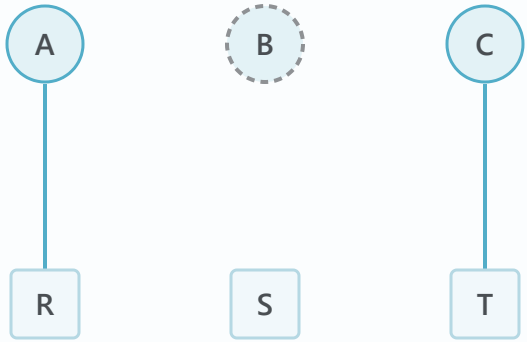
1. A, R'yi ister ve alır.

Tanenbaum MOS Şekil 6-4 (k)–(q). "Hangi isteği ertelemeli?" sorusu, ilerideki **kaçınma** (Bankacı algoritması) konusunun özüdür.

Kilitlenme Nasıl Önlenebilir?

```
A          B          C
R iste     S iste     T iste
S iste     T iste     R iste
R bırak    S bırak    T bırak
S bırak    T bırak    R bırak
```

İS, bir isteği vermenin kilitlenmeye yol açacağını **önceden bilseydi**, B'yi bekleterek tehlikeli sırayı engelleyebilirdi.



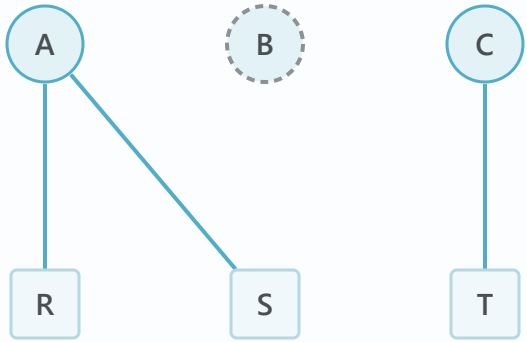
2. C, T'yi ister ve alır.

Tanenbaum MOS Şekil 6-4 (k)–(q). "Hangi isteği ertelemeli?" sorusu, ilerideki **kaçınma** (Bankacı algoritması) konusunun özüdür.

Kilitlenme Nasıl Önlenebilir?

```
A          B          C
R iste     S iste     T iste
S iste     T iste     R iste
R bırak    S bırak    T bırak
S bırak    T bırak    R bırak
```

İS, bir isteği vermenin kilitlenmeye yol açacağını **önceden bilseydi**, B'yi bekleterek tehlikeli sırayı engelleyebilirdi.



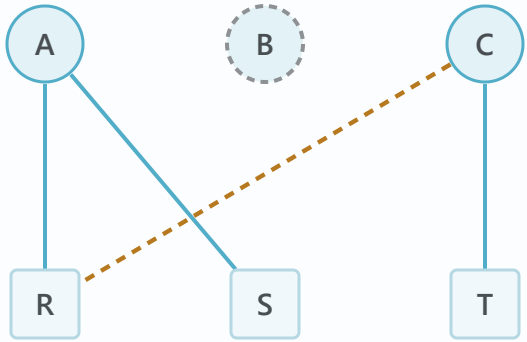
3. A, S'yi ister ve alır (S boş).

Tanenbaum MOS Şekil 6-4 (k)–(q). "Hangi isteği ertelemeli?" sorusu, ilerideki **kaçınma** (Bankacı algoritması) konusunun özüdür.

Kilitlenme Nasıl Önlenebilir?

```
A          B          C
R iste     S iste     T iste
S iste     T iste     R iste
R bırak    S bırak    T bırak
S bırak    T bırak    R bırak
```

İS, bir isteği vermenin kilitlenmeye yol açacağını **önceden bilseydi**, B'yi bekleterek tehlikeli sırayı engelleyebilirdi.



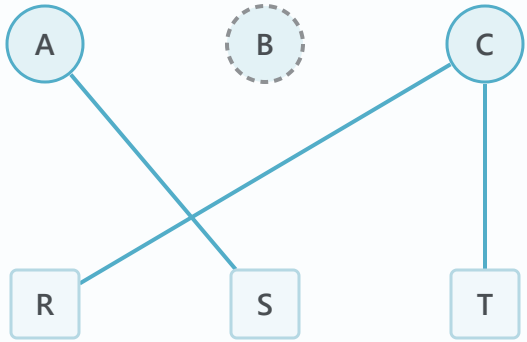
4. C, R'yi ister: R, A'da → C bloke. Döngü yok, A ilerleyebilir.

Tanenbaum MOS Şekil 6-4 (k)–(q). "Hangi isteği ertelemeli?" sorusu, ilerideki **kaçınma** (Bankacı algoritması) konusunun özüdür.

Kilitlenme Nasıl Önlenebilir?

```
A          B          C
R iste     S iste     T iste
S iste     T iste     R iste
R bırak    S bırak    T bırak
S bırak    T bırak    R bırak
```

İS, bir isteği vermenin kilitlenmeye yol açacağını **önceden bilseydi**, B'yi bekleterek tehlikeli sırayı engelleyebilirdi.



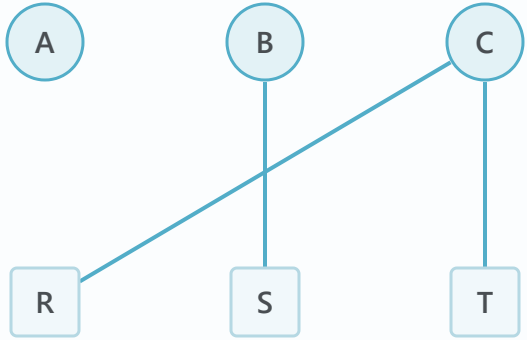
5. A, R'yi bırakır → R, C'ye verilir; C ilerler.

Tanenbaum MOS Şekil 6-4 (k)–(q). "Hangi isteği ertelemeli?" sorusu, ilerideki **kaçınma** (Bankacı algoritması) konusunun özüdür.

Kilitlenme Nasıl Önlenebilir?

```
A          B          C
R iste     S iste     T iste
S iste     T iste     R iste
R bırak    S bırak    T bırak
S bırak    T bırak    R bırak
```

İS, bir isteği vermenin kilitlenmeye yol açacağını **önceden bilseydi**, B'yi bekleterek tehlikeli sırayı engelleyebilirdi.



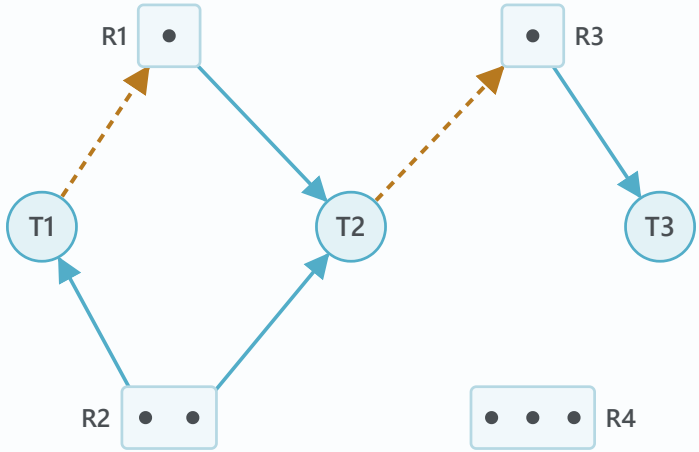
6. A, S'yi bırakıp biter. Artık B çalıştırılabilir: S'yi alır, T'yi C bitince alır. Kilitlenme yok.

Tanenbaum MOS Şekil 6-4 (k)–(q). "Hangi isteği ertelemeli?" sorusu, ilerideki **kaçınma** (Bankacı algoritması) konusunun özüdür.

MODELLEME

Çok Örnekli Kaynaklar

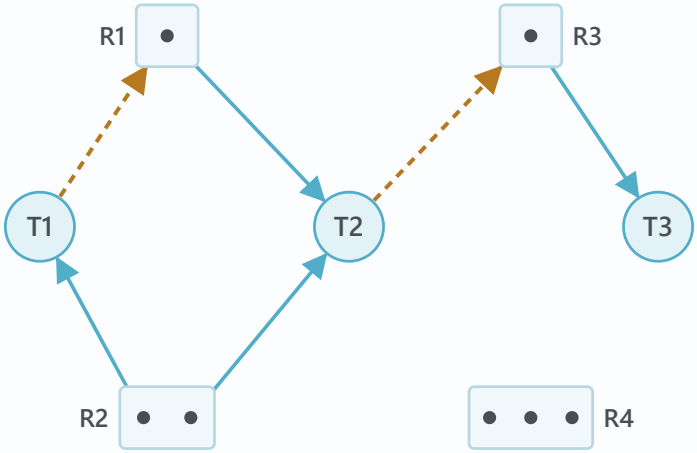
- Kaynak kümeleri: R1 ×1, **R2 ×2**, R3 ×1, **R4 ×3**.
- **T1:** bir R2 örneğini tutar, R1'i bekler.
- **T2:** R1'i ve bir R2 örneğini tutar, R3'ü bekler.
- **T3:** R3'ü tutar.
- Atama kenarı karedeki **noktadan** (örnekten) çıkar; istek kenarı karenin kendisine gider.
- Çizgede döngü yok → **kilitlenme yok**: T3 bitince R3'ü bırakır, T2 sonra T1 ilerler.



Silberschatz 10e Şekil 8.4. R4'ün üç örneği de boşta.

Döngü ve Kilitlenme

- Tek bir istek kenarı (T3 → R2) iki döngü oluşturur.
- T3, T1 ya da T2'nin bir R2 bırakmasını bekler; T2, T3'ün R3'ünü; T1, T2'nin R1'ini.
- Döngüdeki her süreç döngüdeki bir başkasını bekliyor ve dışarıdan bir "kurtarıcı" yok.



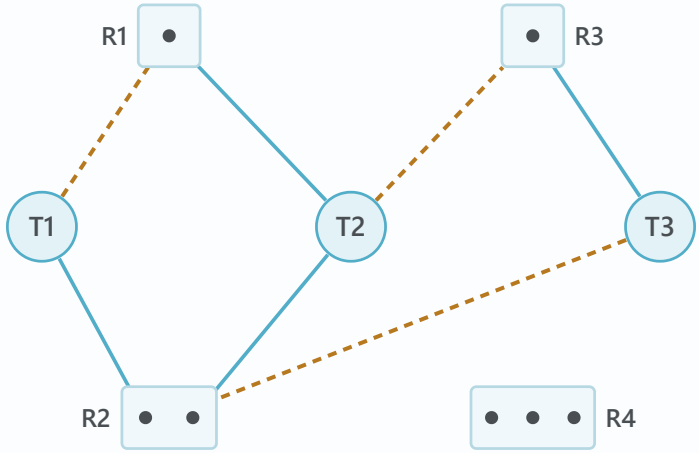
Önceki durum: döngü yok.

Adım 1 / 4

Silberschatz 10e Şekil 8.5.

Döngü ve Kilitlenme

- Tek bir istek kenarı (T3 → R2) iki döngü oluşturur.
- T3, T1 ya da T2'nin bir R2 bırakmasını bekler; T2, T3'ün R3'ünü; T1, T2'nin R1'ini.
- Döngüdeki her süreç döngüdeki bir başkasını bekliyor ve dışarıdan bir "kurtarıcı" yok.

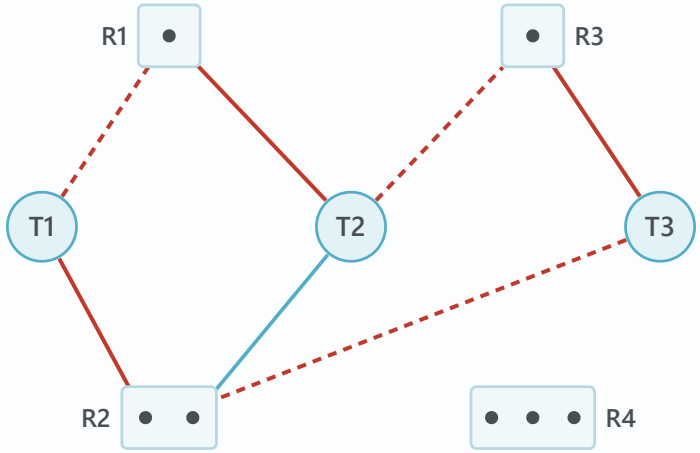


T3, bir R2 örneği ister. İki örnek de dolu (T1 ve T2'de) → T3 bloke.

Silberschatz 10e Şekil 8.5.

Döngü ve Kilitlenme

- Tek bir istek kenarı (T3 → R2) iki döngü oluşturur.
- T3, T1 ya da T2'nin bir R2 bırakmasını bekler; T2, T3'ün R3'ünü; T1, T2'nin R1'ini.
- Döngüdeki her süreç döngüdeki bir başkasını bekliyor ve dışarıdan bir "kurtarıcı" yok.

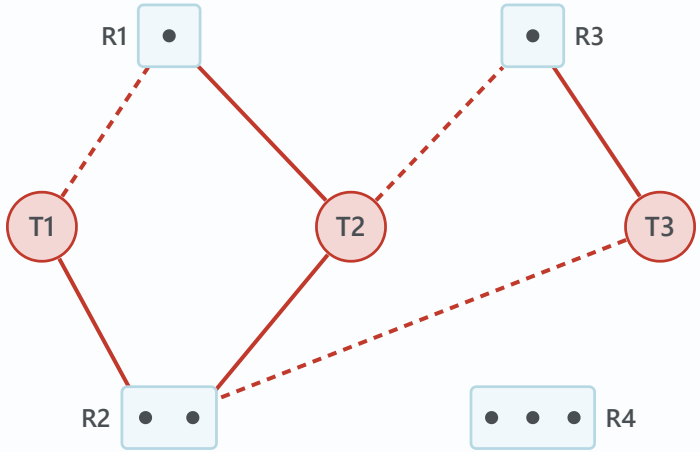


Döngü 1: T1 → R1 → T2 → R3 → T3 → R2 → T1.

Silberschatz 10e Şekil 8.5.

Döngü ve Kilitlenme

- Tek bir istek kenarı (T3 → R2) iki döngü oluşturur.
- T3, T1 ya da T2'nin bir R2 bırakmasını bekler; T2, T3'ün R3'ünü; T1, T2'nin R1'ini.
- Döngüdeki her süreç döngüdeki bir başkasını bekliyor ve dışarıdan bir "kurtarıcı" yok.

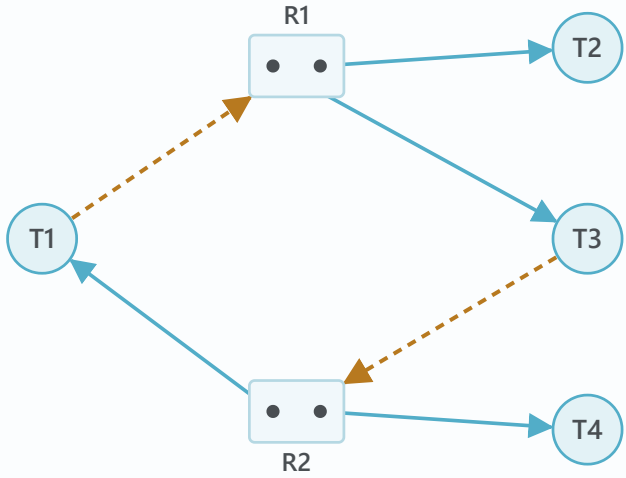


Döngü 2: T2 → R3 → T3 → R2 → T2. R2'nin *her iki* örneğinin sahibi de döngüde → **kilitlenme**.

Silberschatz 10e Şekil 8.5.

Döngü Var, Kilitlenme Yok

- Kaynak türlerinin birden çok örneği varsa döngü **gerekli ama yeterli değildir**.
- Döngü dışındaki bir süreç (T2 ya da T4) bitip örneğini bırakınca döngü çözülür.
- Bu yüzden çok örnekli durumda çizgede döngü aramak yetmez; **matris tabanlı** bir algoritma gerekir.



T1 bir R1 ister; R1'in iki örneği T2 ve T3'te. T3 bir R2 ister; R2'nin örnekleri T1 ve T4'te.

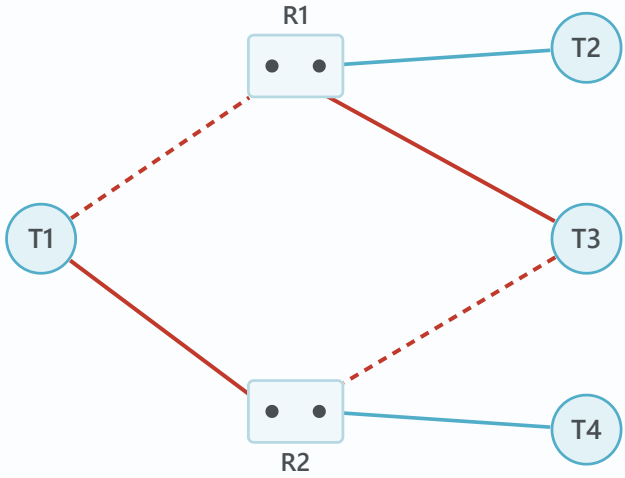
Adım 1 / 3

Silberschatz 10e Şekil 8.6.

MODELLEME

Döngü Var, Kilitlenme Yok

- Kaynak türlerinin birden çok örneği varsa döngü **gerekli ama yeterli değildir**.
- Döngü dışındaki bir süreç (T2 ya da T4) bitip örneğini bırakınca döngü çözülür.
- Bu yüzden çok örnekli durumda çizgede döngü aramak yetmez; **matris tabanlı** bir algoritma gerekir.

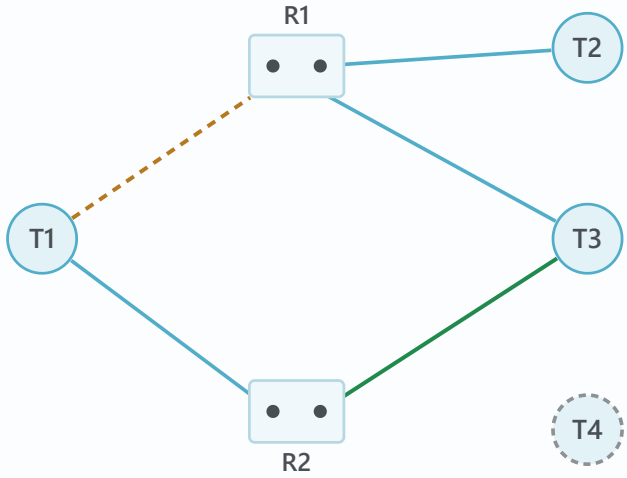


Döngü var: T1 → R1 → T3 → R2 → T1. Ama T2 ve T4 döngünün *dışında* ve hiçbir şey beklemiyor.

Silberschatz 10e Şekil 8.6.

Döngü Var, Kilitlenme Yok

- Kaynak türlerinin birden çok örneği varsa döngü **gerekli ama yeterli değildir**.
- Döngü dışındaki bir süreç (T2 ya da T4) bitip örneğini bırakınca döngü çözülür.
- Bu yüzden çok örnekli durumda çizgede döngü aramak yetmez; **matris tabanlı** bir algoritma gerekir.



T4 işini bitirip R2 örneğini bırakır → örnek T3'e verilir. Döngü kırıldı: **kilitlenme yoktu**.

Adım 3 / 3

Silberschatz 10e Şekil 8.6.

MODELLEME

Döngü Kuralı

Çizge	Her kaynak türünde tek örnek	Bazı türlerde birden çok örnek
Döngü yok	Kilitlenme yok	Kilitlenme yok
Döngü var	**Kilitlenme var** (döngü ⇔ kilitlenme)	Kilitlenme **olabilir**; döngü gerekli ama yeterli değil

Tek örnek

Döngü arama yeterli: DFS ile O(V + E). Döngüdeki süreçler kilitlenmiştir.

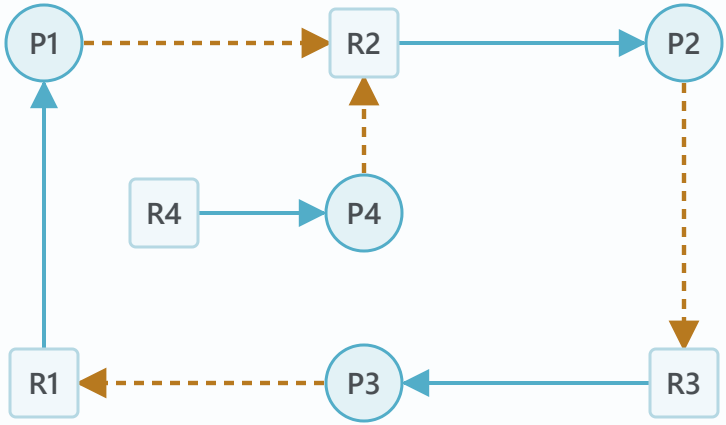
Çok örnek

Döngü yalnızca bir uyarıdır. Kesin karar için tahsis/istek matrisleriyle “herkes bitebilir mi?” diye sorulur.

Döngüsel bekleme koşulu, döngü içeren çizgenin karşılığıdır; tek örnekli kaynaklarda dört koşulun birlikte varlığı kilitlenmeyi garanti eder.

MODELLEME

Bekleme (Wait-For) Çizgesi



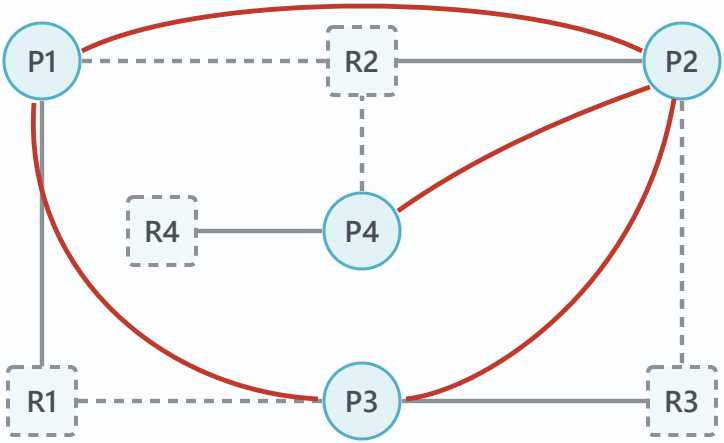
Tek örnekli kaynaklarla bir tahsis çizgesi: P1 R1'i tutar R2'yi ister, P2 R2'yi tutar R3'ü ister, P3 R3'ü tutar R1'i ister, P4 R4'ü tutar R2'yi ister.

Adım 1 / 4

Yalnızca **her türden tek örnek** olduğunda kullanılabilir. Veritabanları (ör. PostgreSQL, MySQL InnoDB) kilit bekleme çizgesinde döngü arayarak kilitlenme tespit eder.

MODELLEME

Bekleme (Wait-For) Çizgesi

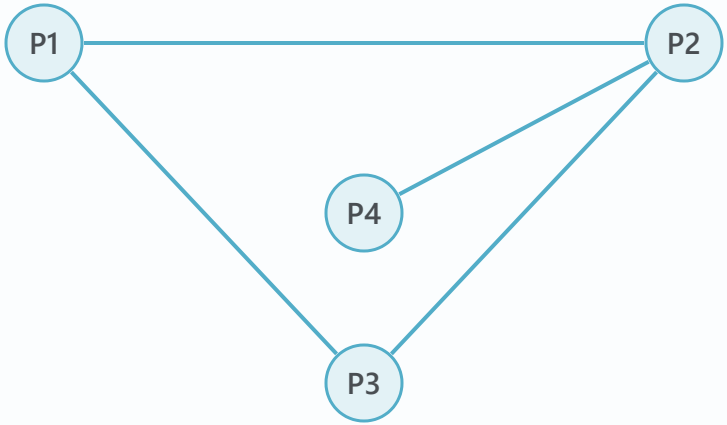


Her $P_i \rightarrow R \rightarrow P_j$ yolu tek bir $P_i \rightarrow P_j$ kenarına daraltılır (kırmızı): "P_i, P_j'yi bekliyor".

Adım 2 / 4

Yalnızca **her türden tek örnek** olduğunda kullanılabilir. Veritabanları (ör. PostgreSQL, MySQL InnoDB) kilit bekleme çizgesinde döngü arayarak kilitlenme tespit eder.

Bekleme (Wait-For) Çizgesi

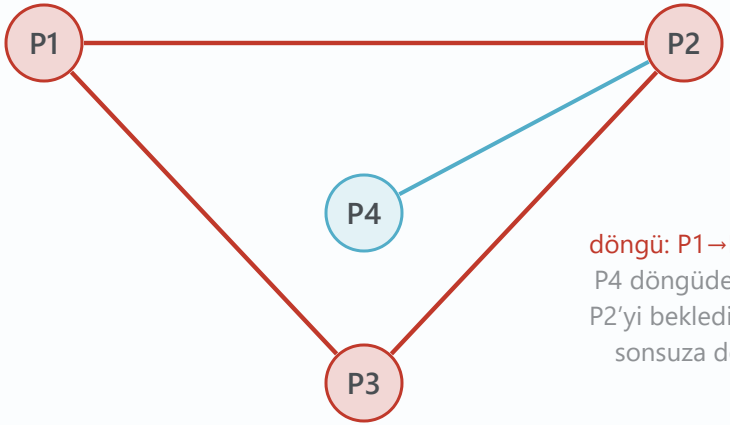


Kaynak düğümleri silinir: **bekleme (wait-for) çizgesi**.

Yalnızca **her türden tek örnek** olduğunda kullanılabilir. Veritabanları (ör. PostgreSQL, MySQL InnoDB) kilit bekleme çizgesinde döngü arayarak kilitlenme tespit eder.

MODELLEME

Bekleme (Wait-For) Çizgesi



döngü: P1→P2→P3→P1
P4 döngüde değil ama P2'yi beklediği için o da sonsuza dek bekler

Bekleme çizgesinde döngü ⇔ kilitlenme. Algoritma çizgeyi periyodik tarar; maliyet $O(n^2)$ (n: süreç sayısı).

Yalnızca **her türden tek örnek** olduğunda kullanılabilir. Veritabanları (ör. PostgreSQL, MySQL InnoDB) kilit bekleme çizgesinde döngü arayarak kilitlenme tespit eder.

STRATEJİLER

Kilitlenmelerle Başa Çıkma Stratejileri

1. Görmezden gel

"Devekuşu algoritması". Kilitlenme nadir ve önleme maliyeti yüksekse sorun yok sayılır, gerekirse sistem yeniden başlatılır.

2. Tespit et ve kurtar

Tepkisel. Kilitlenmeye izin verilir, periyodik olarak tespit edilir, sonra kesme, geri sarma ya da sonlandırma ile çözülür.

3. Dinamik kaçınma

Önleyici, çalışma anında. Sistemi güvensiz duruma sokacak istek ertelenir (Bankacı). Azami ihtiyaçların önceden bilinmesini ister.

4. Önleme

Önleyici, tasarım anında. Dört koşuldan en az biri yapısal olarak imkânsız kılınır (ör. kilit sıralaması).

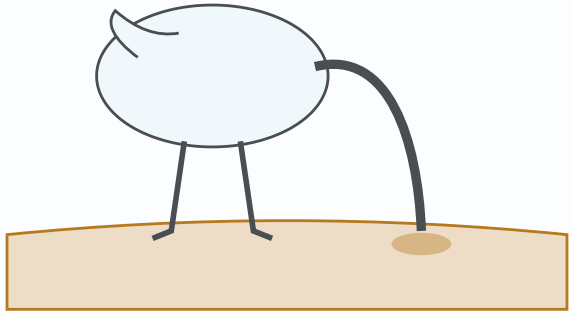
"Kurtarma" ayrı bir teknik değil, tespitin ikinci yarısıdır.

STRATEJİLER

Devekuşu Algoritması

- Strateji: **hiçbir şey yapma**. Kilitlenme olursa kullanıcı/yönetici süreci öldürür ya da sistemi yeniden başlatır.
- Mühendislik dengesi: kilitlenme yılda bir kez oluyor, çökmeler ve donanım hataları ise çok daha sık görülüyorsa, pahalı ve kısıtlayıcı önlemler değmez.
- Genel amaçlı İS'ler (Linux, Windows, macOS) kullanıcı süreçlerinin kilitleri, dosya kilitleri, boru (pipe) beklemeleri için bunu uygular; sorumluluk uygulama geliştiricisindedir.
- Ama İS kendi içinde kilitlenmeye karşı önlem alır: çekirdek kilit sıralaması, `lockdep` denetimi.
- Gerçek zamanlı ve güvenlik-kritik sistemlerde kabul edilemez.

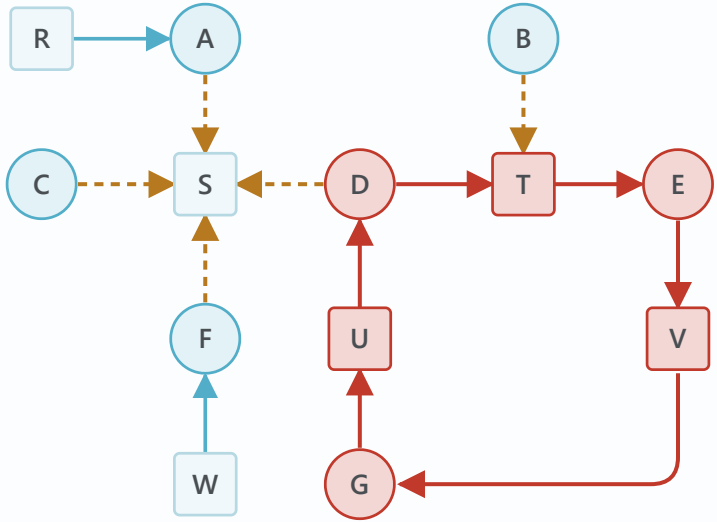
"Kilitlenme mi? Ne kilitlenmesi?"



Devekuşu yaklaşımı tespit ile kaçınma arasında bir denge değildir; ne tespit ne kaçınma yapar, yalnızca maliyet–olasılık karşılaştırmasıdır.

Tespit: Her Türden Tek Kaynak

- A: R'yi tutar, S'yi ister. B: T'yi ister.
- C: S'yi ister. D: U'yu tutar, S ve T'yi ister.
- E: T'yi tutar, V'yi ister. F: W'yi tutar, S'yi ister.
- G: V'yi tutar, U'yu ister.
- **Soru:** sistem kilitlenmiş mi, kilitlendiyse hangi süreçler?
- Çizgede döngü: **D** → **T** → **E** → **V** → **G** → **U** → **D**. D, E, G kilitli.
- A, C, F kilitli değil: S boşta; biri alır, bitirir, bırakır. B ise döngüdeki T'yi beklediği için ilerleyemez (döngüde olmasa da takılı).



Tanenbaum MOS Şekil 6-5. Göz ile kolay; yüzlerce düğümlü bir çizgede sistematik bir algoritma gerekir.

TESPİT VE KURTARMA

Döngü Arama Algoritması (DFS)

1. Çizgedeki her N düğümü için 2–6. adımları N'den başlayarak uygula.
2. L'yi boş liste yap; tüm kenarları işaretsiz say.
3. Geçerli düğümü L'nin sonuna ekle. Düğüm L'de **iki kez** geçiyorsa çizgede döngü vardır: L'deki döngüyü raporla, dur.
4. Geçerli düğümden çıkan işaretsiz kenar var mı? Varsa 5'e, yoksa 6'ya git.
5. Rastgele bir işaretsiz kenar seç, işaretle, gösterdiği düğüme geç; 3'e dön.
6. Başlangıç düğümündeysek bu N'den erişilebilen döngü yoktur. Değilsek çıkmaz sokaktayız: düğümü L'den çıkar, önceki düğüme dön, 4'ten devam et (o düğüm zaten L'de).

Özünde

Her düğümden başlayan bir *derinlik öncelikli arama*; yığın görevini L listesi görür.

Maliyet

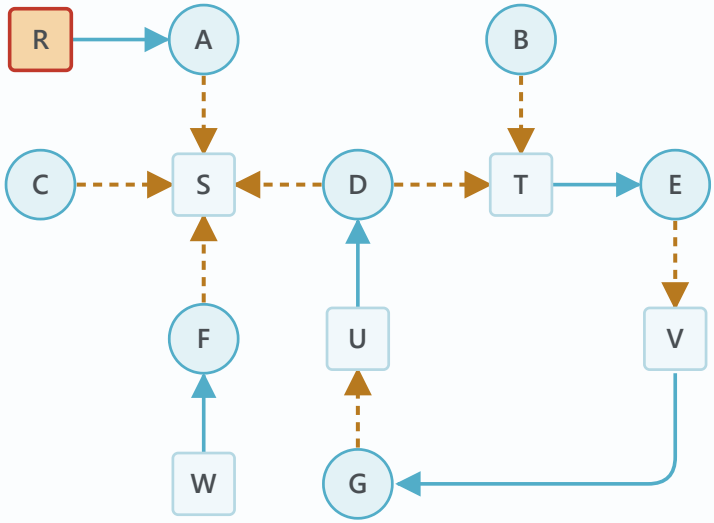
Basit sürüm her düğümden yeniden başladığı için $O(V \cdot (V+E))$; renkli DFS (beyaz/gri/siyah) tek geçişte $O(V+E)$.

Çıktı

L'nin tekrar eden düğümden sonraki kısmı döngüdür; içindeki süreçler kilitlenmiştir.

Döngü Arama: Adım Adım

- Başlangıç düğümleri şekildeki sırayla (soldan sağa, yukarıdan aşağı) seçilir: R, A, B, C, …
- Turuncu düğümler o anki L listesinde; kırmızı çerçeve geçerli düğüm.
- Kenar seçimi “rastgele” olabilir; burada önce S, sonra T seçiliyor.



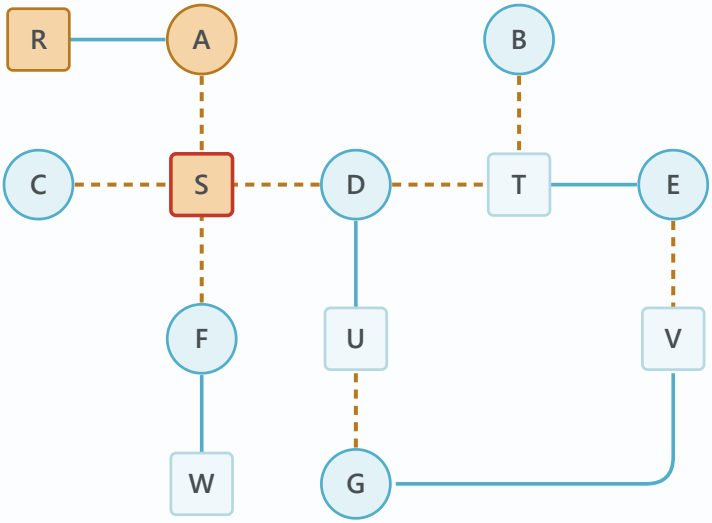
`L = [R]`

R’den başlanır. L = [R]. Tek çıkış R → A.

Adım 1 / 7

Döngü Arama: Adım Adım

- Başlangıç düğümleri şekildeki sırayla (soldan sağa, yukarıdan aşağı) seçilir: R, A, B, C, …
- Turuncu düğümler o anki L listesinde; kırmızı çerçeve geçerli düğüm.
- Kenar seçimi “rastgele” olabilir; burada önce S, sonra T seçiliyor.

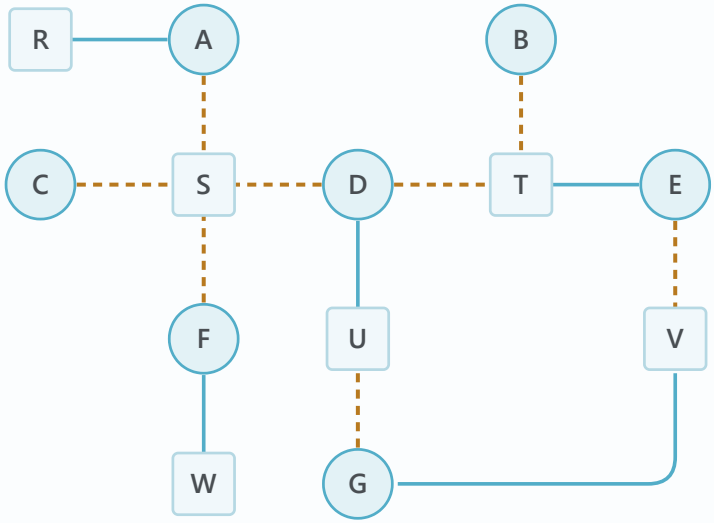


`L = [R, A, S]`

R → A → S. S’den çıkan kenar yok: çıkmaz sokak.

Döngü Arama: Adım Adım

- Başlangıç düğümleri şekildeki sırayla (soldan sağa, yukarıdan aşağı) seçilir: R, A, B, C, …
- Turuncu düğümler o anki L listesinde; kırmızı çerçeve geçerli düğüm.
- Kenar seçimi “rastgele” olabilir; burada önce S, sonra T seçiliyor.

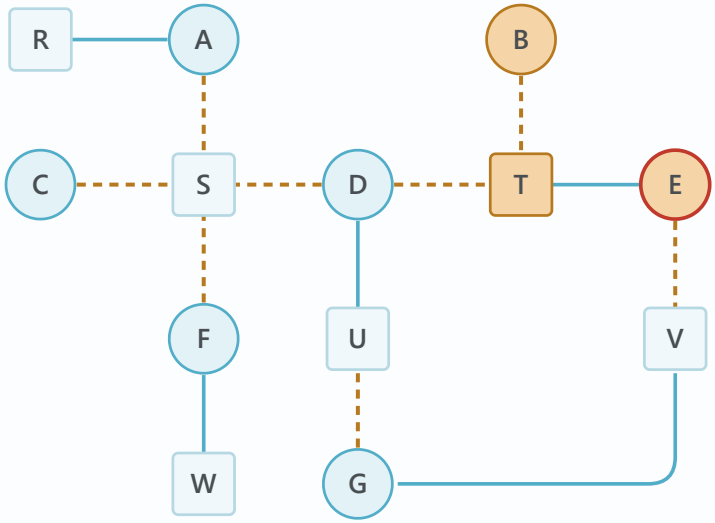


`L = []`

S, sonra A ve R L’den çıkarılır; R’ye dönüldü: R’den döngü yok. Sonra A’dan başlanır (yine yok)…

Döngü Arama: Adım Adım

- Başlangıç düğümleri şekildeki sırayla (soldan sağa, yukarıdan aşağı) seçilir: R, A, B, C, …
- Turuncu düğümler o anki L listesinde; kırmızı çerçeve geçerli düğüm.
- Kenar seçimi “rastgele” olabilir; burada önce S, sonra T seçiliyor.

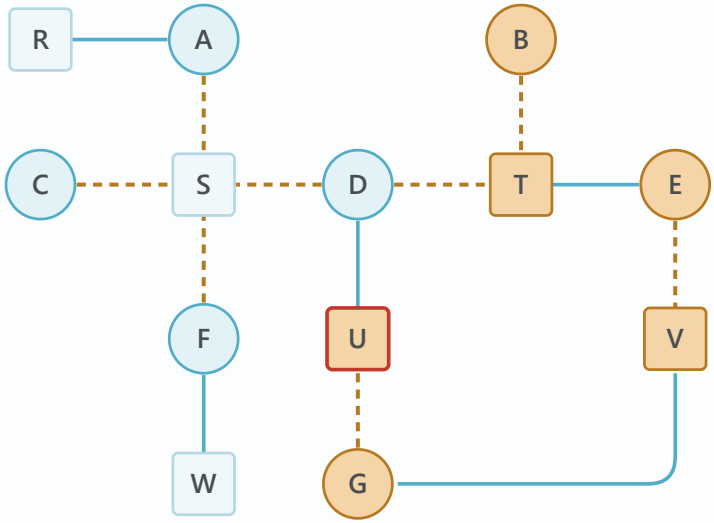


`L = [B, T, E]`

B’den başlanır: B → T → E.

Döngü Arama: Adım Adım

- Başlangıç düğümleri şekildeki sırayla (soldan sağa, yukarıdan aşağı) seçilir: R, A, B, C, …
- Turuncu düğümler o anki L listesinde; kırmızı çerçeve geçerli düğüm.
- Kenar seçimi “rastgele” olabilir; burada önce S, sonra T seçiliyor.

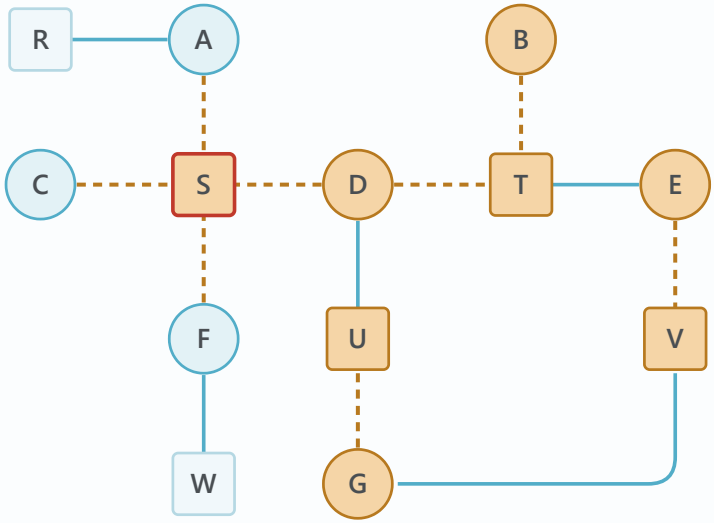


`L = [B, T, E, V, G, U]`

E → V → G → U. Hâlâ tekrar yok.

Döngü Arama: Adım Adım

- Başlangıç düğümleri şekildeki sırayla (soldan sağa, yukarıdan aşağı) seçilir: R, A, B, C, …
- Turuncu düğümler o anki L listesinde; kırmızı çerçeve geçerli düğüm.
- Kenar seçimi “rastgele” olabilir; burada önce S, sonra T seçiliyor.

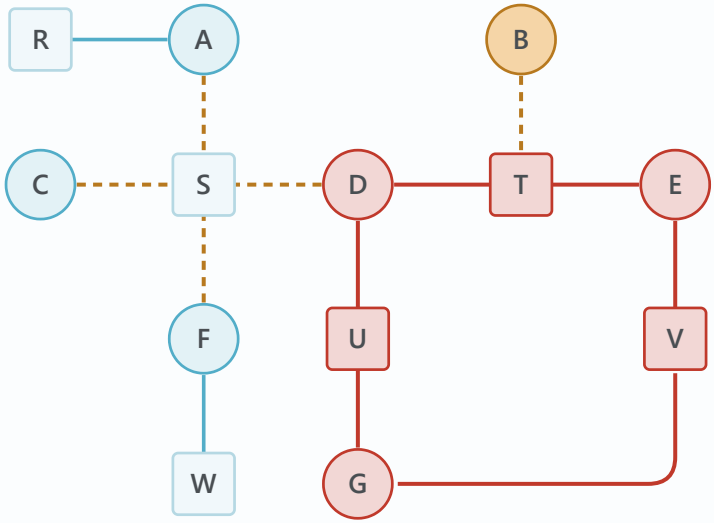


`L = [B, T, E, V, G, U, D, S]`

U → D. D’den önce S denenir: çıkmaz sokak, S L’den çıkar.

Döngü Arama: Adım Adım

- Başlangıç düğümleri şekildeki sırayla (soldan sağa, yukarıdan aşağı) seçilir: R, A, B, C, …
- Turuncu düğümler o anki L listesinde; kırmızı çerçeve geçerli düğüm.
- Kenar seçimi “rastgele” olabilir; burada önce S, sonra T seçiliyor.

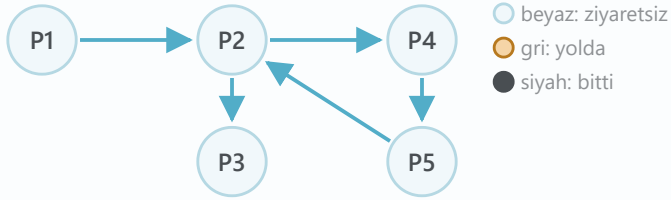


`L = [B, T, E, V, G, U, D, T]`

D → T: T, L’de **ikinci kez** → döngü: T, E, V, G, U, D. Algoritma durur.

Döngü Arama: Renkli DFS Kodu

```
1  enum { BEYAZ, GRI, SIYAH };
2  int renk[N];          /* hepsi BEYAZ */
3  bool dongu(int u) {
4    renk[u] = GRI;      /* yolda */
5    for (int v = 0; v < N; v++) {
6      if (!bekler[u][v]) continue;
7      if (renk[v] == GRI)
8        return true;    /* geri kenar */
9      if (renk[v] == BEYAZ && dongu(v))
10       return true;
11   }
12   renk[u] = SIYAH;    /* bitti */
13   return false;
14 }
```



P2 → P3, P4 sırasıyla denenir

Bekleme çizgesi: P1 → P2, P2 → P3 ve P4, P4 → P5, P5 → P2. Hepsi **beyaz**; arama P1'den başlar.

Adım 1 / 9

DEĞİŞKENLER

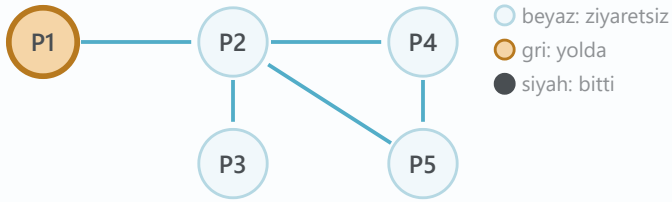
P1 = B | P2 = B | P3 = B | P4 = B | P5 = B

ÇAĞRI YIĞINI

`main`

Döngü Arama: Renkli DFS Kodu

```
1  enum { BEYAZ, GRI, SIYAH };
2  int renk[N];          /* hepsi BEYAZ */
3  bool dongu(int u) {
▶    renk[u] = GRI;      /* yolda */
5    for (int v = 0; v < N; v++) {
6      if (!bekler[u][v]) continue;
7      if (renk[v] == GRI)
8        return true;    /* geri kenar */
9      if (renk[v] == BEYAZ && dongu(v))
10       return true;
11   }
12   renk[u] = SIYAH;    /* bitti */
13   return false;
14 }
```



P2 → P3, P4 sırasıyla denenir

P1 griye boyanır: şu anki yolun üzerinde.

Adım 2 / 9

DEĞİŞKENLER

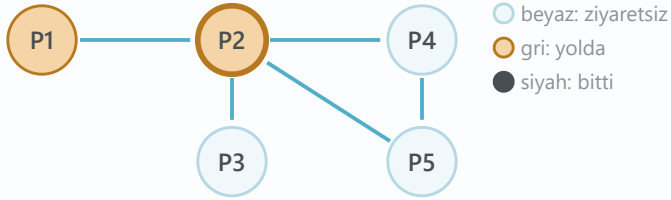
P1 = G | P2 = B | P3 = B | P4 = B | P5 = B

ÇAĞRI YIĞINI

dongu(P1) | main

Döngü Arama: Renkli DFS Kodu

```
1  enum { BEYAZ, GRI, SIYAH };
2  int renk[N];          /* hepsi BEYAZ */
3  bool dongu(int u) {
▶    renk[u] = GRI;      /* yolda */
5    for (int v = 0; v < N; v++) {
6      if (!bekler[u][v]) continue;
7      if (renk[v] == GRI)
8        return true;    /* geri kenar */
9      if (renk[v] == BEYAZ && dongu(v))
10       return true;
11   }
12   renk[u] = SIYAH;    /* bitti */
13   return false;
14 }
```



P2 → P3, P4 sırasıyla denenir

P1, P2'yi bekliyor ve P2 beyaz → `dongu(P2)` çağrılır; P2 gri olur.

Adım 3 / 9

DEĞİŞKENLER

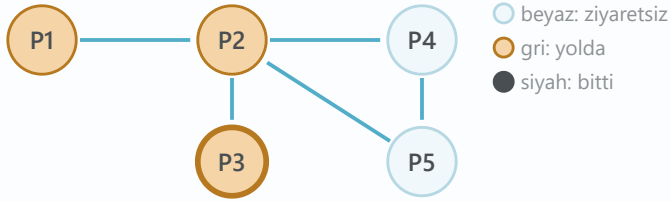
P1 = G | P2 = G | P3 = B | P4 = B | P5 = B

ÇAĞRI YIĞINI

`dongu(P2)` | `dongu(P1)` | `main`

Döngü Arama: Renkli DFS Kodu

```
1  enum { BEYAZ, GRI, SIYAH };
2  int renk[N];          /* hepsi BEYAZ */
3  bool dongu(int u) {
▶    renk[u] = GRI;      /* yolda */
5    for (int v = 0; v < N; v++) {
6      if (!bekler[u][v]) continue;
7      if (renk[v] == GRI)
8        return true;    /* geri kenar */
9      if (renk[v] == BEYAZ && dongu(v))
10       return true;
11   }
12   renk[u] = SIYAH;    /* bitti */
13   return false;
14 }
```



P2 → P3, P4 sırasıyla denenir

P2, P3'ü bekliyor ve P3 beyaz → `dongu(P3)` çağrılır; P3 gri olur.

Adım 4 / 9

DEĞİŞKENLER

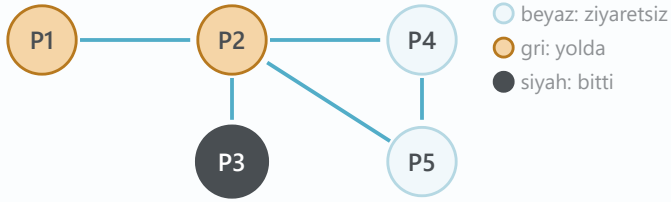
P1 = **G** | P2 = **G** | P3 = **G** | P4 = **B** | P5 = **B**

ÇAĞRI YIĞINI

`dongu(P3)` | `dongu(P2)` | `dongu(P1)` | `main`

Döngü Arama: Renkli DFS Kodu

```
1  enum { BEYAZ, GRI, SIYAH };
2  int renk[N];          /* hepsi BEYAZ */
3  bool dongu(int u) {
4    renk[u] = GRI;      /* yolda */
5    for (int v = 0; v < N; v++) {
6      if (!bekler[u][v]) continue;
7      if (renk[v] == GRI)
8        return true;    /* geri kenar */
9      if (renk[v] == BEYAZ && dongu(v))
10       return true;
11   }
▶    renk[u] = SIYAH;    /* bitti */
▶    return false;
14 }
```



P2 → P3, P4 sırasıyla denenir

P3 kimseyi beklemiyor: `for` boş biter, P3 **siyah** olur ve `false` döner; P2'ye geri dönülür.

Adım 5 / 9

DEĞİŞKENLER

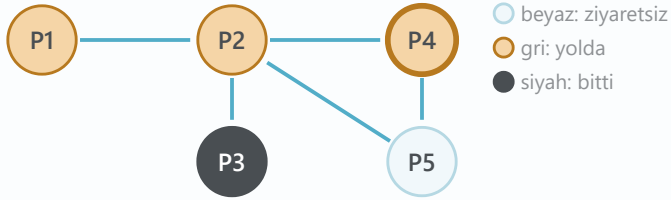
P1 = **G** | P2 = **G** | P3 = **S** | P4 = **B** | P5 = **B**

ÇAĞRI YIĞINI

`dongu(P3)` | `dongu(P2)` | `dongu(P1)` | `main`

Döngü Arama: Renkli DFS Kodu

```
1  enum { BEYAZ, GRI, SIYAH };
2  int renk[N];          /* hepsi BEYAZ */
3  bool dongu(int u) {
▶    renk[u] = GRI;      /* yolda */
5    for (int v = 0; v < N; v++) {
6      if (!bekler[u][v]) continue;
7      if (renk[v] == GRI)
8        return true;    /* geri kenar */
9      if (renk[v] == BEYAZ && dongu(v))
10       return true;
11   }
12   renk[u] = SIYAH;    /* bitti */
13   return false;
14 }
```



P2 → P3, P4 sırasıyla denenir

P2'nin sıradaki komşusu P4 beyaz → `dongu(P4)`; P4 gri.

Adım 6 / 9

DEĞİŞKENLER

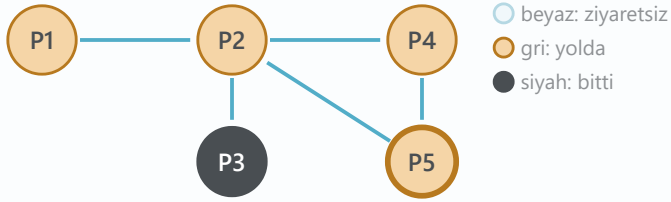
P1 = **G** | P2 = **G** | P3 = **S** | P4 = **G** | P5 = **B**

ÇAĞRI YIĞINI

`dongu(P4)` | `dongu(P2)` | `dongu(P1)` | `main`

Döngü Arama: Renkli DFS Kodu

```
1  enum { BEYAZ, GRI, SIYAH };
2  int renk[N];          /* hepsi BEYAZ */
3  bool dongu(int u) {
▶    renk[u] = GRI;      /* yolda */
5    for (int v = 0; v < N; v++) {
6      if (!bekler[u][v]) continue;
7      if (renk[v] == GRI)
8        return true;    /* geri kenar */
9      if (renk[v] == BEYAZ && dongu(v))
10       return true;
11   }
12   renk[u] = SIYAH;    /* bitti */
13   return false;
14 }
```



P2 → P3, P4 sırasıyla denenir

P4, P5'i bekliyor ve P5 beyaz → `dongu(P5)` çağrılır; P5 gri olur.

Adım 7 / 9

DEĞİŞKENLER

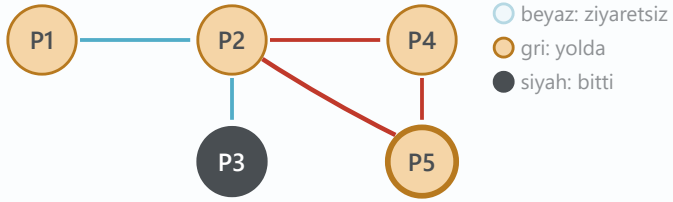
P1 = **G** | P2 = **G** | P3 = **S** | P4 = **G** | P5 = **G**

ÇAĞRI YIĞINI

`dongu(P5)` | `dongu(P4)` | `dongu(P2)` | `dongu(P1)` | `main`

Döngü Arama: Renkli DFS Kodu

```
1  enum { BEYAZ, GRI, SIYAH };
2  int renk[N];          /* hepsi BEYAZ */
3  bool dongu(int u) {
4    renk[u] = GRI;      /* yolda */
5    for (int v = 0; v < N; v++) {
6      if (!bekler[u][v]) continue;
▶      if (renk[v] == GRI)
▶        return true;    /* geri kenar */
9      if (renk[v] == BEYAZ && dongu(v))
10       return true;
11   }
12   renk[u] = SIYAH;    /* bitti */
13   return false;
14 }
```



P2 → P3, P4 sırasıyla denenir

P5 → P2 kenarında P2 **gri**: yığında hâlâ açık. Geri kenar = döngü: P2 → P4 → P5 → P2.

Adım 8 / 9

DEĞİŞKENLER

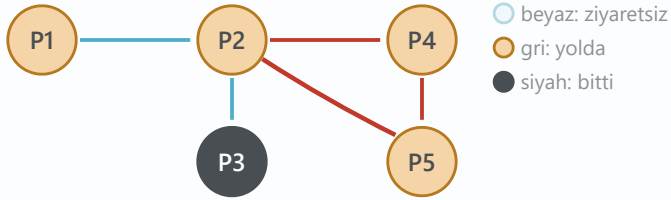
`P1 = G` `P2 = G` `P3 = S` `P4 = G` `P5 = G`

ÇAĞRI YIĞINI

`dongu(P5)` `dongu(P4)` `dongu(P2)` `dongu(P1)` `main`

Döngü Arama: Renkli DFS Kodu

```
1  enum { BEYAZ, GRI, SIYAH };
2  int renk[N];          /* hepsi BEYAZ */
3  bool dongu(int u) {
4    renk[u] = GRI;      /* yolda */
5    for (int v = 0; v < N; v++) {
6      if (!bekler[u][v]) continue;
7      if (renk[v] == GRI)
8        return true;    /* geri kenar */
▶      if (renk[v] == BEYAZ && dongu(v))
▶        return true;
11   }
12   renk[u] = SIYAH;    /* bitti */
13   return false;
14 }
```



P2 → P3, P4 sırasıyla denenir

`true` yığın boyunca yukarı taşınır. Döngüdeki P2, P4, P5 kilitlenmiştir; P1 döngüde değil ama P2'yi beklediği için o da takılıdır.

Adım 9 / 9

DEĞİŞKENLER

P1 = **G** P2 = **G** P3 = **S** P4 = **G** P5 = **G**

ÇAĞRI YIĞINI

main

Tespit: Her Türden Çok Kaynak

Var olan kaynaklar

$E = (E_1, E_2, E_3, \ldots, E_m)$

Boştaki kaynaklar

$A = (A_1, A_2, A_3, \ldots, A_m)$

Tahsis matrisi C (satır i: Pi'nin tuttukları)

$$\begin{matrix} C_{11} & C_{12} & C_{13} & \ldots & C_{1m} \\ C_{21} & C_{22} & C_{23} & \ldots & C_{2m} \\ \vdots & \vdots & & & \\ C_{n1} & C_{n2} & C_{n3} & \ldots & C_{nm} \end{matrix}$$

İstek matrisi R (satır i: Pi'nin istekleri)

$$\begin{matrix} R_{11} & R_{12} & R_{13} & \ldots & R_{1m} \\ R_{21} & R_{22} & R_{23} & \ldots & R_{2m} \\ \vdots & \vdots & & & \\ R_{n1} & R_{n2} & R_{n3} & \ldots & R_{nm} \end{matrix}$$

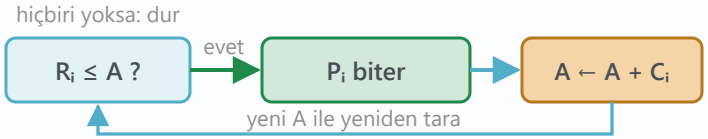
Değişmez: $\Sigma_i C_{ij} + A_j = E_j$ (her j kaynak türü için)

- n süreç, m kaynak türü. C'nin i. satırı P_i'nin *şu an tuttuğu*, R'nin i. satırı P_i'nin *şu an istediği* kaynaklardır.
- Vektör karşılaştırması: X ≤ Y ancak ve ancak **her** j için $X_j \le Y_j$. Ör. (1 0 1 0) ≤ (2 1 0 0) **değil**.

Tanenbaum MOS Şekil 6-6. Silberschatz aynı yapıları Available, Allocation, Request adlarıyla verir.

Tespit Algoritması

```
A ← boş kaynak vektörü
tüm Pi işaretsiz
tekrarla:
  Ri ≤ A olan işaretsiz bir Pi bul
  bulunamazsa: çık
  A ← A + Ci   /* Pi biter, bırakır */
  Pi'yi işaretle
işaretsiz Pi'ler kilitlenmiştir
```



- **Fikir:** isteği şu anki boş kaynaklarla karşılanabilen bir süreç *bitebilir*; bitince tuttuklarını geri verir.
- Bu iyimser bir varsayımdır: süreç ek kaynak istemeden bitecek sayılır. Yanılırsa sonraki çalıştırmada yakalanır.
- Sonda işaretsiz kalanlar **kilitlenmiştir**.
- Karmaşıklık: $O(m \cdot n^2)$.

C'nin i. satırı A'ya eklenir: **A ← A + C_i** (biten sürecin tuttukları boş havuza döner).

Tespit Örneği

```
▶ A ← (2 1 0 0)
▶ tüm Pi işaretsiz
3 tekrarla:
4   Ri ≤ A olan işaretsiz Pi bul
5   bulunamazsa: çık
6   A ← A + Ci
7   Pi'yi işaretle
8 işaretsizler kilitli
```

C (tutulan)

	Tp	Pl	Tr	CD
P1	0	0	1	0
P2	2	0	0	1
P3	0	1	2	0

R (istek)

	Tp	Pl	Tr	CD
P1	2	0	0	1
P2	1	0	1	0
P3	2	1	0	0

E = (4 2 3 1)

A	2	1	0	0

Teyp (Tp), çizici (Pl), tarayıcı (Tr), CD. E = (4 2 3 1), boşta A = (2 1 0 0).

Adım 1 / 9

DEĞİŞKENLER

`A = (2 1 0 0)` `işaretli = –`

Tanenbaum MOS Şekil 6-7. Bulunan sıra (P3, P2, P1), algoritmanın satırları hangi sırayla taradığına bağlıdır; sonuç değişmez.

Tespit Örneği

```
1 A ← (2 1 0 0)
2 tüm Pi işaretsiz
3 tekrarla:
▶   Ri ≤ A olan işaretsiz Pi bul
5   bulunamazsa: çık
6   A ← A + Ci
7   Pi'yi işaretle
8 işaretsizler kilitli
```

C (tutulan)

	Tp	Pl	Tr	CD
P1	0	0	1	0
P2	2	0	0	1
P3	0	1	2	0

R (istek)

	Tp	Pl	Tr	CD
P1	2	0	0	1
P2	1	0	1	0
P3	2	1	0	0

E = (4 2 3 1)

A	2	1	0	0

(2 0 0 1) ≤ (2 1 0 0)? CD yok ✗

P1: R_1 = (2 0 0 1). Boş CD yok → şimdilik bitemez.

Adım 2 / 9

DEĞİŞKENLER

`A = (2 1 0 0)` `işaretli = –`

Tanenbaum MOS Şekil 6-7. Bulunan sıra (P3, P2, P1), algoritmanın satırları hangi sırayla taradığına bağlıdır; sonuç değişmez.

Tespit Örneği

```
1 A ← (2 1 0 0)
2 tüm Pi işaretsiz
3 tekrarla:
▶   Ri ≤ A olan işaretsiz Pi bul
5   bulunamazsa: çık
6   A ← A + Ci
7   Pi'yi işaretle
8 işaretsizler kilitli
```

C (tutulan)

	Tp	Pl	Tr	CD
P1	0	0	1	0
P2	2	0	0	1
P3	0	1	2	0

R (istek)

	Tp	Pl	Tr	CD
P1	2	0	0	1
P2	1	0	1	0
P3	2	1	0	0

E = (4 2 3 1)

A	2	1	0	0

(1 0 1 0) ≤ (2 1 0 0)? tarayıcı yok ✗

P2: R_2 = (1 0 1 0). Boş tarayıcı yok ✗.

Adım 3 / 9

DEĞİŞKENLER

A = (2 1 0 0) | işaretli = –

Tanenbaum MOS Şekil 6-7. Bulunan sıra (P3, P2, P1), algoritmanın satırları hangi sırayla taradığına bağlıdır; sonuç değişmez.

Tespit Örneği

```
1 A ← (2 1 0 0)
2 tüm Pi işaretsiz
3 tekrarla:
▶   Ri ≤ A olan işaretsiz Pi bul
5   bulunamazsa: çık
6   A ← A + Ci
7   Pi'yi işaretle
8 işaretsizler kilitli
```

C (tutulan)

	Tp	Pl	Tr	CD
P1	0	0	1	0
P2	2	0	0	1
P3	0	1	2	0

R (istek)

	Tp	Pl	Tr	CD
P1	2	0	0	1
P2	1	0	1	0
P3	2	1	0	0

E = (4 2 3 1)

A = 2 1 0 0

(2 1 0 0) ≤ (2 1 0 0) ✓

P3: R_3 = (2 1 0 0) ≤ A ✓. P3 çalışıp bitebilir.

Adım 4 / 9

DEĞİŞKENLER

A = **(2 1 0 0)** işaretli = **–**

Tanenbaum MOS Şekil 6-7. Bulunan sıra (P3, P2, P1), algoritmanın satırları hangi sırayla taradığına bağlıdır; sonuç değişmez.

Tespit Örneği

```
1 A ← (2 1 0 0)
2 tüm Pi işaretsiz
3 tekrarla:
4   Ri ≤ A olan işaretsiz Pi bul
5   bulunamazsa: çık
▶   A ← A + Ci
▶   Pi'yi işaretle
8 işaretsizler kilitli
```

C (tutulan)

	Tp	Pl	Tr	CD
P1	0	0	1	0
P2	2	0	0	1
P3	0	1	2	0

R (istek)

	Tp	Pl	Tr	CD
P1	2	0	0	1
P2	1	0	1	0
P3	2	1	0	0

E = (4 2 3 1)

A	2	2	2	0

P3 biter, tuttuğu C_3 = (0 1 2 0) geri gelir: A = (2 2 2 0).

Adım 5 / 9

DEĞİŞKENLER

A = (2 2 2 0) işaretli = P3

Tanenbaum MOS Şekil 6-7. Bulunan sıra (P3, P2, P1), algoritmanın satırları hangi sırayla taradığına bağlıdır; sonuç değişmez.

Tespit Örneği

```
1 A ← (2 1 0 0)
2 tüm Pi işaretsiz
3 tekrarla:
▶   Ri ≤ A olan işaretsiz Pi bul
5   bulunamazsa: çık
6   A ← A + Ci
7   Pi'yi işaretle
8 işaretsizler kilitli
```

C (tutulan)

	Tp	Pl	Tr	CD
P1	0	0	1	0
P2	2	0	0	1
P3	0	1	2	0

R (istek)

	Tp	Pl	Tr	CD
P1	2	0	0	1
P2	1	0	1	0
P3	2	1	0	0

E = (4 2 3 1)

A	2	2	2	0

CD hâlâ yok ✗

Yeni tur. P1 yine bekler (CD = 0).

Adım 6 / 9

DEĞİŞKENLER

`A = (2 2 2 0)` `işaretli = P3`

Tanenbaum MOS Şekil 6-7. Bulunan sıra (P3, P2, P1), algoritmanın satırları hangi sırayla taradığına bağlıdır; sonuç değişmez.

Tespit Örneği

```
1 A ← (2 1 0 0)
2 tüm Pi işaretsiz
3 tekrarla:
▶   Ri ≤ A olan işaretsiz Pi bul
5   bulunamazsa: çık
▶   A ← A + Ci
▶   Pi'yi işaretle
8 işaretsizler kilitli
```

C (tutulan)

	Tp	Pl	Tr	CD
P1	0	0	1	0
P2	2	0	0	1
P3	0	1	2	0

R (istek)

	Tp	Pl	Tr	CD
P1	2	0	0	1
P2	1	0	1	0
P3	2	1	0	0

E = (4 2 3 1)

A	4	2	2	1

P2: (1 0 1 0) ≤ (2 2 2 0) ✓ → biter, C_2 = (2 0 0 1) döner: A = (4 2 2 1).

Adım 7 / 9

DEĞİŞKENLER

`A = (4 2 2 1)` `işaretli = P3, P2`

Tanenbaum MOS Şekil 6-7. Bulunan sıra (P3, P2, P1), algoritmanın satırları hangi sırayla taradığına bağlıdır; sonuç değişmez.

Tespit Örneği

```
1 A ← (2 1 0 0)
2 tüm Pi işaretsiz
3 tekrarla:
▶   Ri ≤ A olan işaretsiz Pi bul
5   bulunamazsa: çık
▶   A ← A + Ci
▶   Pi'yi işaretle
8 işaretsizler kilitli
```

C (tutulan)

	Tp	Pl	Tr	CD
P1	0	0	1	0
P2	2	0	0	1
P3	0	1	2	0

R (istek)

	Tp	Pl	Tr	CD
P1	2	0	0	1
P2	1	0	1	0
P3	2	1	0	0

E = (4 2 3 1)

A	4	2	3	1

P1: (2 0 0 1) ≤ (4 2 2 1) ✓ → biter: A = (4 2 3 1) = E.

Adım 8 / 9

DEĞİŞKENLER

`A = (4 2 3 1)` `işaretli = P3, P2, P1`

Tanenbaum MOS Şekil 6-7. Bulunan sıra (P3, P2, P1), algoritmanın satırları hangi sırayla taradığına bağlıdır; sonuç değişmez.

Tespit Örneği

```
1 A ← (2 1 0 0)
2 tüm Pi işaretsiz
3 tekrarla:
4   Ri ≤ A olan işaretsiz Pi bul
5   bulunamazsa: çık
6   A ← A + Ci
7   Pi'yi işaretle
▶ işaretsizler kilitli
```

C (tutulan)

	Tp	Pl	Tr	CD
P1	0	0	1	0
P2	2	0	0	1
P3	0	1	2	0

R (istek)

	Tp	Pl	Tr	CD
P1	2	0	0	1
P2	1	0	1	0
P3	2	1	0	0

E = (4 2 3 1)

A	4	2	3	1

sıra: P3 → P2 → P1

Tüm süreçler işaretlendi: **kilitlenme yok**.

Adım 9 / 9

DEĞİŞKENLER

`A = (4 2 3 1)` `işaretli = P3, P2, P1`

Tanenbaum MOS Şekil 6-7. Bulunan sıra (P3, P2, P1), algoritmanın satırları hangi sırayla taradığına bağlıdır; sonuç değişmez.

Bir İstek Fazlası: Kilitlenme

- Tek bir hücrenin (R_3'teki CD) değişmesi sonucu tamamen değiştirir.
- İlk turda hiçbir satır karşılanamazsa algoritma hemen biter.
- Çok örnekli kaynakta çizge yerine bu tarama kullanılır; döngü bilgisi kendiliğinden gelir.

C (tutulan)

	Tp	Pl	Tr	CD
P1	0	0	1	0
P2	2	0	0	1
P3	0	1	2	0

R (istek)

	Tp	Pl	Tr	CD
P1	2	0	0	1
P2	1	0	1	0
P3	2	1	0	1

E = (4 2 3 1)

A	2	1	0	0

Aynı durum, tek fark: P3 iki teyp ve bir çiziciye ek olarak **bir CD** de istiyor: R_3 = (2 1 0 1).

Bir İstek Fazlası: Kilitlenme

- Tek bir hücrenin (R_3'teki CD) değişmesi sonucu tamamen değiştirir.
- İlk turda hiçbir satır karşılanamazsa algoritma hemen biter.
- Çok örnekli kaynakta çizge yerine bu tarama kullanılır; döngü bilgisi kendiliğinden gelir.

C (tutulan)

	Tp	Pl	Tr	CD
P1	0	0	1	0
P2	2	0	0	1
P3	0	1	2	0

R (istek)

	Tp	Pl	Tr	CD
P1	2	0	0	1
P2	1	0	1	0
P3	2	1	0	1

E = (4 2 3 1)

A	2	1	0	0

P1 ✗ (CD yok)

P1 ✗ (CD gerekli, boşta yok).

Bir İstek Fazlası: Kilitlenme

- Tek bir hücrenin (R_3'teki CD) değişmesi sonucu tamamen değiştirir.
- İlk turda hiçbir satır karşılanamazsa algoritma hemen biter.
- Çok örnekli kaynakta çizge yerine bu tarama kullanılır; döngü bilgisi kendiliğinden gelir.

C (tutulan)

	Tp	Pl	Tr	CD
P1	0	0	1	0
P2	2	0	0	1
P3	0	1	2	0

R (istek)

	Tp	Pl	Tr	CD
P1	2	0	0	1
P2	1	0	1	0
P3	2	1	0	1

E = (4 2 3 1)

A	2	1	0	0

P2 ✗ (tarayıcı yok)

P2 ✗ (tarayıcı gerekli, boşta yok).

Bir İstek Fazlası: Kilitlenme

- Tek bir hücrenin (R_3'teki CD) değişmesi sonucu tamamen değiştirir.
- İlk turda hiçbir satır karşılanamazsa algoritma hemen biter.
- Çok örnekli kaynakta çizge yerine bu tarama kullanılır; döngü bilgisi kendiliğinden gelir.

C (tutulan)

	Tp	Pl	Tr	CD
P1	0	0	1	0
P2	2	0	0	1
P3	0	1	2	0

R (istek)

	Tp	Pl	Tr	CD
P1	2	0	0	1
P2	1	0	1	0
P3	2	1	0	1

E = (4 2 3 1)

A	2	1	0	0

P3 ✗ → P1, P2, P3 kilitli

P3 ✗ (CD tek ve P2'de). Hiçbir satır karşılanamıyor: **üç süreç de kilitlenmiş**.

Tespit Ne Zaman Çalıştırılmalı?

Her istekte

Bekletilen her istekten sonra. Kilitlenme anında, suçlu istekle birlikte bulunur; ama CPU maliyeti yüksektir.

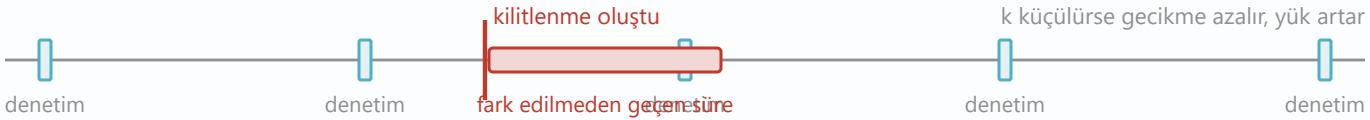
Periyodik

Her k dakikada bir. Ucuzdur; kilitlenme bir süre fark edilmeden kalır, döngüye başka süreçler de takılabilir.

Belirtiye göre

CPU kullanımı belirli bir eşiğin altına düştüğünde ya da bir kilit beklemesi uzadığında.

- Ne sıklıkla? **Kilitlenmenin ne kadar olası olduğuna** ve kaç sürecin etkileneceğine bağlı.
- Örnek: PostgreSQL bir kilidi `deadlock_timeout` (varsayılan 1 sn) kadar bekleyen işlem için bekleme çizgesini tarar; döngü bulursa bir işlemi iptal eder (`ERROR: deadlock detected`).
- Tespitin yükü ihmal edilemez; çalıştırma sıklığıyla doğru orantılıdır; ayrıca kurtarmanın maliyeti (kaybolan iş) de hesaba katılmalıdır.



TESPİT VE KURTARMA

Kilitlenmeden Kurtarma

Kaynak kesme
Kaynak sahibinden geçici olarak alınıp başka sürece verilir (ör. yazıcı işini kenara alıp kâğıtları ayırmak). Kaynağın doğasına bağlı; çoğu zaman elle ya da imkânsız.

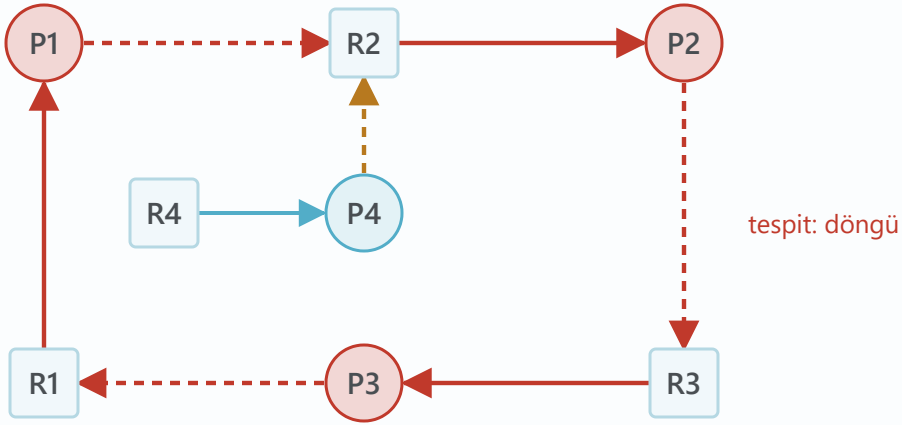
Geri sarma (rollback)
Süreçler periyodik olarak *denetim noktası* (checkpoint) kaydeder. Kurban, kaynağı almadan önceki noktaya döndürülür; sonraki iş kaybolur. Veritabanlarında işlem iptali (abort) budur.

Sonlandırma (killing)
En kaba yöntem. Ya döngüdeki tüm süreçler ya da döngü kırılana kadar *teker teker* (her öldürmeden sonra tespit yeniden çalışır). Döngü dışından bir süreç de seçilebilir.

Kurban seçimi (en düşük maliyet): öncelik, ne kadar çalıştığı ve ne kadar kaldığı, tuttuğu kaynak sayısı, bitmek için kaç kaynak daha istediği, etkileşimli mi toplu mu, yeniden başlatılabilir mi (ör. derleme evet, veritabanı güncellemesi dikkat). Aynı süreç sürekli kurban seçilirse **açlık**: geri sarma sayısı maliyete eklenir.

Önleme ve tespit, birer kurtarma yöntemi değildir. Hiçbir kurtarma kesintisiz değildir: her kurtarma yöntemi kaybedilen iş ya da yeniden çalıştırma maliyeti getirir.

Kurtarma: Adım Adım

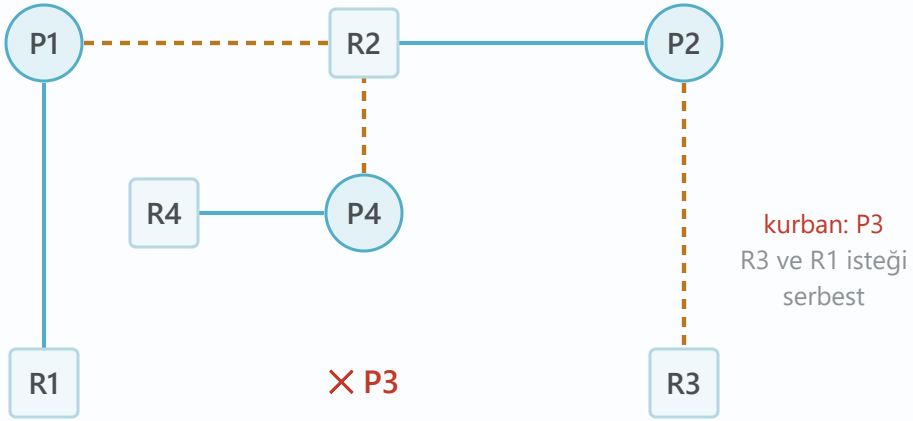


Tespit algoritması P1 → R2 → P2 → R3 → P3 → R1 → P1 döngüsünü buldu. P4 de R2'yi beklediği için takılı.

Adım 1 / 4

Teker teker sonlandırmada her kurbandan sonra tespit algoritması yeniden çalışır; döngü kırılınca durulur.

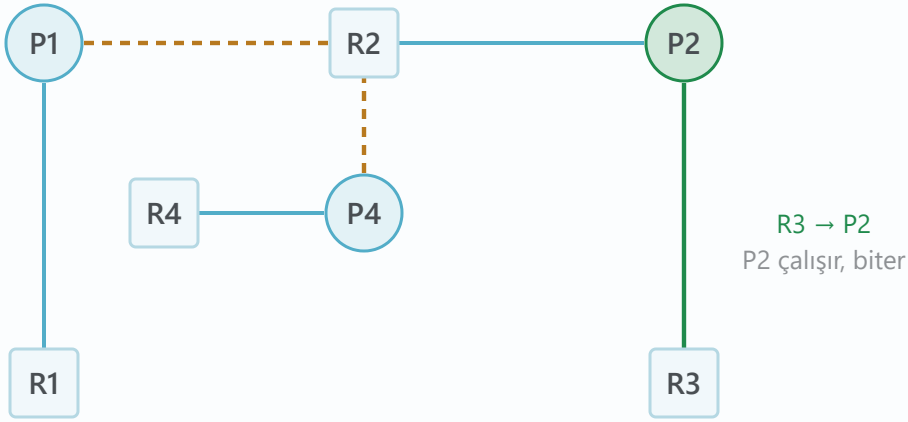
Kurtarma: Adım Adım



Kurban: en az iş kaybettirecek süreç, P3. Sonlandırılır (ya da denetim noktasına geri sarılır); tuttuğu R3 serbest kalır.

Teker teker sonlandırmada her kurbandan sonra tespit algoritması yeniden çalışır; döngü kırılınca durulur.

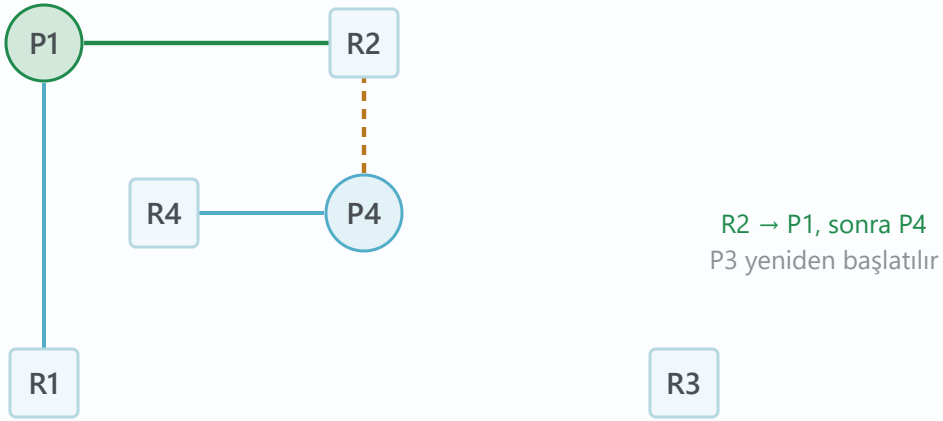
Kurtarma: Adım Adım



R3 bekleyen P2'ye verilir. P2 işini bitirip R2 ve R3'ü bırakır.

Teker teker sonlandırmada her kurbandan sonra tespit algoritması yeniden çalışır; döngü kırılınca durulur.

Kurtarma: Adım Adım



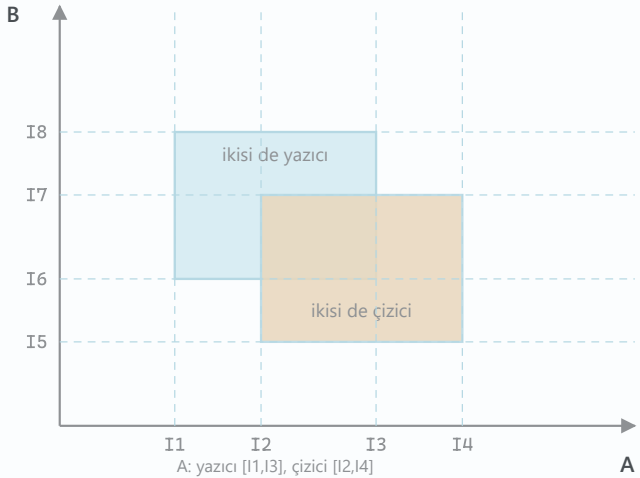
R2 sıradaki bekleyene (P1) verilir; P1 bitince P4. P3 baştan (ya da denetim noktasından) yeniden çalıştırılır.

Teker teker sonlandırmada her kurbandan sonra tespit algoritması yeniden çalışır; döngü kırılınca durulur.

KAÇINMA

Kaynak Yörüngeleri

- A: I1–I3 arası yazıcı, I2–I4 arası çizici kullanır.
- B: I5–I7 arası çizici, I6–I8 arası yazıcı kullanır.
- Taralı bölgeler karşılıklı dışlama nedeniyle **girilemez**.
- Kaçınma = yörüngeyi güvensiz bölgeye sokacak isteği **ertelemek**.



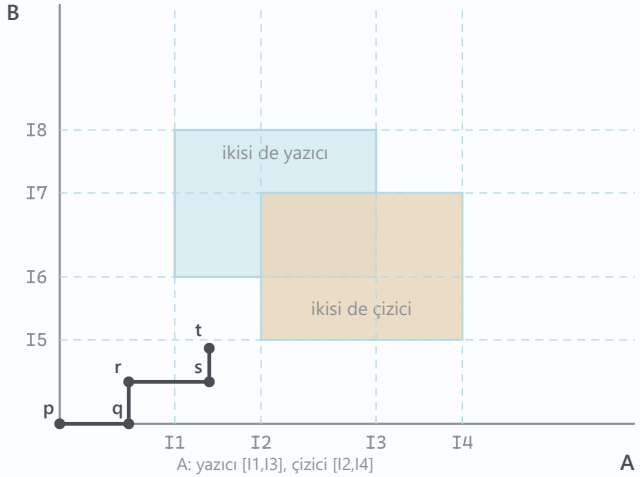
Yatay eksen A'nın, dikey eksen B'nin yürüttüğü komutlar. Tek CPU: yörünge yalnızca **sağa** (A çalışır) ya da **yukarı** (B çalışır) ilerler.

Adım 1 / 4

Tanenbaum MOS Şekil 6-8 (Dijkstra'nın iki süreçli gösterimi).

Kaynak Yörüngeleri

- A: I1–I3 arası yazıcı, I2–I4 arası çizici kullanır.
- B: I5–I7 arası çizici, I6–I8 arası yazıcı kullanır.
- Taralı bölgeler karşılıklı dışlama nedeniyle **girilemez**.
- Kaçınma = yörüngeyi güvensiz bölgeye sokacak isteği **ertelemek**.



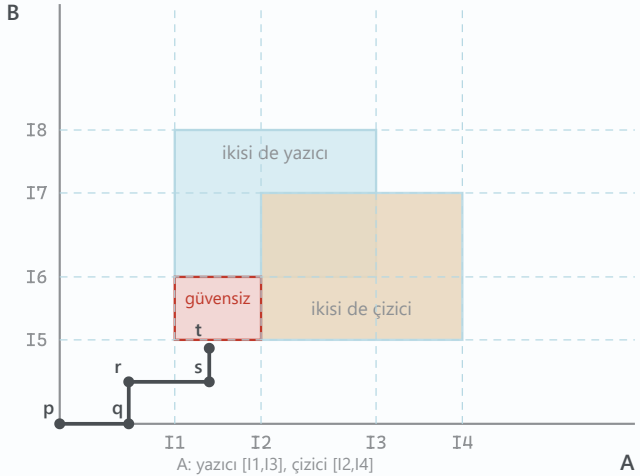
p → q: A çalışır; q → r: B çalışır; r → s: A yazıcıyı alır (I1). s → t: B ilerler, I5'te çiziciyi isteyecek.

Adım 2 / 4

Tanenbaum MOS Şekil 6-8 (Dijkstra'nın iki süreçli gösterimi).

Kaynak Yörüngeleri

- A: I1–I3 arası yazıcı, I2–I4 arası çizici kullanır.
- B: I5–I7 arası çizici, I6–I8 arası yazıcı kullanır.
- Taralı bölgeler karşılıklı dışlama nedeniyle **girilemez**.
- Kaçınma = yörüngeyi güvensiz bölgeye sokacak isteği **ertelemek**.



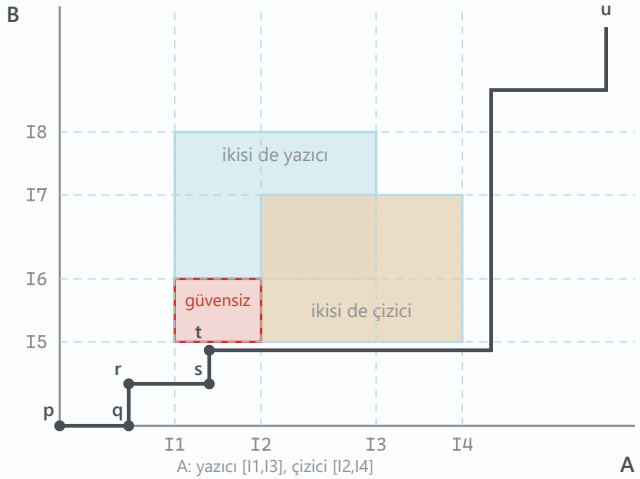
Kırmızı kutuya girilirse A yazıcıyı, B çiziciyi tutar ve ikisi de taralı bölgeye girmek zorundadır: kilitlenme **kaçınılmaz**. Bu bölge **güvensizdir**.

Tanenbaum MOS Şekil 6-8 (Dijkstra'nın iki süreçli gösterimi).

KAÇINMA

Kaynak Yörüngeleri

- A: I1–I3 arası yazıcı, I2–I4 arası çizici kullanır.
- B: I5–I7 arası çizici, I6–I8 arası yazıcı kullanır.
- Taralı bölgeler karşılıklı dışlama nedeniyle **girilemez**.
- Kaçınma = yörüngeyi güvensiz bölgeye sokacak isteği **ertelemek**.



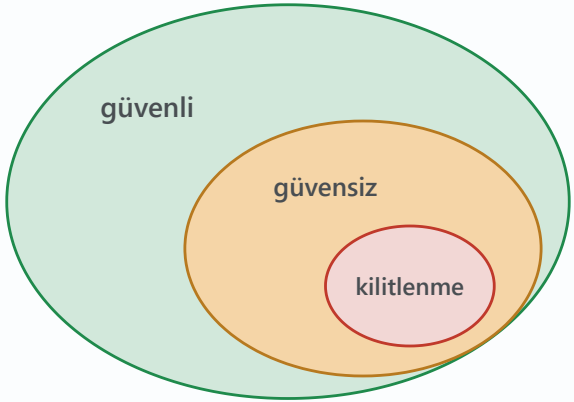
t noktasında İS, B'nin çizici isteğini erteler; A I4'e kadar çalışıp ikisini de bırakır. Sonra B devam eder: kilitlenme yok.

Adım 4 / 4

Tanenbaum MOS Şekil 6-8 (Dijkstra'nın iki süreçli gösterimi).

Güvenli ve Güvensiz Durumlar

- **Durum:** E, A, C ve azami ihtiyaçlar (Max) ile belirlenir.
- **Güvenli durum:** her süreç anında azami isteğini yapsa bile, tüm süreçleri *bir sırayla* sonuna kadar çalıştırabilecek bir **güvenli dizi** $<P_1, \ldots, P_n>$ vardır.
- P_i'nin kalan ihtiyacı ≤ boştakiler + P_j ($j < i$) tarafından tutulanlar.
- **Güvensiz ≠ kilitlenmiş.** Güvensiz durumda henüz kilitlenme yoktur; süreçler azami isteğini yapmazsa her şey yolunda gidebilir. Ama sistem artık bitişi *garanti* edemez.
- Kaçınma algoritması sistemi her zaman güvenli durumda tutar.



Güvenli Durum Örneği

- Tek kaynak türü, 10 örnek.
- **Has:** şu an tutulan; **Max:** bitene kadar gerekecek en çok.
- Kalan ihtiyaç = Max − Has.
- İS her adımda isteklerin dizisini kendisi seçebilecek konumdadır.

	Has	Max
A	3	9
B	2	4
C	2	7

Boş: 3

(a) Toplam 10 örnek; 7'si dağıtılmış, 3 boş. Güvenli mi? Bir güvenli dizi arayalım.

Tanenbaum MOS Şekil 6-9.

Güvenli Durum Örneği

- Tek kaynak türü, 10 örnek.
- **Has:** şu an tutulan; **Max:** bitene kadar gerekecek en çok.
- Kalan ihtiyaç = Max − Has.
- İS her adımda isteklerin dizisini kendisi seçebilecek konumdadır.

Tanenbaum MOS Şekil 6-9.

	Has	Max
A	3	9
B	4	4
C	2	7

Boş: 1

(b) B'nin kalan ihtiyacı 2 ≤ 3: B'ye 2 verilir, B azamisine ulaşır.

Güvenli Durum Örneği

- Tek kaynak türü, 10 örnek.
- **Has:** şu an tutulan; **Max:** bitene kadar gerekecek en çok.
- Kalan ihtiyaç = Max − Has.
- İS her adımda isteklerin dizisini kendisi seçebilecek konumdadır.

Tanenbaum MOS Şekil 6-9.

	Has	Max
A	3	9
B	0	–
C	2	7

Boş: 5

(c) B biter ve 4 örneği bırakır: boş 5.

Güvenli Durum Örneği

- Tek kaynak türü, 10 örnek.
- **Has:** şu an tutulan; **Max:** bitene kadar gerekecek en çok.
- Kalan ihtiyaç = Max − Has.
- İS her adımda isteklerin dizisini kendisi seçebilecek konumdadır.

Tanenbaum MOS Şekil 6-9.

	Has	Max
A	3	9
B	0	–
C	7	7

Boş: 0

(d) C'nin kalan ihtiyacı 5 ≤ 5: C'ye 5 verilir.

Güvenli Durum Örneği

- Tek kaynak türü, 10 örnek.
- **Has:** şu an tutulan; **Max:** bitene kadar gerekecek en çok.
- Kalan ihtiyaç = Max − Has.
- İS her adımda isteklerin dizisini kendisi seçebilecek konumdadır.

	Has	Max
A	3	9
B	0	–
C	0	–

Boş: 7

güvenli dizi: B, C, A

(e) C biter: boş 7. A'nın kalan 6'sı karşılanır. **B → C → A** güvenli dizisi var: durum güvenli.

Tanenbaum MOS Şekil 6-9.

KAÇINMA

Güvensiz Durum Örneği

- Fark tek bir karar: (a)'da A'ya fazladan bir örnek vermek.
- (b) henüz kilitlenme değildir. A, ek isteğini yapmadan bir kaynağı bırakırsa sorun çıkmayabilir.
- Ama İS en kötü durumu varsayar: herkes azami isteğini yaparsa bitiş garanti değil.

	Has	Max
A	3	9
B	2	4
C	2	7

Boş: 3

Aynı başlangıç (güvenli). Şimdi A bir örnek daha ister ve İS bunu *verir*.

Adım 1 / 4

Tanenbaum MOS Şekil 6-10.

Güvensiz Durum Örneği

- Fark tek bir karar: (a)'da A'ya fazladan bir örnek vermek.
- (b) henüz kilitlenme değildir. A, ek isteğini yapmadan bir kaynağı bırakırsa sorun çıkmayabilir.
- Ama İS en kötü durumu varsayar: herkes azami isteğini yaparsa bitiş garanti değil.

Tanenbaum MOS Şekil 6-10.

	Has	Max
A	4	9
B	2	4
C	2	7

Boş: 2

(b) A: 4/9, boş 2. Güvenli dizi var mı?

Güvensiz Durum Örneği

- Fark tek bir karar: (a)'da A'ya fazladan bir örnek vermek.
- (b) henüz kilitlenme değildir. A, ek isteğini yapmadan bir kaynağı bırakırsa sorun çıkmayabilir.
- Ama İS en kötü durumu varsayar: herkes azami isteğini yaparsa bitiş garanti değil.

	Has	Max
A	4	9
B	4	4
C	2	7

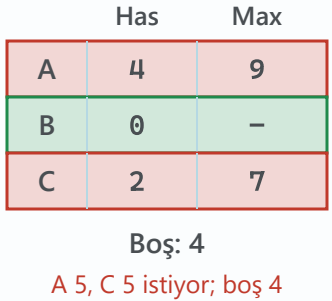
Boş: 0

(c) Yalnızca B bitebilir (ihtiyaç 2 ≤ 2): B'ye 2 verilir.

Tanenbaum MOS Şekil 6-10.

Güvensiz Durum Örneği

- Fark tek bir karar: (a)'da A'ya fazladan bir örnek vermek.
- (b) henüz kilitlenme değildir. A, ek isteğini yapmadan bir kaynağı bırakırsa sorun çıkmayabilir.
- Ama İS en kötü durumu varsayar: herkes azami isteğini yaparsa bitiş garanti değil.



(d) B biter: boş 4. A'nın 5, C'nin 5 ihtiyacı var: hiçbiri garanti edilemez. (b) durumu **güvensizdi**; A'nın isteği ertelenmeliydi.

Tanenbaum MOS Şekil 6-10.

Bankacı Algoritması: Tek Kaynak

- Her istekte: “Bu isteği verirsem durum **güvenli** kalır mı?”
- Evet → ver. Hayır → **ertele** (süreç bekler, istek sonra yeniden değerlendirilir).
- Güvenlik: kalan ihtiyacı boştakilerle karşılanabilen bir müşteri bul, bitmiş say, kredisini kasaya ekle; tekrarla.

	Has	Max
A	0	6
B	0	5
C	0	4
D	0	7

Boş: 10

Max toplamı 22 > 10, yine de güvenli

(a) Bankacı (Dijkstra, 1965): kasada 10 birim, 4 müşterinin kredi limitleri toplamı 22. Herkes aynı anda azami istemez.

Durum, yalnızca *bir* süreç değil **tüm** süreçler bir sırayla bitebiliyorsa güvenlidir. Bankacı kilitlenmeyi sonradan çözmez: sistemi hiç kilitlenmeye sokmaz.

Bankacı Algoritması: Tek Kaynak

- Her istekte: “Bu isteği verirsem durum **güvenli** kalır mı?”
- Evet → ver. Hayır → **ertele** (süreç bekler, istek sonra yeniden değerlendirilir).
- Güvenlik: kalan ihtiyacı boştakilerle karşılanabilen bir müşteri bul, bitmiş say, kredisini kasaya ekle; tekrarla.

	Has	Max
A	1	6
B	1	5
C	2	4
D	4	7

Boş: 2

güvenli dizi: C, D, A, B

(b) Boş 2. C’nin ihtiyacı 2 ≤ 2 → C biter (boş 4), sonra D (3 ≤ 4 → boş 8), A, B. **Güvenli.**

Durum, yalnızca *bir* süreç değil **tüm** süreçler bir sırayla bitebiliyorsa güvenlidir. Bankacı kilitlenmeyi sonradan çözmez: sistemi hiç kilitlenmeye sokmaz.

Bankacı Algoritması: Tek Kaynak

- Her istekte: “Bu isteği verirsem durum **güvenli** kalır mı?”
- Evet → ver. Hayır → **ertele** (süreç bekler, istek sonra yeniden değerlendirilir).
- Güvenlik: kalan ihtiyacı boştakilerle karşılanabilen bir müşteri bul, bitmiş say, kredisini kasaya ekle; tekrarla.

	Has	Max
A	1	6
B	2	5
C	2	4
D	4	7

Boş: 1

hiçbir ihtiyaç ≤ 1 değil: güvensiz

(c) (b)’deyken B bir birim daha isterse: boş 1, en küçük ihtiyaç 2. **Güvensiz** → istek reddedilip B bekletilmelidir.

Durum, yalnızca *bir* süreç değil **tüm** süreçler bir sırayla bitebiliyorsa güvenlidir. Bankacı kilitlenmeyi sonradan çözmez: sistemi hiç kilitlenmeye sokmaz.

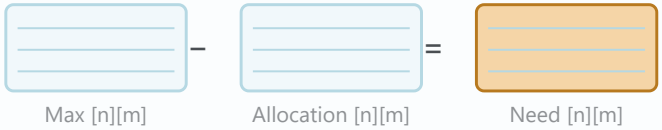
Bankacı Algoritması: Çoklu Kaynak

- `Available[m]` : her türden boş örnek.
- `Max[n][m]` : her sürecin **önceden bildirdiği** azami istek.
- `Allocation[n][m]` : şu an tutulan.
- `Need = Max − Allocation` : kalan ihtiyaç.

```
/* güvenlik algoritması */
Work ← Available
Finish[i] ← false  (tüm i)
while Finish[i] = false ve
      Need[i] ≤ Work olan i var:
  Work ← Work + Allocation[i]
  Finish[i] ← true
tüm i için Finish[i] ise GÜVENLİ
```

```
/* Ti, Request[i] isterse */
if Request[i] > Need[i]:
  hata  /* Max aşıldı */
if Request[i] > Available:
  Ti bekler  /* kaynak yok */
/* varmış gibi yap: */
Available  -= Request[i]
Allocation[i] += Request[i]
Need[i]    -= Request[i]
if güvenli: isteği ver
else: geri al, Ti bekler
```

Karmaşıklık: güvenlik testi $O(m \cdot n^2)$; her istekte çalışır.



Güvenlik Algoritması: Örnek

```
▶ Work ← Available
▶ Finish[i] ← false
3 for i = T0 … T4 (tekrar tekrar):
4   if Finish[i] = false and
5       Need[i] ≤ Work:
6     Work ← Work + Allocation[i]
7     Finish[i] ← true
8 if tüm Finish[i]: GÜVENLİ
```

Allocation

	A	B	C
T0	0	1	0
T1	2	0	0
T2	3	0	2
T3	2	1	1
T4	0	0	2

Need

	A	B	C
T0	7	4	3
T1	1	2	2
T2	6	0	0
T3	0	1	1
T4	4	3	1

Work

3	3	2

Sıra: —

A=10, B=5, C=7 örnek. Available = (3 3 2). Need = Max − Allocation hesaplandı.

Adım 1 / 9

DEĞİŞKENLER

`Work = (3 3 2)` `Sıra = –`

Silberschatz 10e §8.6.3.3 örneği. Max: T0 (7 5 3), T1 (3 2 2), T2 (9 0 2), T3 (2 2 2), T4 (4 3 3).

Güvenlik Algoritması: Örnek

```
1 Work ← Available
2 Finish[i] ← false
3 for i = T0 … T4 (tekrar tekrar):
▶   if Finish[i] = false and
▶      Need[i] ≤ Work:
6     Work ← Work + Allocation[i]
7     Finish[i] ← true
8 if tüm Finish[i]: GÜVENLİ
```

Allocation

	A	B	C
T0	0	1	0
T1	2	0	0
T2	3	0	2
T3	2	1	1
T4	0	0	2

Need

	A	B	C
T0	7	4	3
T1	1	2	2
T2	6	0	0
T3	0	1	1
T4	4	3	1

Work

3	3	2

Sıra: —

(7 4 3) ≤ (3 3 2)? ✗

T0: Need (7 4 3) ≤ (3 3 2)? Hayır, geç.

Adım 2 / 9

DEĞİŞKENLER

`Work = (3 3 2)` `Sıra = –`

Silberschatz 10e §8.6.3.3 örneği. Max: T0 (7 5 3), T1 (3 2 2), T2 (9 0 2), T3 (2 2 2), T4 (4 3 3).

Güvenlik Algoritması: Örnek

```
1 Work ← Available
2 Finish[i] ← false
3 for i = T0 … T4 (tekrar tekrar):
4   if Finish[i] = false and
5      Need[i] ≤ Work:
▶     Work ← Work + Allocation[i]
▶     Finish[i] ← true
8 if tüm Finish[i]: GÜVENLİ
```

Allocation

	A	B	C
T0	0	1	0
T1	2	0	0
T2	3	0	2
T3	2	1	1
T4	0	0	2

Need

	A	B	C
T0	7	4	3
T1	1	2	2
T2	6	0	0
T3	0	1	1
T4	4	3	1

Work

5	3	2

Sıra: T1

(1 2 2) ≤ (3 3 2) ✓ Work += (2 0 0)

T1: (1 2 2) ≤ (3 3 2) ✓ → biter, Work = (3 3 2) + (2 0 0) = (5 3 2).

Adım 3 / 9

DEĞİŞKENLER

`Work = (5 3 2)` `Sıra = T1`

Silberschatz 10e §8.6.3.3 örneği. Max: T0 (7 5 3), T1 (3 2 2), T2 (9 0 2), T3 (2 2 2), T4 (4 3 3).

Güvenlik Algoritması: Örnek

```
1 Work ← Available
2 Finish[i] ← false
3 for i = T0 … T4 (tekrar tekrar):
▶   if Finish[i] = false and
▶      Need[i] ≤ Work:
6     Work ← Work + Allocation[i]
7     Finish[i] ← true
8 if tüm Finish[i]: GÜVENLİ
```

Allocation

	A	B	C
T0	0	1	0
T1	2	0	0
T2	3	0	2
T3	2	1	1
T4	0	0	2

Need

	A	B	C
T0	7	4	3
T1	1	2	2
T2	6	0	0
T3	0	1	1
T4	4	3	1

Work

5	3	2

Sıra: T1

(6 0 0) ≤ (5 3 2)? ✗

T2: (6 0 0) ≤ (5 3 2)? A yetmez ✗.

Adım 4 / 9

DEĞİŞKENLER

`Work = (5 3 2)` `Sıra = T1`

Silberschatz 10e §8.6.3.3 örneği. Max: T0 (7 5 3), T1 (3 2 2), T2 (9 0 2), T3 (2 2 2), T4 (4 3 3).

Güvenlik Algoritması: Örnek

```
1 Work ← Available
2 Finish[i] ← false
3 for i = T0 … T4 (tekrar tekrar):
4   if Finish[i] = false and
5      Need[i] ≤ Work:
▶     Work ← Work + Allocation[i]
▶     Finish[i] ← true
8 if tüm Finish[i]: GÜVENLİ
```

Allocation

	A	B	C
T0	0	1	0
T1	2	0	0
T2	3	0	2
T3	2	1	1
T4	0	0	2

Need

	A	B	C
T0	7	4	3
T1	1	2	2
T2	6	0	0
T3	0	1	1
T4	4	3	1

Work

7	4	3

Sıra: T1 → T3

(0 1 1) ≤ (5 3 2) ✓ Work += (2 1 1)

T3: (0 1 1) ✓ → Work = (7 4 3).

Adım 5 / 9

DEĞİŞKENLER

`Work = (7 4 3)` `Sıra = T1, T3`

Silberschatz 10e §8.6.3.3 örneği. Max: T0 (7 5 3), T1 (3 2 2), T2 (9 0 2), T3 (2 2 2), T4 (4 3 3).

Güvenlik Algoritması: Örnek

```
1 Work ← Available
2 Finish[i] ← false
3 for i = T0 … T4 (tekrar tekrar):
4   if Finish[i] = false and
5      Need[i] ≤ Work:
▶     Work ← Work + Allocation[i]
▶     Finish[i] ← true
8 if tüm Finish[i]: GÜVENLİ
```

Allocation

	A	B	C
T0	0	1	0
T1	2	0	0
T2	3	0	2
T3	2	1	1
T4	0	0	2

Need

	A	B	C
T0	7	4	3
T1	1	2	2
T2	6	0	0
T3	0	1	1
T4	4	3	1

Work

7	4	5

Sıra: T1 → T3 → T4

(4 3 1) ≤ (7 4 3) ✓ Work += (0 0 2)

T4: (4 3 1) ✓ → Work = (7 4 5).

Adım 6 / 9

DEĞİŞKENLER

`Work = (7 4 5)` `Sıra = T1, T3, T4`

Silberschatz 10e §8.6.3.3 örneği. Max: T0 (7 5 3), T1 (3 2 2), T2 (9 0 2), T3 (2 2 2), T4 (4 3 3).

Güvenlik Algoritması: Örnek

```
1 Work ← Available
2 Finish[i] ← false
3 for i = T0 … T4 (tekrar tekrar):
4   if Finish[i] = false and
5      Need[i] ≤ Work:
      Work ← Work + Allocation[i]
      Finish[i] ← true
8 if tüm Finish[i]: GÜVENLİ
```

Allocation

	A	B	C
T0	0	1	0
T1	2	0	0
T2	3	0	2
T3	2	1	1
T4	0	0	2

Need

	A	B	C
T0	7	4	3
T1	1	2	2
T2	6	0	0
T3	0	1	1
T4	4	3	1

Work

7	5	5

Sıra: T1 → T3 → T4 → T0

ikinci tur: T0 (7 4 3) ≤ (7 4 5) ✓

İkinci tur. T0: (7 4 3) ≤ (7 4 5) ✓ → Work = (7 5 5).

DEĞİŞKENLER

`Work = (7 5 5)` `Sıra = T1, T3, T4, T0`

Silberschatz 10e §8.6.3.3 örneği. Max: T0 (7 5 3), T1 (3 2 2), T2 (9 0 2), T3 (2 2 2), T4 (4 3 3).

Güvenlik Algoritması: Örnek

```
1 Work ← Available
2 Finish[i] ← false
3 for i = T0 … T4 (tekrar tekrar):
4   if Finish[i] = false and
5      Need[i] ≤ Work:
▶     Work ← Work + Allocation[i]
▶     Finish[i] ← true
8 if tüm Finish[i]: GÜVENLİ
```

Allocation

	A	B	C
T0	0	1	0
T1	2	0	0
T2	3	0	2
T3	2	1	1
T4	0	0	2

Need

	A	B	C
T0	7	4	3
T1	1	2	2
T2	6	0	0
T3	0	1	1
T4	4	3	1

Work

10	5	7

Sıra: T1 → T3 → T4 → T0 → T2

T2 (6 0 0) ≤ (7 5 5) ✓ Work = toplam

T2: (6 0 0) ✓ → Work = (10 5 7), yani tüm kaynaklar.

Adım 8 / 9

DEĞİŞKENLER

`Work = (10 5 7)` `Sıra = T1, T3, T4, T0, T2`

Silberschatz 10e §8.6.3.3 örneği. Max: T0 (7 5 3), T1 (3 2 2), T2 (9 0 2), T3 (2 2 2), T4 (4 3 3).

Güvenlik Algoritması: Örnek

```
1 Work ← Available
2 Finish[i] ← false
3 for i = T0 … T4 (tekrar tekrar):
4   if Finish[i] = false and
5      Need[i] ≤ Work:
6     Work ← Work + Allocation[i]
7     Finish[i] ← true
▶ if tüm Finish[i]: GÜVENLİ
```

Allocation

	A	B	C
T0	0	1	0
T1	2	0	0
T2	3	0	2
T3	2	1	1
T4	0	0	2

Need

	A	B	C
T0	7	4	3
T1	1	2	2
T2	6	0	0
T3	0	1	1
T4	4	3	1

Work

10	5	7

Sıra: T1 → T3 → T4 → T0 → T2

GÜVENLİ

Tüm Finish true: durum **güvenli**. <T1, T3, T4, T2, T0> gibi başka güvenli diziler de vardır.

Adım 9 / 9

DEĞİŞKENLER

`Work = (10 5 7)` `Sıra = T1, T3, T4, T0, T2`

Silberschatz 10e §8.6.3.3 örneği. Max: T0 (7 5 3), T1 (3 2 2), T2 (9 0 2), T3 (2 2 2), T4 (4 3 3).

İstek Kabul: T1 (1 0 2)

- Üç adım: **Need** kontrolü, **Available** kontrolü, **güvenlik** testi.
- Güvenlik testi “varmış gibi” durumda yapılır; durum güvensiz çıksaydı değişiklikler geri alınırdı.
- Hesaplar: Python betiğiyle doğrulandı.

Allocation

	A	B	C
T0	0	1	0
T1	2	0	0
T2	3	0	2
T3	2	1	1
T4	0	0	2

Need

	A	B	C
T0	7	4	3
T1	1	2	2
T2	6	0	0
T3	0	1	1
T4	4	3	1

Avail

3	3	2

$Request_1 = (1\ 0\ 2) \leq Need_1 = (1\ 2\ 2)$ ✓

T1, (1 0 2) ister. 1. kontrol: bildirdiği azami ihtiyacı aşmıyor ✓.

İstek Kabul: T1 (1 0 2)

- Üç adım: **Need** kontrolü, **Available** kontrolü, **güvenlik** testi.
- Güvenlik testi “varmış gibi” durumda yapılır; durum güvensiz çıksaydı değişiklikler geri alınırdı.
- Hesaplar: Python betiğiyle doğrulandı.

Allocation

	A	B	C
T0	0	1	0
T1	2	0	0
T2	3	0	2
T3	2	1	1
T4	0	0	2

Need

	A	B	C
T0	7	4	3
T1	1	2	2
T2	6	0	0
T3	0	1	1
T4	4	3	1

Avail

3	3	2

(1 0 2) ≤ Available = (3 3 2) ✓

2. kontrol: kaynaklar şu an boşta ✓. Ama bu tek başına yetmez.

İstek Kabul: T1 (1 0 2)

- Üç adım: **Need** kontrolü, **Available** kontrolü, **güvenlik** testi.
- Güvenlik testi “varmış gibi” durumda yapılır; durum güvensiz çıksaydı değişiklikler geri alınırdı.
- Hesaplar: Python betiğiyle doğrulandı.

Allocation

	A	B	C
T0	0	1	0
T1	3	0	2
T2	3	0	2
T3	2	1	1
T4	0	0	2

Need

	A	B	C
T0	7	4	3
T1	0	2	0
T2	6	0	0
T3	0	1	1
T4	4	3	1

Avail

2	3	0

varmış gibi: Avail (2 3 0), $Alloc_1$ (3 0 2), $Need_1$ (0 2 0)

İstek **verilmiş gibi** yeni durum hesaplanır.

İstek Kabul: T1 (1 0 2)

- Üç adım: **Need** kontrolü, **Available** kontrolü, **güvenlik** testi.
- Güvenlik testi “varmış gibi” durumda yapılır; durum güvensiz çıksaydı değişiklikler geri alınırdı.
- Hesaplar: Python betiğiyle doğrulandı.

Allocation

	A	B	C
T0	0	1	0
T1	3	0	2
T2	3	0	2
T3	2	1	1
T4	0	0	2

Need

	A	B	C
T0	7	4	3
T1	0	2	0
T2	6	0	0
T3	0	1	1
T4	4	3	1

Work

10	5	7

Sıra: T1 → T3 → T4 → T0 → T2

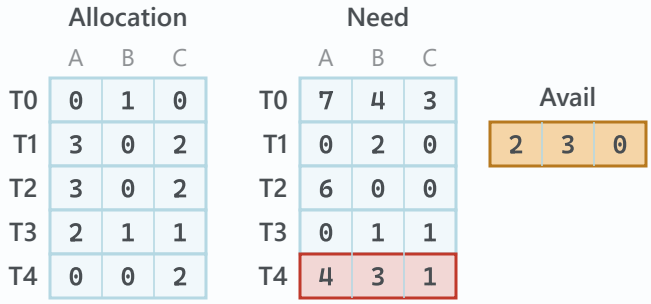
yeni durum güvenli → istek VERİLİR

Güvenlik algoritması yeni durumda <T1, T3, T4, T0, T2> dizisini bulur: **güvenli → istek verilir**.

KAÇINMA

İstek Reddi: T4 (3 3 0) ve T0 (0 2 0)

- **Bekleme nedeni 1:** kaynak yok (T4).
- **Bekleme nedeni 2:** kaynak var ama vermek durumu güvensiz yapar (T0).
- Kaçınmanın bedeli budur: kaynaklar bazen boşta beklerken istek ertelenir; kullanım oranı düşer.



T4: (3 3 0) ≤ Avail (2 3 0)? ✗ → bekle

T1'in isteği verildikten sonra T4, (3 3 0) ister. Need'i aşmıyor ama A türünden yalnızca 2 boş: **T4 bekler**.

İstek Reddi: T4 (3 3 0) ve T0 (0 2 0)

- **Bekleme nedeni 1:** kaynak yok (T4).
- **Bekleme nedeni 2:** kaynak var ama vermek durumu güvensiz yapar (T0).
- Kaçınmanın bedeli budur: kaynaklar bazen boşta beklerken istek ertelenir; kullanım oranı düşer.

Allocation

	A	B	C
T0	0	1	0
T1	3	0	2
T2	3	0	2
T3	2	1	1
T4	0	0	2

Need

	A	B	C
T0	7	4	3
T1	0	2	0
T2	6	0	0
T3	0	1	1
T4	4	3	1

Avail

2	3	0

T0: (0 2 0) ≤ $Need_0$ ✓ ve ≤ Avail (2 3 0) ✓

T0, (0 2 0) ister. İki kontrol de geçer; kaynak *fiilen* boşta.

İstek Reddi: T4 (3 3 0) ve T0 (0 2 0)

- **Bekleme nedeni 1:** kaynak yok (T4).
- **Bekleme nedeni 2:** kaynak var ama vermek durumu güvensiz yapar (T0).
- Kaçınmanın bedeli budur: kaynaklar bazen boşta beklerken istek ertelenir; kullanım oranı düşer.

Allocation

	A	B	C
T0	0	3	0
T1	3	0	2
T2	3	0	2
T3	2	1	1
T4	0	0	2

Need

	A	B	C
T0	7	2	3
T1	0	2	0
T2	6	0	0
T3	0	1	1
T4	4	3	1

Work

2	1	0

varmış gibi: Avail (2 1 0), $Need_0$ (7 2 3)

Varmış gibi: Available = (2 1 0). Hangi Need ≤ (2 1 0)? T1 (0 2 0): B yetmez; T3 (0 1 1): C yetmez…

İstek Reddi: T4 (3 3 0) ve T0 (0 2 0)

- **Bekleme nedeni 1:** kaynak yok (T4).
- **Bekleme nedeni 2:** kaynak var ama vermek durumu güvensiz yapar (T0).
- Kaçınmanın bedeli budur: kaynaklar bazen boşta beklerken istek ertelenir; kullanım oranı düşer.

Allocation

	A	B	C
T0	0	3	0
T1	3	0	2
T2	3	0	2
T3	2	1	1
T4	0	0	2

Need

	A	B	C
T0	7	2	3
T1	0	2	0
T2	6	0	0
T3	0	1	1
T4	4	3	1

Work

2	1	0

Sıra: —

hiçbir satır uymuyor → GÜVENSİZ → REDDET

Hiçbir süreç bitirilemiyor: durum **güvensiz**. Değişiklikler geri alınır, T0 bekletilir; kaynak boşta olduğu hâlde.

Bankacı İstek Algoritması: Kod İzleme

```
1  /* Ti, Request isterse */
▶  if Request > Need[i]: hata /* Max aşıldı */
3  if Request > Available: Ti bekler
4  Available -= Request  /* varmış gibi */
5  Allocation[i] += Request
6  Need[i] -= Request
7  if guvenli(): isteği VER
8  else: geri al, Ti bekler
9  guvenli():
10   Work ← Available; Finish ← false
11   for i = T0 … T4 (tekrar tekrar):
12     if !Finish[i] and Need[i] ≤ Work:
13       Work ← Work + Allocation[i]
14       Finish[i] ← true
15   return tüm Finish[i]
```

	Allocation	Need	Finish
T0	0 1 0	7 4 3	—
T1	2 0 0	1 2 2	—
T2	3 0 2	6 0 0	—
T3	2 1 1	0 1 1	—
T4	0 0 2	4 3 1	—

Durum güvenli bir başlangıç: Available = (3 3 2). T3, **Request = (0 1 1)** ister. Need[3] = (0 1 1): istek azami bildirimi aşmıyor.

Adım 1 / 10

DEĞİŞKENLER

`Request = (0 1 1)` `Need[3] = (0 1 1)` `Available = (3 3 2)`

Aynı Silberschatz durumu; yalnız istek farklı. Değerler Python simülasyonuyla doğrulandı.

Bankacı İstek Algoritması: Kod İzleme

```
1  /* Ti, Request isterse */
2  if Request > Need[i]: hata /* Max aşıldı */
▶  if Request > Available: Ti bekler
4  Available -= Request  /* varmış gibi */
5  Allocation[i] += Request
6  Need[i] -= Request
7  if guvenli(): isteği VER
8  else: geri al, Ti bekler
9  guvenli():
10   Work ← Available; Finish ← false
11   for i = T0 … T4 (tekrar tekrar):
12     if !Finish[i] and Need[i] ≤ Work:
13       Work ← Work + Allocation[i]
14       Finish[i] ← true
15   return tüm Finish[i]
```

	Allocation	Need	Finish
T0	0 1 0	7 4 3	—
T1	2 0 0	1 2 2	—
T2	3 0 2	6 0 0	—
T3	2 1 1	0 1 1	—
T4	0 0 2	4 3 1	—

(0 1 1) ≤ Available (3 3 2): kaynaklar şu an var. Yine de hemen verilmez.

Adım 2 / 10

DEĞİŞKENLER

`Request = (0 1 1)` `Need[3] = (0 1 1)` `Available = (3 3 2)`

Aynı Silberschatz durumu; yalnız istek farklı. Değerler Python simülasyonuyla doğrulandı.

Bankacı İstek Algoritması: Kod İzleme

```
1  /* Ti, Request isterse */
2  if Request > Need[i]: hata /* Max aşıldı */
3  if Request > Available: Ti bekler
▶  Available -= Request  /* varmış gibi */
▶  Allocation[i] += Request
▶  Need[i] -= Request
7  if guvenli(): isteği VER
8  else: geri al, Ti bekler
9  guvenli():
10   Work ← Available; Finish ← false
11   for i = T0 … T4 (tekrar tekrar):
12     if !Finish[i] and Need[i] ≤ Work:
13       Work ← Work + Allocation[i]
14       Finish[i] ← true
15   return tüm Finish[i]
```

	Allocation	Need	Finish
T0	0 1 0	7 4 3	—
T1	2 0 0	1 2 2	—
T2	3 0 2	6 0 0	—
T3	2 2 2	0 0 0	—
T4	0 0 2	4 3 1	—

Varmış gibi yapılır: Available = **(3 2 1)**, Allocation[3] = (2 2 2), Need[3] = (0 0 0).

Adım 3 / 10

DEĞİŞKENLER

`Available = (3 2 1)`

`Allocation[3] = (2 2 2)`

`Need[3] = (0 0 0)`

Aynı Silberschatz durumu; yalnız istek farklı. Değerler Python simülasyonuyla doğrulandı.

Bankacı İstek Algoritması: Kod İzleme

```
1  /* Ti, Request isterse */
2  if Request > Need[i]: hata /* Max aşıldı */
3  if Request > Available: Ti bekler
4  Available -= Request  /* varmış gibi */
5  Allocation[i] += Request
6  Need[i] -= Request
▶  if guvenli(): isteği VER
8  else: geri al, Ti bekler
9  guvenli():
▶    Work ← Available; Finish ← false
11   for i = T0 … T4 (tekrar tekrar):
12     if !Finish[i] and Need[i] ≤ Work:
13       Work ← Work + Allocation[i]
14       Finish[i] ← true
15   return tüm Finish[i]
```

	Allocation	Need	Finish
T0	0 1 0	7 4 3	—
T1	2 0 0	1 2 2	—
T2	3 0 2	6 0 0	—
T3	2 2 2	0 0 0	—
T4	0 0 2	4 3 1	—

Work (3 2 1)

Güvenlik testi: Work = (3 2 1). T0 (7 4 3), T1 (1 2 2), T2 (6 0 0) bu Work ile bitemez.

Adım 4 / 10

DEĞİŞKENLER

`Work = (3 2 1)` `Sıra = –`

Aynı Silberschatz durumu; yalnız istek farklı. Değerler Python simülasyonuyla doğrulandı.

Bankacı İstek Algoritması: Kod İzleme

```
1  /* Ti, Request isterse */
2  if Request > Need[i]: hata /* Max aşıldı */
3  if Request > Available: Ti bekler
4  Available -= Request  /* varmış gibi */
5  Allocation[i] += Request
6  Need[i] -= Request
7  if guvenli(): isteği VER
8  else: geri al, Ti bekler
9  guvenli():
10   Work ← Available; Finish ← false
11   for i = T0 … T4 (tekrar tekrar):
▶      if !Finish[i] and Need[i] ≤ Work:
▶        Work ← Work + Allocation[i]
▶        Finish[i] ← true
15   return tüm Finish[i]
```

	Allocation	Need	Finish
T0	0 1 0	7 4 3	—
T1	2 0 0	1 2 2	—
T2	3 0 2	6 0 0	—
T3	2 2 2	0 0 0	✓
T4	0 0 2	4 3 1	—

Work (5 4 3)

T3: Need (0 0 0) ≤ Work (3 2 1) ✓ → biter, Work = (3 2 1) + (2 2 2) = **(5 4 3)**.

Adım 5 / 10

DEĞİŞKENLER

`Work = (5 4 3)` `Sıra = T3`

Aynı Silberschatz durumu; yalnız istek farklı. Değerler Python simülasyonuyla doğrulandı.

Bankacı İstek Algoritması: Kod İzleme

```
1  /* Ti, Request isterse */
2  if Request > Need[i]: hata /* Max aşıldı */
3  if Request > Available: Ti bekler
4  Available -= Request  /* varmış gibi */
5  Allocation[i] += Request
6  Need[i] -= Request
7  if guvenli(): isteği VER
8  else: geri al, Ti bekler
9  guvenli():
10   Work ← Available; Finish ← false
11   for i = T0 … T4 (tekrar tekrar):
▶      if !Finish[i] and Need[i] ≤ Work:
▶        Work ← Work + Allocation[i]
▶        Finish[i] ← true
15   return tüm Finish[i]
```

	Allocation	Need	Finish
T0	0 1 0	7 4 3	—
T1	2 0 0	1 2 2	—
T2	3 0 2	6 0 0	—
T3	2 2 2	0 0 0	✓
T4	0 0 2	4 3 1	✓

Work (5 4 5)

T4: Need (4 3 1) ≤ Work (5 4 3) ✓ → biter, Work = (5 4 3) + (0 0 2) = **(5 4 5)**.

Adım 6 / 10

DEĞİŞKENLER

`Work = (5 4 5)` `Sıra = T3, T4`

Aynı Silberschatz durumu; yalnız istek farklı. Değerler Python simülasyonuyla doğrulandı.

Bankacı İstek Algoritması: Kod İzleme

```
1  /* Ti, Request isterse */
2  if Request > Need[i]: hata /* Max aşıldı */
3  if Request > Available: Ti bekler
4  Available -= Request  /* varmış gibi */
5  Allocation[i] += Request
6  Need[i] -= Request
7  if guvenli(): isteği VER
8  else: geri al, Ti bekler
9  guvenli():
10   Work ← Available; Finish ← false
11   for i = T0 … T4 (tekrar tekrar):
▶      if !Finish[i] and Need[i] ≤ Work:
▶        Work ← Work + Allocation[i]
▶        Finish[i] ← true
15   return tüm Finish[i]
```

	Allocation	Need	Finish
T0	0 1 0	7 4 3	—
T1	2 0 0	1 2 2	✓
T2	3 0 2	6 0 0	—
T3	2 2 2	0 0 0	✓
T4	0 0 2	4 3 1	✓

Work (7 4 5)

İkinci tur. T0 (7 4 3) için A yetmez; T1: Need (1 2 2) ≤ Work (5 4 5) ✓ → biter, Work = (5 4 5) + (2 0 0) = **(7 4 5)**.

Adım 7 / 10

DEĞİŞKENLER

`Work = (7 4 5)` `Sıra = T3, T4, T1`

Aynı Silberschatz durumu; yalnız istek farklı. Değerler Python simülasyonuyla doğrulandı.

Bankacı İstek Algoritması: Kod İzleme

```
1  /* Ti, Request isterse */
2  if Request > Need[i]: hata /* Max aşıldı */
3  if Request > Available: Ti bekler
4  Available -= Request  /* varmış gibi */
5  Allocation[i] += Request
6  Need[i] -= Request
7  if guvenli(): isteği VER
8  else: geri al, Ti bekler
9  guvenli():
10   Work ← Available; Finish ← false
11   for i = T0 … T4 (tekrar tekrar):
▶      if !Finish[i] and Need[i] ≤ Work:
▶        Work ← Work + Allocation[i]
▶        Finish[i] ← true
15   return tüm Finish[i]
```

	Allocation	Need	Finish
T0	0 1 0	7 4 3	—
T1	2 0 0	1 2 2	✓
T2	3 0 2	6 0 0	✓
T3	2 2 2	0 0 0	✓
T4	0 0 2	4 3 1	✓

Work (10 4 7)

T2: Need (6 0 0) ≤ Work (7 4 5) ✓ → biter, Work = (7 4 5) + (3 0 2) = **(10 4 7)**.

Adım 8 / 10

DEĞİŞKENLER

`Work = (10 4 7)` `Sıra = T3, T4, T1, T2`

Aynı Silberschatz durumu; yalnız istek farklı. Değerler Python simülasyonuyla doğrulandı.

Bankacı İstek Algoritması: Kod İzleme

```
1  /* Ti, Request isterse */
2  if Request > Need[i]: hata /* Max aşıldı */
3  if Request > Available: Ti bekler
4  Available -= Request  /* varmış gibi */
5  Allocation[i] += Request
6  Need[i] -= Request
7  if guvenli(): isteği VER
8  else: geri al, Ti bekler
9  guvenli():
10   Work ← Available; Finish ← false
11   for i = T0 … T4 (tekrar tekrar):
▶      if !Finish[i] and Need[i] ≤ Work:
▶        Work ← Work + Allocation[i]
▶        Finish[i] ← true
15   return tüm Finish[i]
```

	Allocation	Need	Finish
T0	0 1 0	7 4 3	✓
T1	2 0 0	1 2 2	✓
T2	3 0 2	6 0 0	✓
T3	2 2 2	0 0 0	✓
T4	0 0 2	4 3 1	✓

Work (10 5 7)

Üçüncü tur: T0: Need (7 4 3) ≤ Work (10 4 7) ✓ → biter, Work = (10 4 7) + (0 1 0) = **(10 5 7)**.

Adım 9 / 10

DEĞİŞKENLER

`Work = (10 5 7)`

`Sıra = T3, T4, T1, T2, T0`

Aynı Silberschatz durumu; yalnız istek farklı. Değerler Python simülasyonuyla doğrulandı.

Bankacı İstek Algoritması: Kod İzleme

```
1  /* Ti, Request isterse */
2  if Request > Need[i]: hata /* Max aşıldı */
3  if Request > Available: Ti bekler
4  Available -= Request  /* varmış gibi */
5  Allocation[i] += Request
6  Need[i] -= Request
▶  if guvenli(): isteği VER
8  else: geri al, Ti bekler
9  guvenli():
10   Work ← Available; Finish ← false
11   for i = T0 … T4 (tekrar tekrar):
12     if !Finish[i] and Need[i] ≤ Work:
13       Work ← Work + Allocation[i]
14       Finish[i] ← true
▶    return tüm Finish[i]
```

	Allocation	Need	Finish
T0	0 1 0	7 4 3	✓
T1	2 0 0	1 2 2	✓
T2	3 0 2	6 0 0	✓
T3	2 2 2	0 0 0	✓
T4	0 0 2	4 3 1	✓

Work (10 5 7)

Tüm Finish true → güvenli dizi <T3, T4, T1, T2, T0>. İstek **verilir**; durum kalıcı olur.

Adım 10 / 10

DEĞİŞKENLER

`Work = (10 5 7)`

`Sıra = T3, T4, T1, T2, T0` `sonuç = VER`

Aynı Silberschatz durumu; yalnız istek farklı. Değerler Python simülasyonuyla doğrulandı.

KAÇINMA

Çoklu Kaynak: İkinci Örnek

Atanmış

	Tp	Pl	Yz	CD
A	3	0	1	1
B	0	1	0	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0

Hâlâ gereken

	Tp	Pl	Yz	CD
A	1	1	0	0
B	0	1	1	2
C	3	1	0	0
D	0	0	1	0
E	2	1	1	0

```
E = (6 3 4 2)
P = (5 3 2 2)
A = (1 0 2 0)
```

E = var olan, P = atanmış toplamı, A = E − P (boşta)

Beş süreç, dört kaynak türü (Tp, Pl, Yz=yazıcı, CD). A = E − P = (1 0 2 0).

Adım 1 / 4

Tanenbaum MOS Şekil 6-12. “Her satırda ≤ A olan satırı ara” adımları burada güvenlik testidir; aynı mantık tespit algoritmasında *istek* matrisiyle, kaçınmada *kalan ihtiyaç* matrisiyle kullanılır.

Çoklu Kaynak: İkinci Örnek

Atanmış

	Tp	Pl	Yz	CD
A	3	0	1	1
B	0	1	0	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0

Hâlâ gereken

	Tp	Pl	Yz	CD
A	1	1	0	0
B	0	1	1	2
C	3	1	0	0
D	0	0	1	0
E	2	1	1	0

```
E = (6 3 4 2)
P = (5 3 2 2)
A = (1 0 2 0)
```

güvenli dizi: D, E, A, B, C

D (0 0 1 0) ≤ A → biter, A = (2 1 2 1); E, A, B, C de sırayla biter. Durum güvenli.

Tanenbaum MOS Şekil 6-12. “Her satırda ≤ A olan satırı ara” adımları burada güvenlik testidir; aynı mantık tespit algoritmasında *istek* matrisiyle, kaçınmada *kalan ihtiyaç* matrisiyle kullanılır.

Çoklu Kaynak: İkinci Örnek

Atanmış

	Tp	Pl	Yz	CD
A	3	0	1	1
B	0	1	1	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0

Hâlâ gereken

	Tp	Pl	Yz	CD
A	1	1	0	0
B	0	1	0	2
C	3	1	0	0
D	0	0	1	0
E	2	1	1	0

```
E = (6 3 4 2)
P = (5 3 3 2)
A = (1 0 1 0)
```

B bir yazıcı ister → A = (1 0 1 0), dizi D, E, A, B, C hâlâ var → VER

B bir yazıcı ister. Verilirse A = (1 0 1 0); D yine bitebilir ve dizi korunur: **verilir**.

Tanenbaum MOS Şekil 6-12. “Her satırda ≤ A olan satırı ara” adımları burada güvenlik testidir; aynı mantık tespit algoritmasında *istek* matrisiyle, kaçınmada *kalan ihtiyaç* matrisiyle kullanılır.

Çoklu Kaynak: İkinci Örnek

Atanmış

	Tp	Pl	Yz	CD
A	3	0	1	1
B	0	1	1	0
C	1	1	1	0
D	1	1	0	1
E	0	0	1	0

Hâlâ gereken

	Tp	Pl	Yz	CD
A	1	1	0	0
B	0	1	0	2
C	3	1	0	0
D	0	0	1	0
E	2	1	0	0

```
E = (6 3 4 2)
P = (5 3 4 2)
A = (1 0 0 0)
```

sonra E son yazıcıyı ister → A = (1 0 0 0), kimse bitemez → ERTELE

Ardından E son yazıcıyı isterse A = (1 0 0 0) olur ve hiçbir satır karşılanamaz: güvensiz → **ertelenir**.

Adım 4 / 4

Tanenbaum MOS Şekil 6-12. “Her satırda ≤ A olan satırı ara” adımları burada güvenlik testidir; aynı mantık tespit algoritmasında *istek* matrisiyle, kaçınmada *kalan ihtiyaç* matrisiyle kullanılır.

Bankacı Algoritması Pratikte

Varsayımlar

Her süreç azami ihtiyacını *önceden* bildirir; süreç sayısı sabittir; kaynak sayısı değişmez (aygıt bozulmaz); süreçler kaynakları sonlu sürede bırakır.

Gerçek sistemler

Bu varsayımlar genel amaçlı İS'lerde tutmaz; algoritma neredeyse hiç kullanılmaz. Benzer "kabul denetimi" fikri: gerçek zamanlı çizelgeleme, bulut kotaları, havuzlu bağlantı limitleri.

- Kaçınma, her istekte $O(m \cdot n^2)$ test ve gereksiz ertelemeler nedeniyle tespit ve kurtarmadan genellikle **daha pahalıdır**; avantajı kurtarma gerektirmemesidir.
- Algoritma arka planda sürekli çalışmaz: test yalnızca **bir istek geldiğinde** (ve bir kaynak bırakıldığında bekleyenler için) yapılır.

Etkileşimli simülatör: Kilitlenme ve Bankacı Algoritması Simülatörü: Allocation, Max, Need ve Available değerleriyle güvenli dizi ve istek kontrolü adım adım.

Önleme: Koşullara Saldırmak

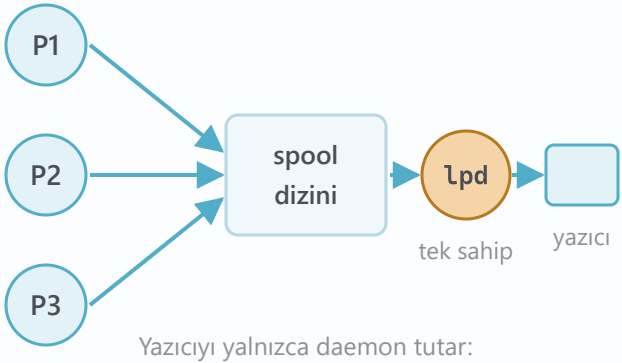
Koşul	Saldırı	Bedeli
Karşılıklı dışlama	Kaynağı paylaşılabilir yap: biriktirme (spooling), salt okunur paylaşım, kilitsiz yapılar	Çoğu kaynak doğası gereği paylaşılamaz (mutex!)
Tut ve bekle	Tüm kaynakları başta tek seferde iste; ya da yeni istekten önce elindekileri bırak	Düşük kullanım, açlık; ihtiyaç önceden bilinmeli
Kesme yok	Bekleyecek sürecin kaynaklarını geri al; `tryLock` + geri çekil; sanallaştır	Yalnızca durumu kaydedilebilen kaynaklarda; canlı kilit riski
Döngüsel bekleme	Kaynaklara toplam sıra ver, yalnızca artan sırada iste (kilit sıralaması)	Sıraya uymak programcı disiplini ister

Karşılıklı dışlamayı kaldırmak “her seferinde yalnızca bir süreç kaynak isteyebilir” demek **değildir** (bu dışlamanın kendisidir). “Kesme yok” koşulunu kaldırmak da “süreç bitene kadar kaynakları tutsun” değil, tersine **kaynakların geri alınabilmesidir**.

ÖNLEME

Karşılıklı Dışlamaya Saldırı

- **Biriktirme (spooling):** süreçler çıktıyı diske yazar; yazıcıyı tek bir daemon sırayla kullanır. Yazıcı için çekişme kalkar.
- **Salt okunur** dosyalar ve kod sayfaları aynı anda paylaşılır; okur kilitleri birbirini dışlamaz.
- **Kilitsiz (lock-free)** yapılar CAS ile ilerler; kilit olmadığı için kilitlenme de olmaz (açlık hâlâ mümkün).
- Sınır: biriktirme diski de bir kaynaktır. Disk, yarım kalmış çıktılarla dolarsa yine kilitlenme olabilir (Tanenbaum).
- Genel ilke: bir kaynağı gerçekten gerekmedikçe **atama**, onu isteyecek süreç sayısını **en aza indir**.



Yazıcıyı yalnızca daemon tutar: kullanıcı süreçleri onu hiç beklemez

ÖNLEME

Tut ve Beklemeye Saldırı

- **Protokol 1:** süreç ihtiyaç duyacağı *tüm* kaynakları çalışmaya başlamadan önce tek bir istekte alır (hepsi ya da hiçbiri).
- **Protokol 2:** süreç yalnızca elinde hiçbir kaynak yokken istek yapabilir: yenisini istemeden önce tuttuklarını **bırakır**.
- Örnek: DVD → disk kopyala, diski sırala, yazıcıya bas. Protokol 1'de yazıcı en baştan sona kadar boşa tutulur.
- Sorunlar: **düşük kaynak kullanımı**; çok kaynak isteyen süreç için **açlık**; ihtiyaçlar çoğu zaman önceden bilinmez.
- Pratikte: `pthread_mutex_trylock` ile "ya hepsi ya hiçbiri" edinimi; veritabanlarında tüm kilitleri baştan alan muhafazakâr 2PL.



Silberschatz 10e §8.5.2.

ÖNLEME

Kesme Yok Koşuluna Saldırı

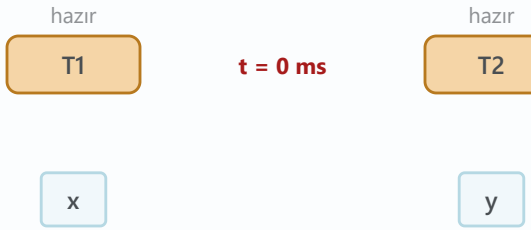
- **Geri alma protokolü:** kaynak tutan bir süreç hemen karşılanamayan bir istek yaparsa, tuttuğu tüm kaynaklar *örtük olarak bırakılır*; süreç hepsini birlikte alabildiğinde yeniden başlar.
- Yalnızca durumu kaydedilip geri yüklenebilen kaynaklara uygundur: CPU yazmaçları, bellek, veritabanı işlemi (abort).
- **Sanallaştırma:** yazıcı biriktirilir, bellek sayfalanır.
- Mutex için: **zaman aşımlı deneme**. Edinemeyen iş parçacığı elindekini kendisi bırakır ve rastgele bir süre sonra yeniden dener.
- Çalıştırıldı (JDK 21): 2 iş parçacığı × 1000 ters yönlü transfer, sonuç `x=1000 y=1000 toplam=2000`.

```
boolean dene(Hesap k, Hesap h, int m)
        throws InterruptedException {
  if (!k.kilit.tryLock(50, MILLISECONDS))
    return false;
  try {
    if (!h.kilit.tryLock(50, MILLISECONDS))
      return false;  /* k'yi de bırak */
    try { k.bakiye -= m; h.bakiye += m;
          return true; }
    finally { h.kilit.unlock(); }
  } finally { k.kilit.unlock(); }
}
/* çağıran: while (!dene(x, y, 1))
              Thread.sleep(rastgele); */
```

Rastgele bekleme önemli: iki iş parçacığı aynı ritimde dener-bırakırsa **canlı kilit** oluşur.

tryLock ile Geri Alma: Kod İzleme

```
▶ boolean dene(Hesap k, Hesap h, int m)
2         throws InterruptedException {
3   if (!k.kilit.tryLock(50, MILLISECONDS))
4     return false;
5   try {
6     if (!h.kilit.tryLock(50, MILLISECONDS))
7       return false;  /* k'yi de bırak */
8     try { k.bakiye -= m; h.bakiye += m;
9           return true; }
10    finally { h.kilit.unlock(); }
11  } finally { k.kilit.unlock(); }
12 }
13 /* çağıran: while (!dene(x, y, 1))
14               Thread.sleep(rastgele); */
```



T1 x'ten y'ye, T2 y'den x'e 1 aktaracak: kilitleri **ters sırada** isterler. İki kilit boş.

Adım 1 / 10

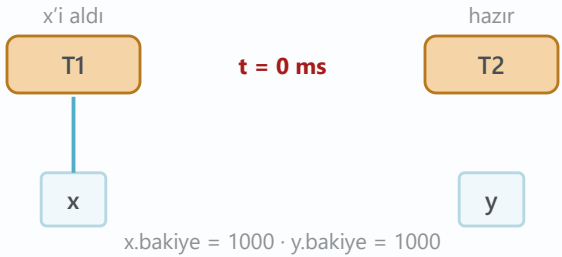
DEĞİŞKENLER

`t = 0 ms` `x = boş` `y = boş` `x.bakiye = 1000` `y.bakiye = 1000`

ÇIKTI

tryLock ile Geri Alma: Kod İzleme

```
1  boolean dene(Hesap k, Hesap h, int m)
2          throws InterruptedException {
▶  if (!k.kilit.tryLock(50, MILLISECONDS))
4    return false;
5  try {
6    if (!h.kilit.tryLock(50, MILLISECONDS))
7      return false;  /* k'yi de bırak */
8    try { k.bakiye -= m; h.bakiye += m;
9          return true; }
10   finally { h.kilit.unlock(); }
11 } finally { k.kilit.unlock(); }
12 }
13 /* çağıran: while (!dene(x, y, 1))
14               Thread.sleep(rastgele); */
```



T1 `x` 'i hemen alır.

Adım 2 / 10

DEĞİŞKENLER

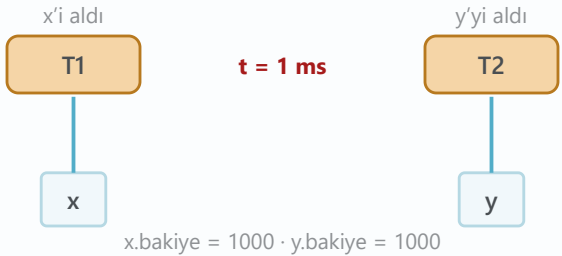
`t = 0 ms` `x = T1` `y = boş`

`x.bakiye = 1000` `y.bakiye = 1000`

ÇIKTI

tryLock ile Geri Alma: Kod İzleme

```
1  boolean dene(Hesap k, Hesap h, int m)
2          throws InterruptedException {
▶  if (!k.kilit.tryLock(50, MILLISECONDS))
4    return false;
5  try {
6    if (!h.kilit.tryLock(50, MILLISECONDS))
7      return false;  /* k'yi de bırak */
8    try { k.bakiye -= m; h.bakiye += m;
9          return true; }
10   finally { h.kilit.unlock(); }
11 } finally { k.kilit.unlock(); }
12 }
13 /* çağıran: while (!dene(x, y, 1))
14               Thread.sleep(rastgele); */
```



T2 `y`'yi alır. Düz `lock()` olsaydı sıradaki adımda kilitlenme kesinleşirdi.

Adım 3 / 10

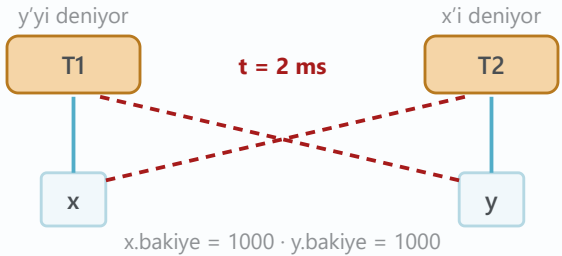
DEĞİŞKENLER

`t = 1 ms` `x = T1` `y = T2` `x.bakiye = 1000` `y.bakiye = 1000`

ÇIKTI

tryLock ile Geri Alma: Kod İzleme

```
1  boolean dene(Hesap k, Hesap h, int m)
2          throws InterruptedException {
3    if (!k.kilit.tryLock(50, MILLISECONDS))
4      return false;
5    try {
▶      if (!h.kilit.tryLock(50, MILLISECONDS))
7        return false;  /* k'yi de bırak */
8      try { k.bakiye -= m; h.bakiye += m;
9            return true; }
10     finally { h.kilit.unlock(); }
11   } finally { k.kilit.unlock(); }
12 }
13 /* çağıran: while (!dene(x, y, 1))
14               Thread.sleep(rastgele); */
```



İkisi de ikinci kilidi **en fazla 50 ms** bekler: döngüsel bekleme oluştu ama süreli.

Adım 4 / 10

DEĞİŞKENLER

`t = 2 ms` `x = T1` `y = T2` `x.bakiye = 1000` `y.bakiye = 1000`

ÇIKTI

tryLock ile Geri Alma: Kod İzleme

```
1  boolean dene(Hesap k, Hesap h, int m)
2          throws InterruptedException {
3    if (!k.kilit.tryLock(50, MILLISECONDS))
4      return false;
5    try {
6      if (!h.kilit.tryLock(50, MILLISECONDS))
▶        return false;  /* k'yi de bırak */
8      try { k.bakiye -= m; h.bakiye += m;
9            return true; }
10     finally { h.kilit.unlock(); }
11   } finally { k.kilit.unlock(); }
12 }
13 /* çağıran: while (!dene(x, y, 1))
14               Thread.sleep(rastgele); */
```



T1'in 50 ms'si doldu: `tryLock` false → `return false`.

Adım 5 / 10

DEĞİŞKENLER

`t = 52 ms` `x = T1` `y = T2`

`x.bakiye = 1000` `y.bakiye = 1000`

ÇIKTI

```
T1: false
```

tryLock ile Geri Alma: Kod İzleme

```
1  boolean dene(Hesap k, Hesap h, int m)
2          throws InterruptedException {
3    if (!k.kilit.tryLock(50, MILLISECONDS))
4      return false;
5    try {
6      if (!h.kilit.tryLock(50, MILLISECONDS))
7        return false;  /* k'yi de bırak */
8      try { k.bakiye -= m; h.bakiye += m;
9            return true; }
10     finally { h.kilit.unlock(); }
▶    } finally { k.kilit.unlock(); }
12 }
13 /* çağıran: while (!dene(x, y, 1))
14               Thread.sleep(rastgele); */
```



`finally` T1'in tuttuğu `x`'i **kendisi bırakır**: kesme yok koşulu gönüllü olarak kırıldı.

Adım 6 / 10

DEĞİŞKENLER

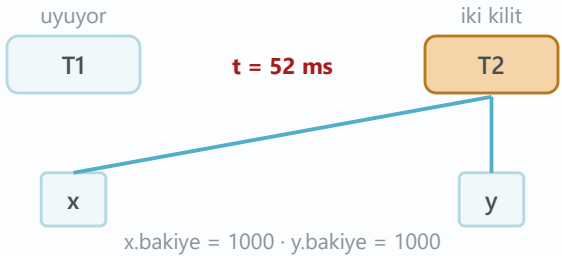
`t = 52 ms` `x = boş` `y = T2` `x.bakiye = 1000` `y.bakiye = 1000`

ÇIKTI

```
T1: false
```

tryLock ile Geri Alma: Kod İzleme

```
1  boolean dene(Hesap k, Hesap h, int m)
2          throws InterruptedException {
3    if (!k.kilit.tryLock(50, MILLISECONDS))
4      return false;
5    try {
▶      if (!h.kilit.tryLock(50, MILLISECONDS))
7        return false;  /* k'yi de bırak */
8      try { k.bakiye -= m; h.bakiye += m;
9            return true; }
10     finally { h.kilit.unlock(); }
11   } finally { k.kilit.unlock(); }
12 }
13 /* çağıran: while (!dene(x, y, 1))
14               Thread.sleep(rastgele); */
```



T2 hâlâ kendi 50 ms'si içinde: `x` boşalınca alır.

Adım 7 / 10

DEĞİŞKENLER

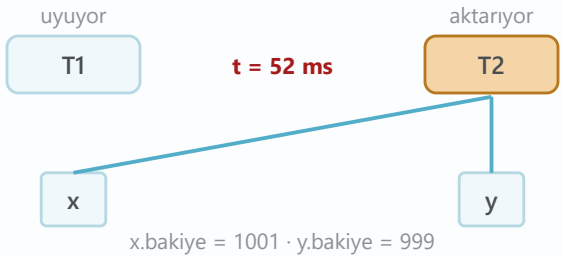
`t = 52 ms` `x = T2` `y = T2` `x.bakiye = 1000` `y.bakiye = 1000`

ÇIKTI

```
T1: false
```

tryLock ile Geri Alma: Kod İzleme

```
1  boolean dene(Hesap k, Hesap h, int m)
2          throws InterruptedException {
3    if (!k.kilit.tryLock(50, MILLISECONDS))
4      return false;
5    try {
6      if (!h.kilit.tryLock(50, MILLISECONDS))
7        return false;  /* k'yi de bırak */
▶      try { k.bakiye -= m; h.bakiye += m;
9            return true; }
10     finally { h.kilit.unlock(); }
11   } finally { k.kilit.unlock(); }
12 }
13 /* çağıran: while (!dene(x, y, 1))
14               Thread.sleep(rastgele); */
```



T2 aktarır: y.bakiye = 999, x.bakiye = 1001.

Adım 8 / 10

DEĞİŞKENLER

`t = 52 ms` `x = T2` `y = T2`

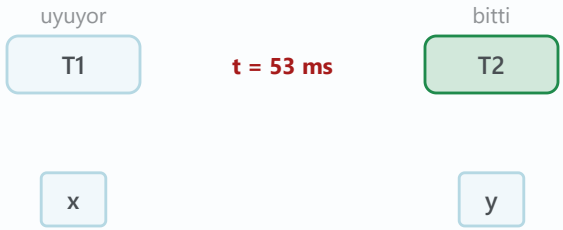
`x.bakiye = 1001` `y.bakiye = 999`

ÇIKTI

```
T1: false
```

tryLock ile Geri Alma: Kod İzleme

```
1  boolean dene(Hesap k, Hesap h, int m)
2          throws InterruptedException {
3    if (!k.kilit.tryLock(50, MILLISECONDS))
4      return false;
5    try {
6      if (!h.kilit.tryLock(50, MILLISECONDS))
7        return false;  /* k'yi de bırak */
8      try { k.bakiye -= m; h.bakiye += m;
9            return true; }
▶      finally { h.kilit.unlock(); }
11   } finally { k.kilit.unlock(); }
12 }
13 /* çağıran: while (!dene(x, y, 1))
14               Thread.sleep(rastgele); */
```



T2 önce `x`'i, sonra `y`'yi bırakır ve true döner.

Adım 9 / 10

DEĞİŞKENLER

`t = 53 ms` `x = boş` `y = boş`

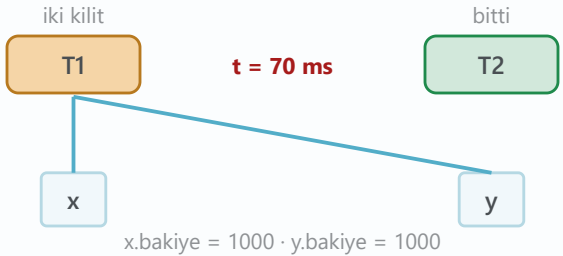
`x.bakiye = 1001` `y.bakiye = 999`

ÇIKTI

```
T1: false
T2: true
```

tryLock ile Geri Alma: Kod İzleme

```
1  boolean dene(Hesap k, Hesap h, int m)
2          throws InterruptedException {
3    if (!k.kilit.tryLock(50, MILLISECONDS))
4      return false;
5    try {
6      if (!h.kilit.tryLock(50, MILLISECONDS))
7        return false;  /* k'yi de bırak */
8      try { k.bakiye -= m; h.bakiye += m;
9            return true; }
10     finally { h.kilit.unlock(); }
11   } finally { k.kilit.unlock(); }
12 }
▶ /* çağıran: while (!dene(x, y, 1))
▶                Thread.sleep(rastgele); */
```



T1 rastgele uyuduktan sonra yeniden dener; iki kilidi de alır ve aktarır: toplam yine 2000.

Adım 10 / 10

DEĞİŞKENLER

`t = 70 ms` `x = T1` `y = T1`

`x.bakiye = 1000` `y.bakiye = 1000`

ÇIKTI

```
T1: false
T2: true
T1: true
```

ÖNLEME

Döngüsel Beklemeye Saldırı

#	Kaynak
1	Görüntüleyici
2	Yazıcı
3	Çizici
4	Teyp sürücüsü
5	Blu-ray sürücüsü

Yazıcı ve teyp gerekiyorsa önce 2, sonra 4 istenir. Uygun bir sıralamayı bulmak en zor kısımdır.



Her kaynak türüne bir numara verilir. Kural: süreç yalnızca **elindekilerden büyük** numaralı kaynak isteyebilir.

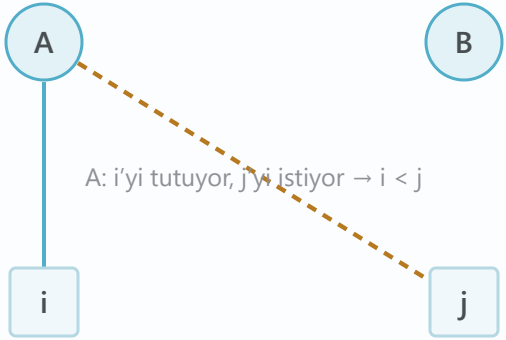
Adım 1 / 3

Tanenbaum MOS Şekil 6-13. Varyant: süreç K_j'yi isteyecekse elindeki tüm K_i ($i \geq j$) kaynaklarını önce bırakmalıdır.

Döngüsel Beklemeye Saldırı

#	Kaynak
1	Görüntüleyici
2	Yazıcı
3	Çizici
4	Teyp sürücüsü
5	Blu-ray sürücüsü

Yazıcı ve teyp gerekiyorsa önce 2, sonra 4 istenir. Uygun bir sıralamayı bulmak en zor kısımdır.



A, i'yi tutup j'yi istiyorsa kurala göre i < j.

Adım 2 / 3

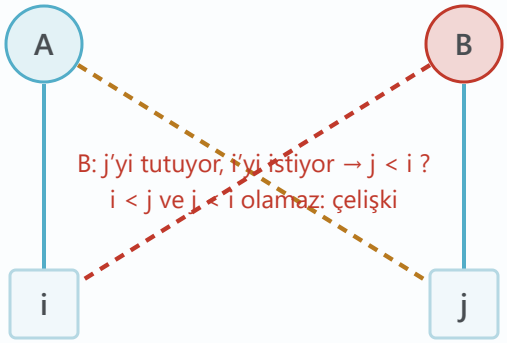
Tanenbaum MOS Şekil 6-13. Varyant: süreç K_j'yi isteyecekse elindeki tüm K_i ($i \geq j$) kaynaklarını önce bırakmalıdır.

ÖNLEME

Döngüsel Beklemeye Saldırı

#	Kaynak
1	Görüntüleyici
2	Yazıcı
3	Çizici
4	Teyp sürücüsü
5	Blu-ray sürücüsü

Yazıcı ve teyp gerekiyorsa önce 2, sonra 4 istenir. Uygun bir sıralamayı bulmak en zor kısımdır.



B, j'yi tutup i'yi isterse j < i olmalı: çelişki. Bu kenar **hiç oluşamaz**. Genel olarak döngüdeki en büyük numaralı kaynağı tutan süreç daha küçük numaralı bir kaynağı isteyemez.

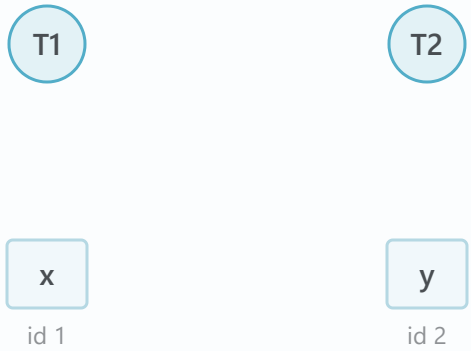
Adım 3 / 3

Tanenbaum MOS Şekil 6-13. Varyant: süreç K_j'yi isteyecekse elindeki tüm K_i ($i \geq j$) kaynaklarını önce bırakmalıdır.

ÖNLEME · KODU ADIM ADIM ÇALIŞTIR

Kilit Sıralaması Pratikte

```
▶ void transfer(Hesap k, Hesap h, int m) {
2   Hesap ilk = k.id < h.id ? k : h;
3   Hesap ikinci = k.id < h.id ? h : k;
4   ilk.kilit.lock();
5   try {
6     ikinci.kilit.lock();
7     try { k.bakiye -= m; h.bakiye += m; }
8     finally { ikinci.kilit.unlock(); }
9   } finally { ilk.kilit.unlock(); }
10 }
11 /* T1: transfer(x, y, 1)   x.id = 1
12    T2: transfer(y, x, 1)   y.id = 2 */
```



T1 x→y, T2 y→x aktarıyor: “doğal” sıra iki iş parçacığında terstir. Kilitlerin `id` sırası ise ortaktır.

Adım 1 / 6

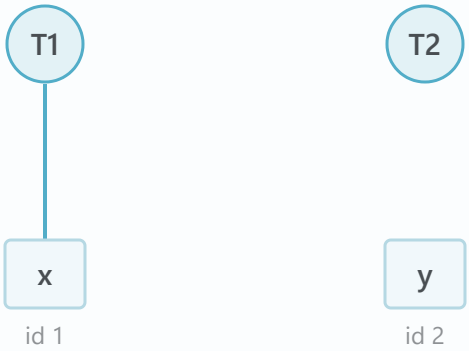
DEĞİŞKENLER

`T1.ilk = ?` `T2.ilk = ?`

Linux çekirdeği aynı türden iki kilidi adres sırasıyla alır (ör. `double_rq_lock`); kilit sırası belgelenir ve `lockdep` ile denetlenir.

Kilit Sıralaması Pratikte

```
1  void transfer(Hesap k, Hesap h, int m) {
2    Hesap ilk = k.id < h.id ? k : h;
3    Hesap ikinci = k.id < h.id ? h : k;
▶    ilk.kilit.lock();
5    try {
6      ikinci.kilit.lock();
7      try { k.bakiye -= m; h.bakiye += m; }
8      finally { ikinci.kilit.unlock(); }
9    } finally { ilk.kilit.unlock(); }
10 }
11 /* T1: transfer(x, y, 1)   x.id = 1
12    T2: transfer(y, x, 1)   y.id = 2 */
```



T1: ilk = x (1 < 2). x kilidini alır.

Adım 2 / 6

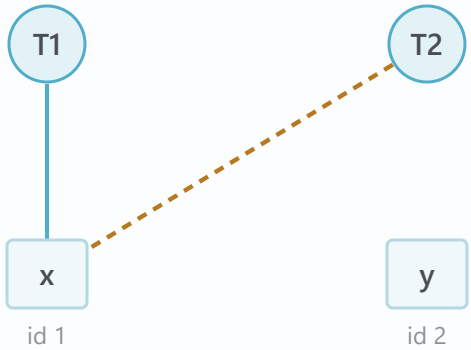
DEĞİŞKENLER

`T1.ilk = x` `T2.ilk = ?`

Linux çekirdeği aynı türden iki kilidi adres sırasıyla alır (ör. `double_rq_lock`); kilit sırası belgelenir ve `lockdep` ile denetlenir.

Kilit Sıralaması Pratikte

```
1  void transfer(Hesap k, Hesap h, int m) {
2    Hesap ilk = k.id < h.id ? k : h;
3    Hesap ikinci = k.id < h.id ? h : k;
▶    ilk.kilit.lock();
5    try {
6      ikinci.kilit.lock();
7      try { k.bakiye -= m; h.bakiye += m; }
8      finally { ikinci.kilit.unlock(); }
9    } finally { ilk.kilit.unlock(); }
10 }
11 /* T1: transfer(x, y, 1)   x.id = 1
12    T2: transfer(y, x, 1)   y.id = 2 */
```



T2: k = y ama ilk yine x (id 1). x'i bekler; **elinde hiçbir kilit yok**.

Adım 3 / 6

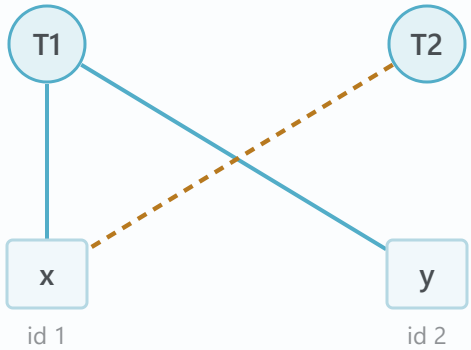
DEĞİŞKENLER

`T1.ilk = x` `T2.ilk = x`

Linux çekirdeği aynı türden iki kilidi adres sırasıyla alır (ör. `double_rq_lock`); kilit sırası belgelenir ve `lockdep` ile denetlenir.

Kilit Sıralaması Pratikte

```
1  void transfer(Hesap k, Hesap h, int m) {
2    Hesap ilk = k.id < h.id ? k : h;
3    Hesap ikinci = k.id < h.id ? h : k;
4    ilk.kilit.lock();
5    try {
▶      ikinci.kilit.lock();
▶      try { k.bakiye -= m; h.bakiye += m; }
8      finally { ikinci.kilit.unlock(); }
9    } finally { ilk.kilit.unlock(); }
10 }
11 /* T1: transfer(x, y, 1)   x.id = 1
12    T2: transfer(y, x, 1)   y.id = 2 */
```



T1 y'yi de alır (kimse tutmuyor) ve aktarımı yapar.

Adım 4 / 6

DEĞİŞKENLER

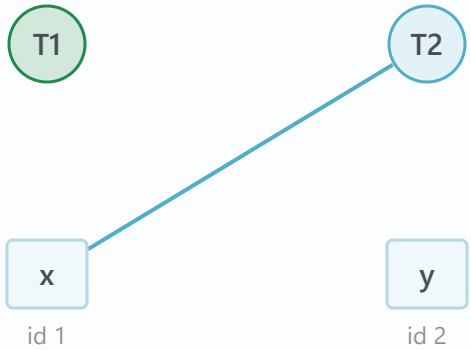
`T1.ilk = x` `T2.ilk = x`

Linux çekirdeği aynı türden iki kilidi adres sırasıyla alır (ör. `double_rq_lock`); kilit sırası belgelenir ve `lockdep` ile denetlenir.

ÖNLEME · KODU ADIM ADIM ÇALIŞTIR

Kilit Sıralaması Pratikte

```
1  void transfer(Hesap k, Hesap h, int m) {
2    Hesap ilk = k.id < h.id ? k : h;
3    Hesap ikinci = k.id < h.id ? h : k;
4    ilk.kilit.lock();
5    try {
6      ikinci.kilit.lock();
7      try { k.bakiye -= m; h.bakiye += m; }
▶      finally { ikinci.kilit.unlock(); }
▶    } finally { ilk.kilit.unlock(); }
10 }
11 /* T1: transfer(x, y, 1)   x.id = 1
12    T2: transfer(y, x, 1)   y.id = 2 */
```



T1 kilitleri bırakır; T2 x'i, sonra y'yi alır. Döngü oluşamaz.

Adım 5 / 6

DEĞİŞKENLER

`T1 = bitti` `T2.ilk = x`

Linux çekirdeği aynı türden iki kilidi adres sırasıyla alır (ör. `double_rq_lock`); kilit sırası belgelenir ve `lockdep` ile denetlenir.

ÖNLEME · KODU ADIM ADIM ÇALIŞTIR

Kilit Sıralaması Pratikte

```
1  void transfer(Hesap k, Hesap h, int m) {
2    Hesap ilk = k.id < h.id ? k : h;
3    Hesap ikinci = k.id < h.id ? h : k;
4    ilk.kilit.lock();
5    try {
6      ikinci.kilit.lock();
7      try { k.bakiye -= m; h.bakiye += m; }
8      finally { ikinci.kilit.unlock(); }
9    } finally { ilk.kilit.unlock(); }
▶  }
11 /* T1: transfer(x, y, 1)   x.id = 1
12    T2: transfer(y, x, 1)   y.id = 2 */
```



Çalıştırıldı (JDK 21): her biri 100.000 ters yönlü aktarım, sonuç `x=1000 y=1000`, kilitlenme yok.

Adım 6 / 6

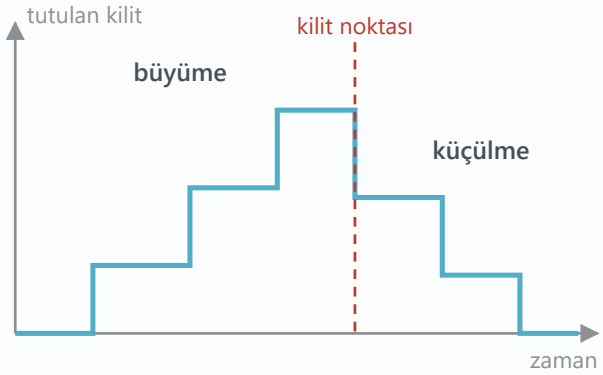
DEĞİŞKENLER

`T1 = bitti` `T2 = bitti`

Linux çekirdeği aynı türden iki kilidi adres sırasıyla alır (ör. `double_rq_lock`); kilit sırası belgelenir ve `lockdep` ile denetlenir.

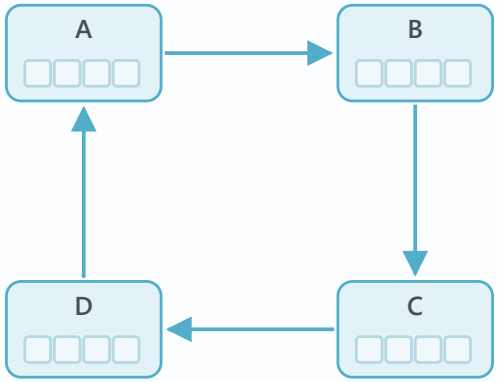
İki Aşamalı Kilitleme (2PL)

- Veritabanı işlemleri için protokol: **büyüme** evresinde yalnızca kilit alınır, **küçülme** evresinde yalnızca bırakılır; ilk bırakıştan sonra yeni kilit alınamaz.
- Amacı **serileştirilebilirlik**tir (işlemler sanki sırayla çalışmış gibi), kilitlenmeyi önlemek değil. **Düz 2PL kilitlenebilir:** T1 a'yı, T2 b'yi kilitler; sonra çapraz isterler.
- Tanenbaum'un varyantı: 1. evrede tüm kayıtları kilitlemeyi dene; biri kilitliyse **hepsini bırak** ve baştan başla. Bu, tut ve beklemeyi kaldırır.
- Katı (strict) 2PL: kilitler işlem sonunda (commit/abort) bırakılır. Kilitlenmeleri DBMS bekleme çizgesiyle tespit edip bir işlemi geri sarar.



İletişim Kilitlenmeleri

- **İletişim kilitlenmesi:** kaynak yoktur, süreçler birbirinden *ileti* bekler. A, B'ye istek gönderip yanıt bekler; istek kaybolursa A yanıtı, B isteği sonsuza dek bekler.
- Kaynak sıralaması ya da kaçınma işe yaramaz; çözüm **zaman aşımı** (timeout) ve yeniden gönderme. Yinelenen iletiler için protokolde sıra numarası gerekir.
- Ağ donanımında ise tamponlar gerçek kaynaktır; aşağıdaki halka bir **kaynak kilitlenmesidir**.

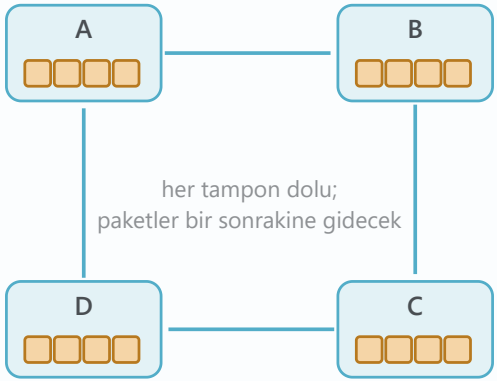


Dört yönlendirici halka biçiminde; her birinin sınırlı tampon belleği var.

Tanenbaum MOS §6.7.2, Şekil 6-15.

İletişim Kilitlenmeleri

- **İletişim kilitlenmesi:** kaynak yoktur, süreçler birbirinden *ileti* bekler. A, B'ye istek gönderip yanıt bekler; istek kaybolursa A yanıtı, B isteği sonsuza dek bekler.
- Kaynak sıralaması ya da kaçınma işe yaramaz; çözüm **zaman aşımı** (timeout) ve yeniden gönderme. Yinelenen iletiler için protokolde sıra numarası gerekir.
- Ağ donanımında ise tamponlar gerçek kaynaktır; aşağıdaki halka bir **kaynak kilitlenmesidir**.



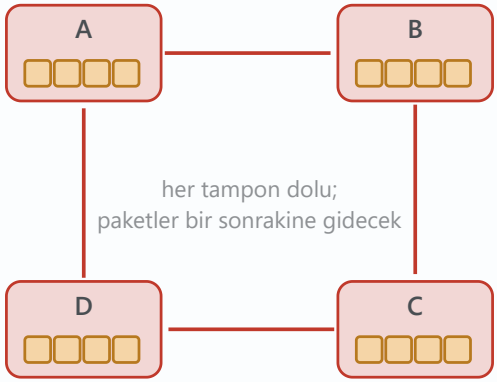
Yoğun trafikte tüm tamponlar, saat yönünde bir sonraki yönlendiriciye gidecek paketlerle dolar.

Adım 2 / 3

Tanenbaum MOS §6.7.2, Şekil 6-15.

İletişim Kilitlenmeleri

- **İletişim kilitlenmesi:** kaynak yoktur, süreçler birbirinden *ileti* bekler. A, B'ye istek gönderip yanıt bekler; istek kaybolursa A yanıtı, B isteği sonsuza dek bekler.
- Kaynak sıralaması ya da kaçınma işe yaramaz; çözüm **zaman aşımı** (timeout) ve yeniden gönderme. Yinelenen iletiler için protokolde sıra numarası gerekir.
- Ağ donanımında ise tamponlar gerçek kaynaktır; aşağıdaki halka bir **kaynak kilitlenmesidir**.



kimse gönderemez: kaynak kilitlenmesi

A, B'nin boş tamponunu bekler; B, C'nin; C, D'nin; D, A'nın: döngüsel bekleme. Çözüm: zaman aşımıyla paket düşürme, sanal kanallar, kredi tabanlı akış denetimi.

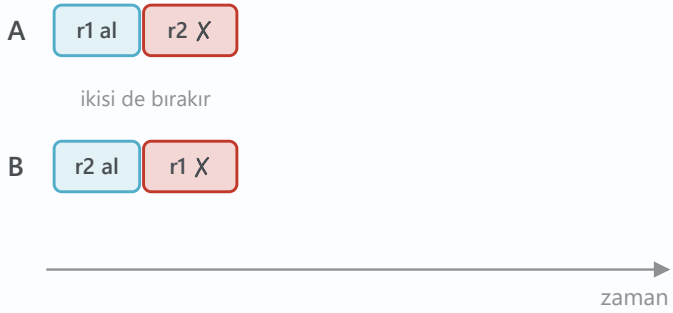
Tanenbaum MOS §6.7.2, Şekil 6-15.

Canlı Kilit (Livelock)

- Koridorda karşılaşan iki kişi aynı anda aynı yana kayar: kimse geçemez.
- Kilitlenmede süreçler **uyur**; canlı kilitte **çalışır** ama faydalı iş yapmaz.

```
void process_A(void) {
  enter_region(&resource_1);
  enter_region(&resource_2);
  use_both_resources();
  leave_region(&resource_2);
  leave_region(&resource_1);
}  /* B: önce resource_2, sonra 1 */
```

Tanenbaum Şekil 6-16: meşgul beklemeli `enter_region` ters sırada çağrılırsa süreçler bloke olmadan sonsuza dek döner.



A r1'i, B r2'yi alır. A r2'yi dener (`tryLock` başarısız), B r1'i dener (başarısız).

Adım 1 / 4

Canlı Kilit (Livelock)

- Koridorda karşılaşan iki kişi aynı anda aynı yana kayar: kimse geçemez.
- Kilitlenmede süreçler **uyur**; canlı kilitte **çalışır** ama faydalı iş yapmaz.

```
void process_A(void) {
  enter_region(&resource_1);
  enter_region(&resource_2);
  use_both_resources();
  leave_region(&resource_2);
  leave_region(&resource_1);
}  /* B: önce resource_2, sonra 1 */
```

Tanenbaum Şekil 6-16: meşgul beklemeli `enter_region` ters sırada çağrılırsa süreçler bloke olmadan sonsuza dek döner.



Kibar davranırlar: ikisi de elindekini bırakır ve *aynı süre* sonra yeniden dener. Aynı şey tekrarlanır.

Canlı Kilit (Livelock)

- Koridorda karşılaşan iki kişi aynı anda aynı yana kayar: kimse geçemez.
- Kilitlenmede süreçler **uyur**; canlı kilitte **çalışır** ama faydalı iş yapmaz.

```
void process_A(void) {
  enter_region(&resource_1);
  enter_region(&resource_2);
  use_both_resources();
  leave_region(&resource_2);
  leave_region(&resource_1);
}  /* B: önce resource_2, sonra 1 */
```

Tanenbaum Şekil 6-16: meşgul beklemeli `enter_region` ters sırada çağrılırsa süreçler bloke olmadan sonsuza dek döner.



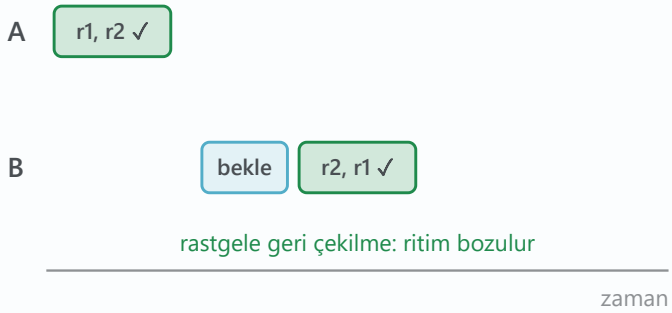
Süreçler **bloke değil**, CPU harcıyor, durumları sürekli değişiyor; ama hiçbiri ilerlemiyor: **canlı kilit**.

Canlı Kilit (Livelock)

- Koridorda karşılaşan iki kişi aynı anda aynı yana kayar: kimse geçemez.
- Kilitlenmede süreçler **uyur**; canlı kilitte **çalışır** ama faydalı iş yapmaz.

```
void process_A(void) {
  enter_region(&resource_1);
  enter_region(&resource_2);
  use_both_resources();
  leave_region(&resource_2);
  leave_region(&resource_1);
}  /* B: önce resource_2, sonra 1 */
```

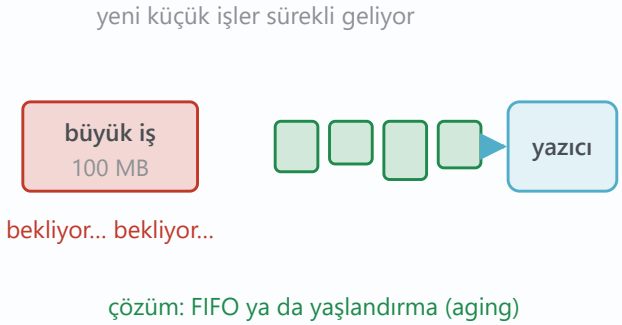
Tanenbaum Şekil 6-16: meşgul beklemeli `enter_region` ters sırada çağrılırsa süreçler bloke olmadan sonsuza dek döner.



Çözüm: rastgele (üstel) geri çekilme, öncelik/sıra numarası ya da kilit sıralaması. Ethernet'in CSMA/CD geri çekilmesi aynı fikirdir.

Açlık (Starvation)

- Bir süreç, kaynak kullanılabilir olduğu hâlde ona **hiç sıra gelmediği** için süresiz bekler. Diğerleri ilerler: sistem kilitlenmemiştir.
- Kaynak dağıtım **politikasının** sonucudur. Örnek: yazıcıyı her seferinde en küçük dosyaya vermek ortalama bekleme süresini azaltır ama büyük dosyayı aç bırakabilir.
- Başka örnekler: katı öncelikli çizelgeleme, okur öncelikli okur-yazar kilidi, kurtarmada hep aynı kurbanın seçilmesi.
- Çözüm: ilk gelen ilk hizmet alır (FIFO) ya da bekledikçe önceliği artırma (**yaşlandırma**).



	Kilitlenme	Canlı kilit	Açlık
Süreç durumu	bloke	çalışıyor	hazır/bloke
Kim etkilenir?	döngüdeki herkes	etkileşen grup	bir ya da birkaç süreç

DİĞER SORUNLAR

Kilitlenme Dışı Hatalar: Atomiklik ve Sıra

En yaygın eş zamanlılık hatası kilitlenme değildir: MySQL, Apache, Mozilla ve OpenOffice'te incelenen 105 hatanın 74'ü kilitlenme dışıdır ve neredeyse hepsi iki kalıba uyar (Lu ve ark., 2008).

Atomiklik ihlali

Birlikte atomik olması gereken iki erişimin arasına başka bir iş parçacığı girer.

```
/* T1 */
if (oturum->ad != NULL)
    puts(oturum->ad);  /* artık NULL? */
/* T2 */
oturum->ad = NULL;
```

Çare: sınama ile kullanımı ve T2'nin yazmasını *aynı* mutex altına almak.

Sıra ihlali

"A, B'den önce olur" diye varsayılır ama bu sıra hiçbir şeyle zorlanmaz.

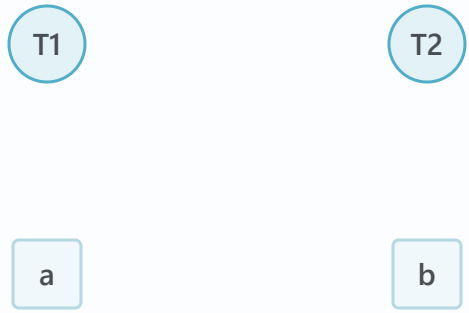
```
/* T1: baslat() */
isci = thread_olustur(govde);
/* T2: govde(), yeni iş parçacığı */
d = isci->durum;  /* isci atandı mı? */
```

Çare: `hazir` bayrağı ve koşul değişkeni. T2 `while (!hazir) wait`, T1 atamadan sonra `hazir = 1; signal` yapar (ya da 0'dan başlayan semafor).

Yanılgı: "her erişim kilitliyse kod doğrudur." Erişimler *tek tek* kilitli olsa da aralarında kilit bırakılıyorsa atomiklik ihlali sürer; ThreadSanitizer gibi veri yarışı araçları bunu görmeyebilir.

Java'da Kilitlenme

```
1  Object a = new Object(), b = new Object();
2
3  Thread t1 = new Thread(() -> {
4    synchronized (a) {
5      uyu(10);   /* pencereyi büyüt */
6      synchronized (b) { isle(); }
7    }
8  }, "T1");
9  Thread t2 = new Thread(() -> {
10   synchronized (b) {
11     uyu(10);
12     synchronized (a) { isle(); }
13   }
14 }, "T2");
▶  t1.start(); t2.start();
```



İki iş parçacığı iki monitörü (`synchronized` kilidi) **ters sırada** alacak.

Adım 1 / 5

DEĞİŞKENLER

`T1 = RUNNABLE` `T2 = RUNNABLE`

`uyu(10)` zamanlamayı çok olası kılar ama garanti etmez. `CountDownLatch` ile iki ilk kilit alınana dek bekletilen sürüm JDK 21'de her çalıştırmada kilitlendi.

Java’da Kilitlenme

```
1  Object a = new Object(), b = new Object();
2
3  Thread t1 = new Thread(() -> {
▶    synchronized (a) {
▶      uyu(10);   /* pencereyi büyüt */
6      synchronized (b) { isle(); }
7    }
8  }, "T1");
9  Thread t2 = new Thread(() -> {
10   synchronized (b) {
11     uyu(10);
12     synchronized (a) { isle(); }
13   }
14 }, "T2");
15 t1.start(); t2.start();
```



T1 a’nın monitörüne girer ve uyur.

Adım 2 / 5

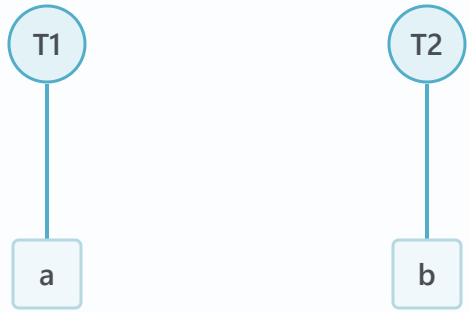
DEĞİŞKENLER

`T1 = TIMED_WAITING` `T2 = RUNNABLE`

`uyu(10)` zamanlamayı çok olası kılar ama garanti etmez. `CountDownLatch` ile iki ilk kilit alınana dek bekletilen sürüm JDK 21’de her çalıştırmada kilitlendi.

Java'da Kilitlenme

```
1  Object a = new Object(), b = new Object();
2
3  Thread t1 = new Thread(() -> {
4    synchronized (a) {
5      uyu(10);   /* pencereyi büyüt */
6      synchronized (b) { isle(); }
7    }
8  }, "T1");
9  Thread t2 = new Thread(() -> {
▶    synchronized (b) {
▶      uyu(10);
12     synchronized (a) { isle(); }
13   }
14 }, "T2");
15 t1.start(); t2.start();
```



Bu sırada T2 b'nin monitörüne girer.

Adım 3 / 5

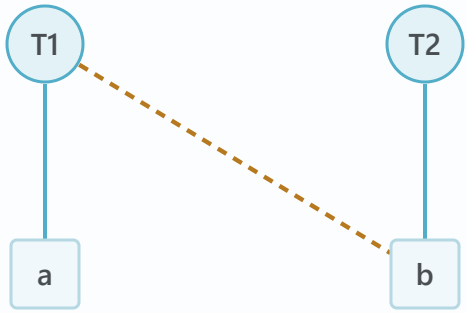
DEĞİŞKENLER

`T1 = TIMED_WAITING` `T2 = TIMED_WAITING`

`uyu(10)` zamanlamayı çok olası kılar ama garanti etmez. `CountDownLatch` ile iki ilk kilit alınana dek bekletilen sürüm JDK 21'de her çalıştırmada kilitlendi.

Java'da Kilitlenme

```
1  Object a = new Object(), b = new Object();
2
3  Thread t1 = new Thread(() -> {
4    synchronized (a) {
5      uyu(10);   /* pencereyi büyüt */
6      synchronized (b) { isle(); }
7    }
8  }, "T1");
9  Thread t2 = new Thread(() -> {
10   synchronized (b) {
11     uyu(10);
12     synchronized (a) { isle(); }
13   }
14 }, "T2");
15 t1.start(); t2.start();
```



T1 uyanır, b'yi ister: b T2'de → T1 `BLOCKED`.

Adım 4 / 5

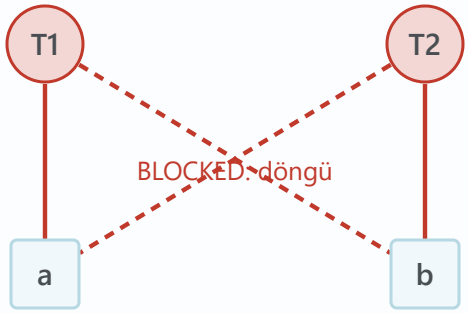
DEĞİŞKENLER

`T1 = BLOCKED` `T2 = TIMED_WAITING`

`uyu(10)` zamanlamayı çok olası kılar ama garanti etmez. `CountDownLatch` ile iki ilk kilit alınana dek bekletilen sürüm JDK 21'de her çalıştırmada kilitlendi.

Java'da Kilitlenme

```
1  Object a = new Object(), b = new Object();
2
3  Thread t1 = new Thread(() -> {
4    synchronized (a) {
5      uyu(10);   /* pencereyi büyüt */
▶      synchronized (b) { isle(); }
7    }
8  }, "T1");
9  Thread t2 = new Thread(() -> {
10   synchronized (b) {
11     uyu(10);
▶      synchronized (a) { isle(); }
13   }
14 }, "T2");
15 t1.start(); t2.start();
```



T2 a'yı ister → `BLOCKED`. JVM monitörleri kesilemez: program asla bitmez.

Adım 5 / 5

DEĞİŞKENLER

`T1 = BLOCKED` `T2 = BLOCKED`

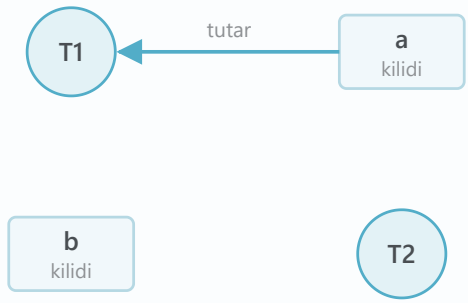
`uyu(10)` zamanlamayı çok olası kılar ama garanti etmez. `CountDownLatch` ile iki ilk kilit alınana dek bekletilen sürüm JDK 21'de her çalıştırmada kilitlendi.

Kapsülleme Kilit Sırasını Gizler

```
var a = Collections.synchronizedList(
          new ArrayList<Integer>());
var b = Collections.synchronizedList(
          new ArrayList<Integer>());
/* a.addAll(b) içeride:              */
/* synchronized (a) { b.toArray() }  */
/*   → synchronized (b) { ... }      */
new Thread(() -> a.addAll(b)).start();
new Thread(() -> b.addAll(a)).start();

/* Düzeltme: kopyayı kilit dışında al */
Integer[] k = b.toArray(new Integer[0]);
a.addAll(Arrays.asList(k));
```

Çapraz çağrı	JDK 8	JDK 21
`Vector` : `a.addAll(b)` ‖ `b.addAll(a)`	10/10	0/10
`synchronizedList` : aynı	10/10	10/10
Düzeltilmiş sürüm	0/10	0/10



T1: `a.addAll(b)` sarmalayıcıda önce **a**'nın kilidini alır.

Adım 1 / 3

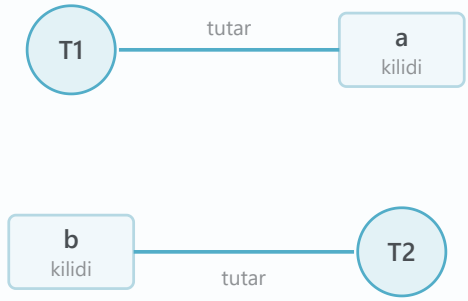
Tablo: 2 iş parçacığı × 200.000 tur, 10 denemede kilitlenen; sonuç çalıştırmaya göre değişebilir (ör. 9/10). JDK 21'de `Vector.addAll` argümanın `toArray()`'ini kilitsiz çağırır. Kural: kilit tutulurken yabancı yöntem çağrılmaz.

Kapsülleme Kilit Sırasını Gizler

```
var a = Collections.synchronizedList(
          new ArrayList<Integer>());
var b = Collections.synchronizedList(
          new ArrayList<Integer>());
/* a.addAll(b) içeride:              */
/* synchronized (a) { b.toArray() }  */
/*   → synchronized (b) { ... }      */
new Thread(() -> a.addAll(b)).start();
new Thread(() -> b.addAll(a)).start();

/* Düzeltme: kopyayı kilit dışında al */
Integer[] k = b.toArray(new Integer[0]);
a.addAll(Arrays.asList(k));
```

Çapraz çağrı	JDK 8	JDK 21
`Vector` : `a.addAll(b)` ‖ `b.addAll(a)`	10/10	0/10
`synchronizedList` : aynı	10/10	10/10
Düzeltilmiş sürüm	0/10	0/10



Aynı anda T2: `b.addAll(a)` önce **b**'nin kilidini alır.

Adım 2 / 3

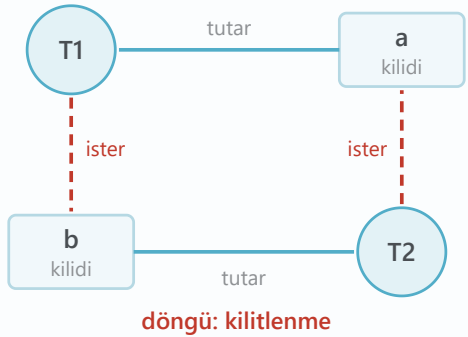
Tablo: 2 iş parçacığı × 200.000 tur, 10 denemede kilitlenen; sonuç çalıştırmaya göre değişebilir (ör. 9/10). JDK 21'de `Vector.addAll` argümanın `toArray()`'ini kilitsiz çağırır. Kural: kilit tutulurken yabancı yöntem çağrılmaz.

Kapsülleme Kilit Sırasını Gizler

```
var a = Collections.synchronizedList(
          new ArrayList<Integer>());
var b = Collections.synchronizedList(
          new ArrayList<Integer>());
/* a.addAll(b) içeride:              */
/* synchronized (a) { b.toArray() }  */
/*   → synchronized (b) { ... }      */
new Thread(() -> a.addAll(b)).start();
new Thread(() -> b.addAll(a)).start();

/* Düzeltme: kopyayı kilit dışında al */
Integer[] k = b.toArray(new Integer[0]);
a.addAll(Arrays.asList(k));
```

Çapraz çağrı	JDK 8	JDK 21
`Vector` : `a.addAll(b)` ‖ `b.addAll(a)`	10/10	0/10
`synchronizedList` : aynı	10/10	10/10
Düzeltilmiş sürüm	0/10	0/10



İçeride `b.toArray()` ve `a.toArray()` çağrılır: her biri ötekinin kilidini ister.

Tablo: 2 iş parçacığı × 200.000 tur, 10 denemede kilitlenen; sonuç çalıştırmaya göre değişebilir (ör. 9/10). JDK 21'de `Vector.addAll` argümanın `toArray()`'ini kilitsiz çağırır. Kural: kilit tutulurken yabancı yöntem çağrılmaz.

Kilitlenmeyi Bulmak: jstack

```
$ jstack <pid>
Found one Java-level deadlock:
=============================
"T1":
  waiting to lock monitor 0x… (object 0x…, a java.lang.Object),
  which is held by "T2"
"T2":
  waiting to lock monitor 0x… (object 0x…, a java.lang.Object),
  which is held by "T1"

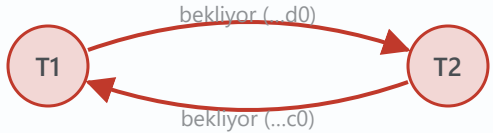
Java stack information for the threads listed above:
"T1":
        at Asili.calis(Asili.java:7)
        - waiting to lock <0x…d0> (a java.lang.Object)
        - locked <0x…c0> (a java.lang.Object)
…
Found 1 deadlock.
```

JDK 21'de gerçek çıktı (adresler kısaltıldı). Aynı rapor: `jcmd <pid> Thread.print`, JConsole/VisualVM "Detect Deadlock".

```
ThreadMXBean mx =
    ManagementFactory.getThreadMXBean();
long[] ids = mx.findDeadlockedThreads();
if (ids != null)
  for (ThreadInfo ti : mx.getThreadInfo(ids))
    System.out.println(ti.getThreadName()
      + " bekliyor, kilit sahibi "
      + ti.getLockOwnerName());
```

```
T1 bekliyor, kilit sahibi T2
T2 bekliyor, kilit sahibi T1
```

- JVM, monitör ve `ReentrantLock` sahipliğinden **bekleme çizgesi** kurup döngü arar.
- Tespit eder ama **kurtarmaz**: süreci yeniden başlatmak gerekir.



Araçlar: lockdep ve Diğerleri

Ortam	Araç / davranış
Linux çekirdeği	`lockdep` (`CONFIG_PROVE_LOCKING`): kilit sırası doğrulayıcı; hung task dedektörü
C/C++	ThreadSanitizer: *lock-order-inversion*; Valgrind Helgrind
Java	`jstack` , `jcmd` , `ThreadMXBean`
Go	`fatal error: all goroutines are asleep - deadlock!`
Veritabanı	PostgreSQL, InnoDB: bekleme çizgesi + kurban işlemi geri sarma



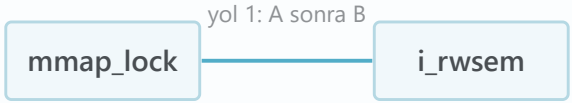
lockdep her kilidi bir **sınıfa** (aynı türdeki tüm örnekler) bağlar.

Adım 1 / 3

lockdep ve TSan *potansiyel* kilitlenmeyi bulur (sıra ihlali); jstack ve Go çalışma zamanı ise *gerçekleşmiş* kilitlenmeyi raporlar.

Araçlar: lockdep ve Diğerleri

Ortam	Araç / davranış
Linux çekirdeği	`lockdep` (`CONFIG_PROVE_LOCKING`): kilit sırası doğrulayıcı; hung task dedektörü
C/C++	ThreadSanitizer: *lock-order-inversion*; Valgrind Helgrind
Java	`jstack` , `jcmd` , `ThreadMXBean`
Go	`fatal error: all goroutines are asleep - deadlock!`
Veritabanı	PostgreSQL, InnoDB: bekleme çizgesi + kurban işlemi geri sarma

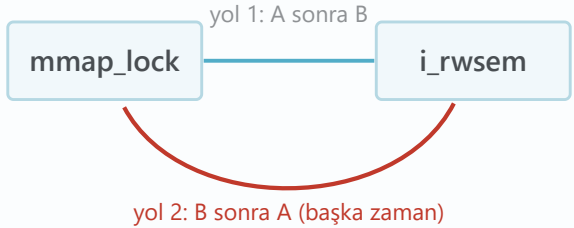


Bir görev A'yı tutarken B'yi alınca "A → B" kenarı kaydedilir.

lockdep ve TSan *potansiyel* kilitlenmeyi bulur (sıra ihlali); jstack ve Go çalışma zamanı ise *gerçekleşmiş* kilitlenmeyi raporlar.

Araçlar: lockdep ve Diğerleri

Ortam	Araç / davranış
Linux çekirdeği	`lockdep` (`CONFIG_PROVE_LOCKING`): kilit sırası doğrulayıcı; hung task dedektörü
C/C++	ThreadSanitizer: *lock-order-inversion*; Valgrind Helgrind
Java	`jstack` , `jcmd` , `ThreadMXBean`
Go	`fatal error: all goroutines are asleep - deadlock!`
Veritabanı	PostgreSQL, InnoDB: bekleme çizgesi + kurban işlemi geri sarma



`possible circular locking dependency`

Başka bir yolda B tutulurken A alınırsa çizgede döngü oluşur: kilitlenme **hiç yaşanmadan** rapor edilir.

lockdep ve TSan *potansiyel* kilitlenmeyi bulur (sıra ihlali); jstack ve Go çalışma zamanı ise *gerçekleşmiş* kilitlenmeyi raporlar.

Özet

Yaklaşım	Ne zaman?	Bilgi gereksinimi	Bedel	Kullanım
Görmezden gel	—	yok	ara sıra yeniden başlatma	genel amaçlı İS
Tespit + kurtarma	sonra	şu anki tahsis ve istekler	periyodik $O(m \cdot n^2)$, kayıp iş	veritabanları, JVM araçları
Kaçınma	her istekte	azami ihtiyaçlar (Max)	ertelemeler, $O(m \cdot n^2)$/istek	nadiren, özel sistemler
Önleme	tasarımda	kaynak sırası	esneklik ve kullanım kaybı	çekirdek, uygulama kodu

- Dört koşul birlikte gereklidir; tek örnekli kaynaklarda döngü ⇔ kilitlenme.
- Güvensiz durum kilitlenme değildir ama bitiş garantisi yoktur.
- Pratikte en etkili araç: **kilit sıralaması** + zaman aşımlı deneme + araçlarla (lockdep, jstack, TSan) denetim.

ALIŞTIRMALAR

Alıştırma: Hatayı Sınıflandır

Her durumu sınıflandırın: **kilitlenme**, **canlı kilit**, **atomiklik ihlali** ya da **sıra ihlali**. Çareyi de söyleyin.

1. İş parçacığı güvenli bir kuyrukta iki tüketici: `if (!q.isEmpty()) x = q.remove();`
2. Ana iş parçacığı önce işçiyi başlatır, sonra `tampon = malloc(N)` yapar; işçi ilk satırda `tampon[0]`'a yazar.
3. `transfer(x, y)` ile `transfer(y, x)` aynı anda; her biri önce kaynak, sonra hedef hesabı kilitler.
4. İki iş parçacığı `lock(A); if (!trylock(B)) { unlock(A); usleep(10); yeniden dene; }` yapar; öteki önce B'yi alır. Bekleme hep 10 µs.
5. 2 CPU ve 4 iş parçacığı: T1 ve T2 hem L1 hem L2'yi, T3 yalnız L2'yi kullanır, T4 kilit kullanmaz. Kilitleri bilen bir çizelgeleyici kilitlenmeyi nasıl önler?

▶ **Cevaplar**

ÖZET

Kontrol Soruları

1. Kilitlenmenin dört koşulunu yazın. Hangisi bir politika değil, durumdur?
2. Çizgede döngü varken kilitlenme olmaması nasıl mümkündür? Örnek verin.
3. E = (4 2 3 1), A = (2 1 0 0) örneğinde P3'ün isteği (2 1 0 1) olsaydı sonuç ne olurdu?
4. Güvensiz durum ile kilitlenmiş durum arasındaki fark nedir?
5. Silberschatz örneğinde T1'in (1 0 2) isteğinden sonra T0 (0 2 0) isterse ne olur? Neden?
6. Bankacı algoritması genel amaçlı İS'lerde neden kullanılmaz?
7. Kaynak numaralandırma döngüsel beklemeyi neden imkânsız kılar?
8. Canlı kilit ile kilitlenme arasındaki fark nedir? Canlı kilit nasıl kırılır?

▶ **Cevaplar**

ÖZET

Kaynaklar

- A. Silberschatz, P. B. Galvin, G. Gagne, *Operating System Concepts*, 10. baskı, Wiley, 2018 — Bölüm 8: Deadlocks.
- A. S. Tanenbaum, H. Bos, *Modern Operating Systems*, 4. baskı (2014) / 5. baskı (2023), Pearson — Bölüm 6: Deadlocks.
- R. H. Arpaci-Dusseau, A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces* (OSTEP), ostep.org — "Common Concurrency Problems".
- E. G. Coffman, M. J. Elphick, A. Shoshani, "System Deadlocks", *ACM Computing Surveys* 3(2), 1971.
- E. W. Dijkstra, "Cooperating Sequential Processes", EWD-123, 1965 (Bankacı algoritması); R. C. Holt, "Some Deadlock Properties of Computer Systems", *ACM Computing Surveys*, 1972.
- Linux çekirdek belgeleri: `Documentation/locking/lockdep-design.rst`.
- Java SE 21: `java.lang.management.ThreadMXBean`, `java.util.concurrent.locks.ReentrantLock`; `jstack` kılavuzu.
- PostgreSQL belgeleri: "Explicit Locking — Deadlocks", `deadlock_timeout`.

Son

Bölüm 7: Kilitlenme

Sonraki bölüm: Bellek Yönetimi · Terim sözlüğü