

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, açıklık

Java, jstack, lockdep

Öğrenme Çıktıları

Tanım ve kaynaklar

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

Modelleme

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

Kaçınma

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

Dört koşul

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

Tespit ve kurtarma

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

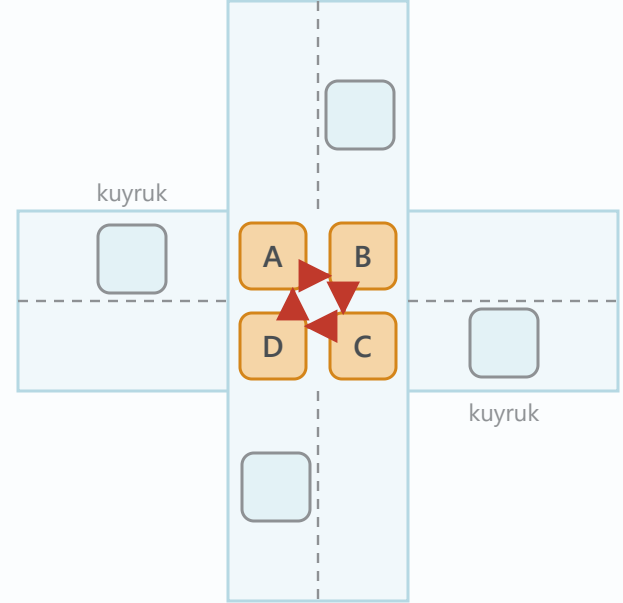
Önleme ve pratik

Kilit sıralaması, iki aşamalı kitleme, canlı kilit ve açılığı 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.

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

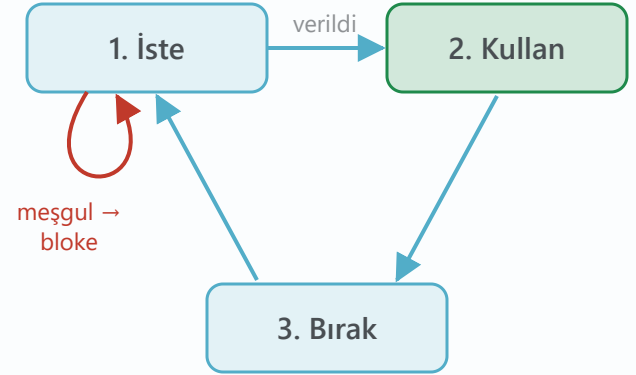
Her kaynak türü R_i 'nin W_i tane özdeş örneği (instance) olabilir; bir istek o türün *herhangi* bir örneğiyle karşılanabiliyorsa örnekler özdeştir.

GİRİŞ

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
<code>open()</code>	<code>close()</code>
<code>malloc()</code> / <code>mmap()</code>	<code>free()</code> / <code>munmap()</code>
<code>pthread_mutex_lock()</code>	<code>pthread_mutex_unlock()</code>
<code>sem_wait()</code> (down)	<code>sem_post()</code> (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

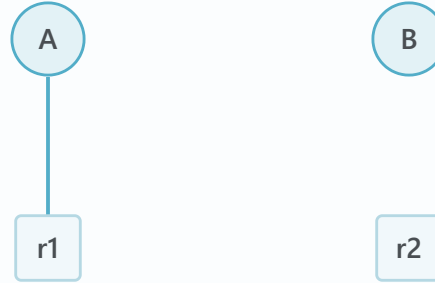
A = hazır

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) {
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 }
```



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

Adım 2 / 6

DEĞİŞKENLER

r1 = 0

r2 = 1

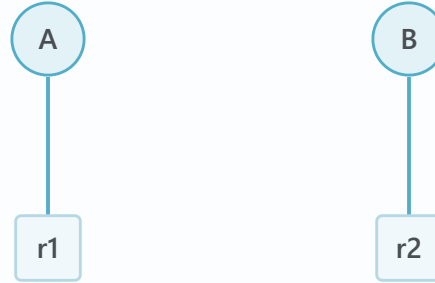
A = r1 tutuyor

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) {
9   ▶ 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

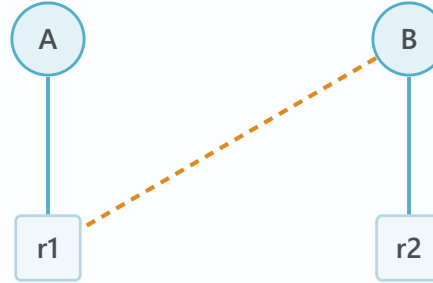
A = r1 tutuyor

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);
10  ▶ 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

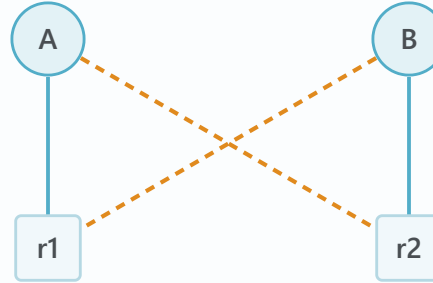
A = r1 tutuyor

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

Adım 5 / 6

DEĞİŞKENLER

r1 = 0

r2 = 0

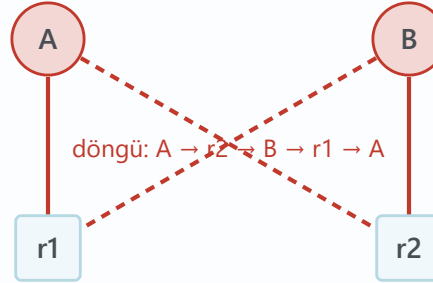
A = **bloke (r2)**

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);
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 }
```



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.

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

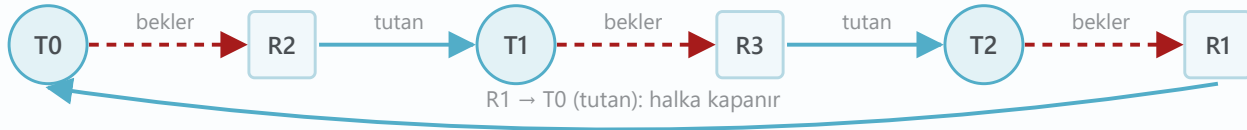
Bir süreç en az bir kaynağı *tutarken* başkasının tuttuğu yeni kaynakları ister ve bekler.

3. Kesme yok (no preemption)

Verilmiş bir kaynak zorla geri alınamaz; yalnızca tutan süreç kendi isteğiyle bırakır.

4. Döngüsel bekleme

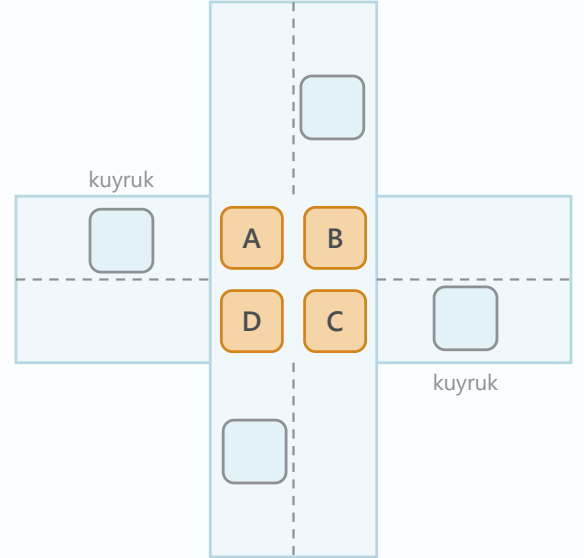
T_0, T_1 'in; T_1, T_2 'nin; ...; T_n, T_0 'ın tuttuğu kaynağı bekler.



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

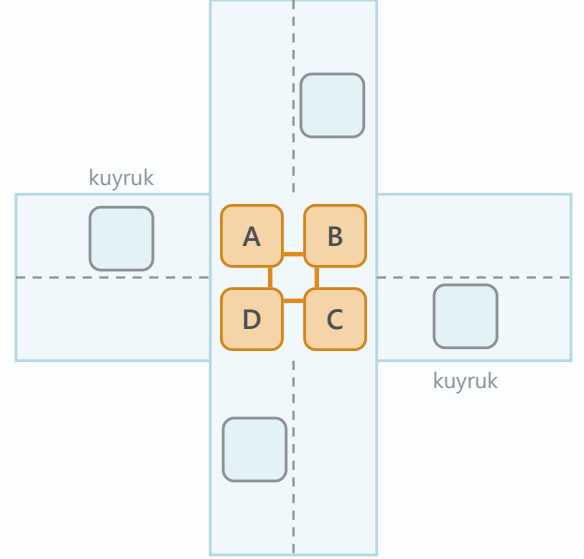
- Kavşağın dört karesi = dört **kesilemez** kaynak; araçlar = süreçler.
- Her adımda bir koşulu işaretleyin.
- Herhangi bir koşulu kırmak çözümdür:
 - trafik polisi bir aracı geri çektirir (*kesme*),
 - “kavşak boşalmadan girme” kuralı (*tut ve bekle yok*),
 - trafik ışığı (sıralama, *döngü yok*).



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

Dört Koşul Bir Arada

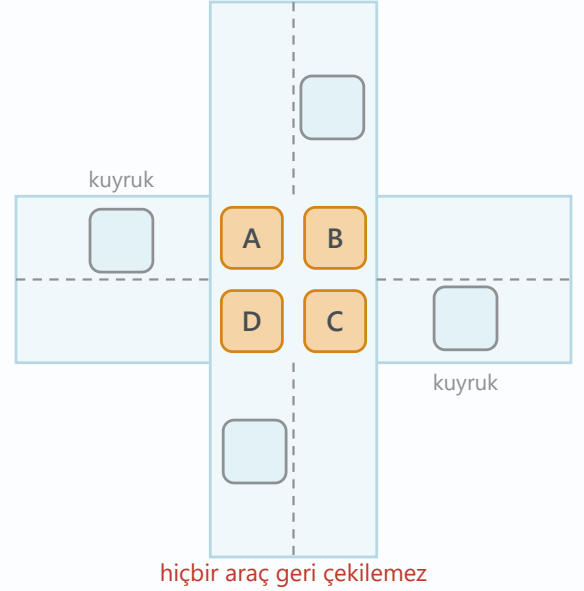
- Kavşağın dört karesi = dört **kesilemez** kaynak; araçlar = süreçler.
- Her adımda bir koşulu işaretleyin.
- Herhangi bir koşulu kırmak çözümdür:
 - trafik polisi bir aracı geri çektirir (*kesme*),
 - “kavşak boşalmadan girme” kuralı (*tut ve bekle* yok),
 - trafik ışığı (sıralama, *döngü* yok).



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

Dört Koşul Bir Arada

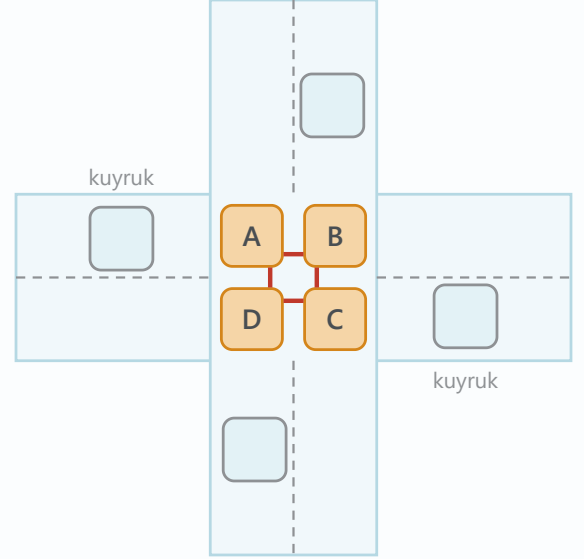
- Kavşağın dört karesi = dört **kesilemez** kaynak; araçlar = süreçler.
- Her adımda bir koşulu işaretleyin.
- Herhangi bir koşulu kırmak çözümdür:
 - trafik polisi bir aracı geri çektirir (*kesme*),
 - “kavşak boşalmadan girme” kuralı (*tut ve bekle yok*),
 - trafik ışığı (sıralama, *döngü yok*).



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 adımda bir koşulu işaretleyin.
- 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: $A \rightarrow B \rightarrow C \rightarrow D$
 $\rightarrow A$. Dört koşul birlikte: trafik kilitlendi.

Kaynak Tahsis Çizgesi

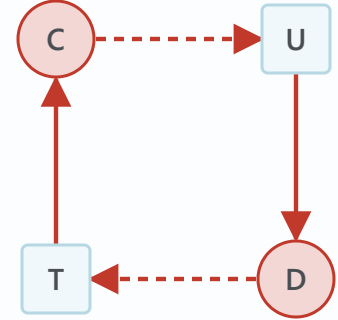
- Holt (1972): iki tür düğüm; **süreçler daire**, **kaynaklar kare**. Çok örnekli kaynakta karenin içinde her örnek için bir nokta.
- Atama kenarı** $R \rightarrow P$: R, P'ye verilmiş. **İstek kenarı** $P \rightarrow R$: P, R'yi bekliyor (bu sunumda kesikli turuncu).
- İstek karşılandığında istek kenarı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



(a) A, R'yi tutuyor



(b) B, S'yi istiyor



(c) Kilitlenme: $C \rightarrow U \rightarrow D \rightarrow T \rightarrow C$

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

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** karşılar: R, S, T verilir, sonra her süreç bir sonrakinin kaynağını ister.

A

B

C

R

S

T

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

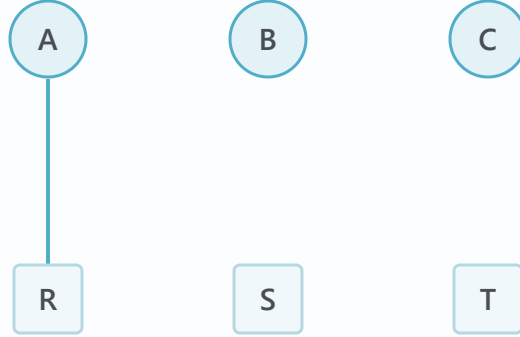
Adım 1 / 7

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

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.

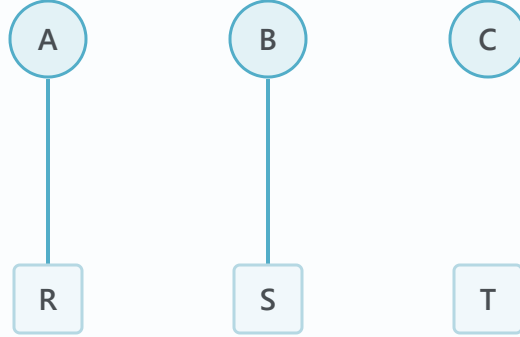
Adım 2 / 7

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

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.

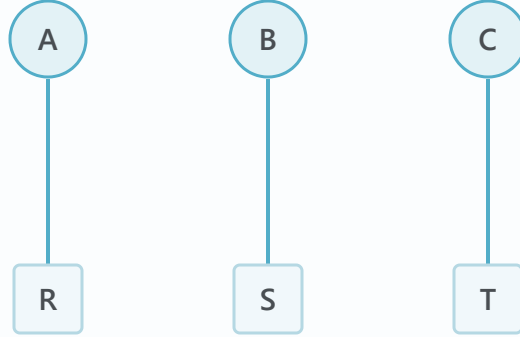
Adım 3 / 7

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

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.

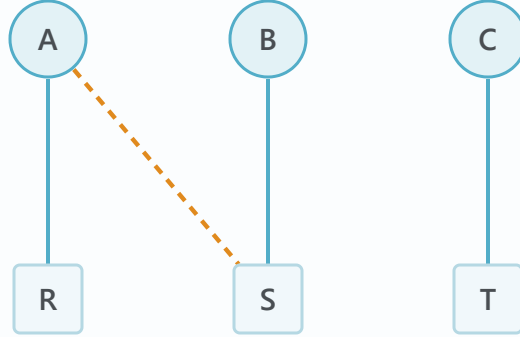
Adım 4 / 7

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

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.

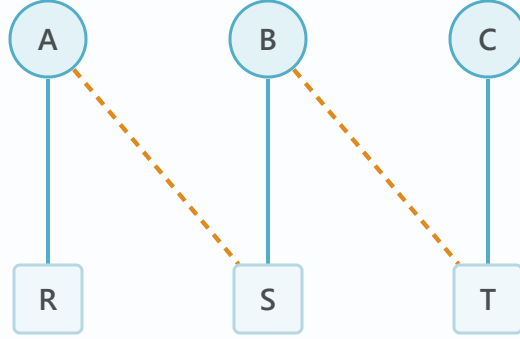
Adım 5 / 7

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.

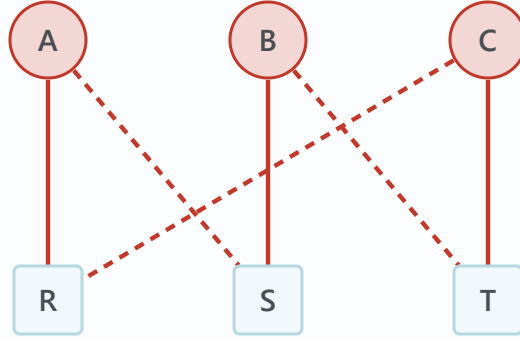
Adım 6 / 7

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** karşılar: R, S, T verilir, sonra her süreç bir sonrakinin kaynağını ister.



döngü $A \rightarrow S \rightarrow B \rightarrow T \rightarrow C \rightarrow R \rightarrow A$

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.

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.

A

B

C

R

S

T

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

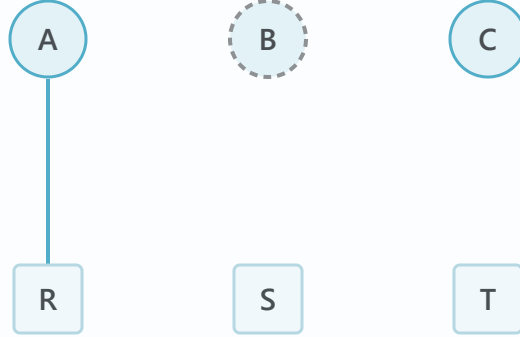
Adım 1 / 7

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.

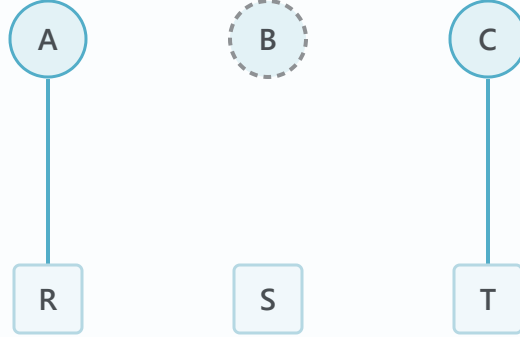
Adım 2 / 7

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.



2. C, T'yi ister ve alır.

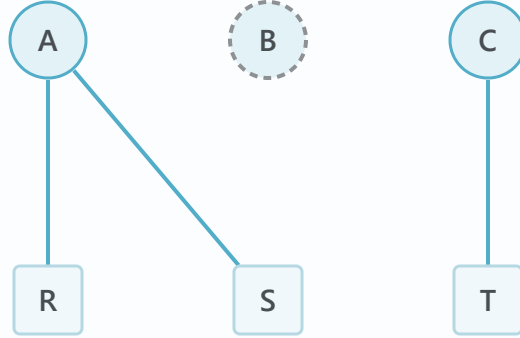
Adım 3 / 7

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.



3. A, S'yi ister ve alır (S boş).

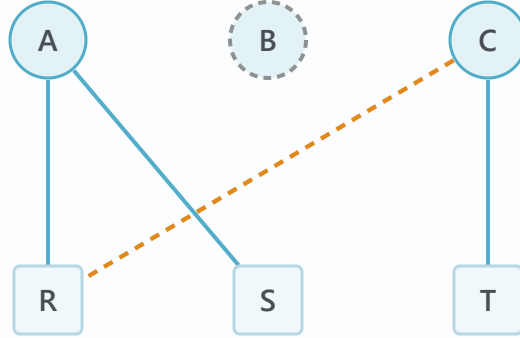
Adım 4 / 7

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.



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

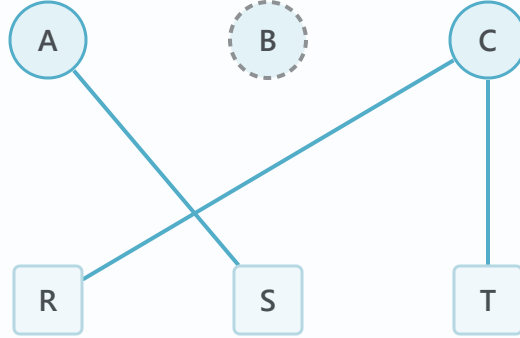
Adım 5 / 7

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.



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

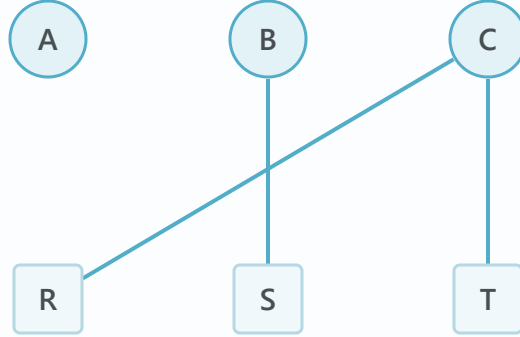
Adım 6 / 7

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.



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

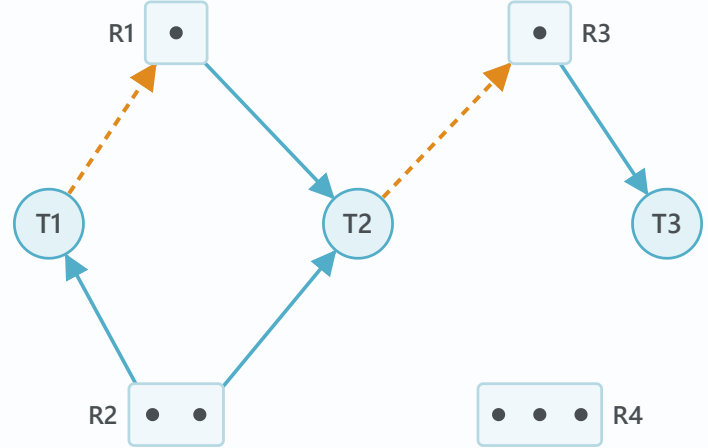
Adım 7 / 7

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

MODELLEME

Çok Örnekli Kaynaklar

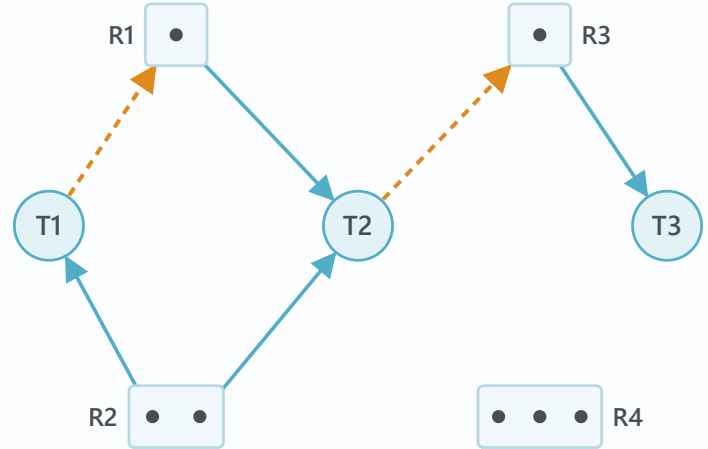
- Kaynak kümeleri: $R1 \times 1$, **$R2 \times 2$** , $R3 \times 1$, **$R4 \times 3$** .
- **T1**: bir R2 örneğini tutar, R1'i bekler.
- **T2**: R1'i ve bir R2 örneğini tutar, R3'ü bekler.
- **T3**: R3'ü tutar.
- Atama kenarı karedeki **noktadan** (örnekten) çıkar; istek kenarı karenin kendisine gider.
- 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 ($T3 \rightarrow R2$) 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.

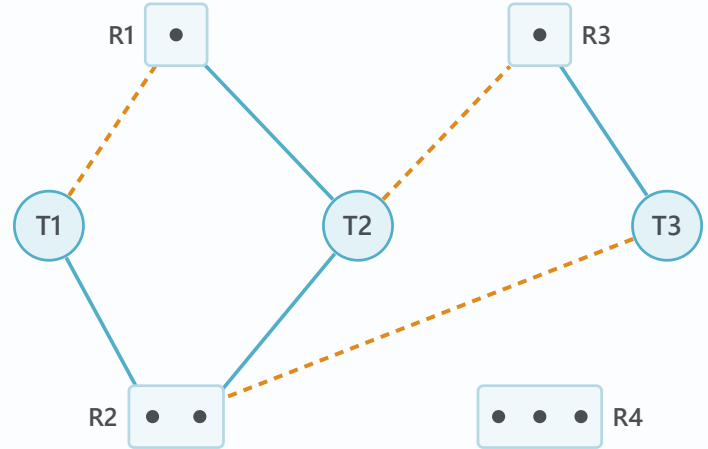


Önceki durum: döngü yok.

Adım 1 / 4

Döngü ve Kilitlenme

- Tek bir istek kenarı ($T3 \rightarrow R2$) 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.

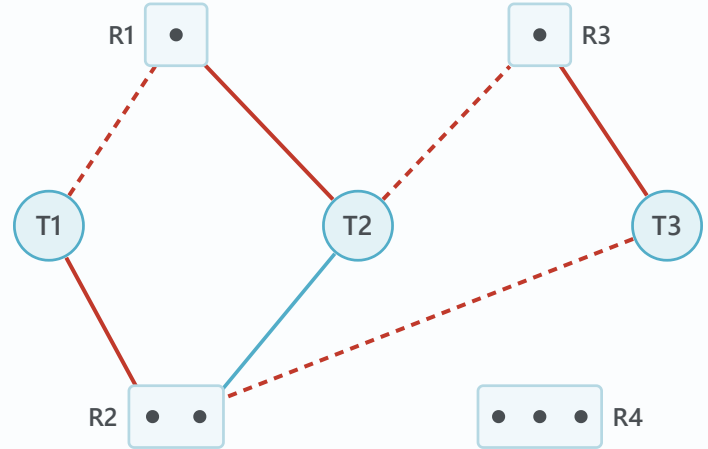


$T3$, bir $R2$ örneği ister. İki örnek de dolu ($T1$ ve $T2$ 'de) $\rightarrow T3$ bloke.

Adım 2 / 4

Döngü ve Kilitlenme

- Tek bir istek kenarı ($T3 \rightarrow R2$) 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.

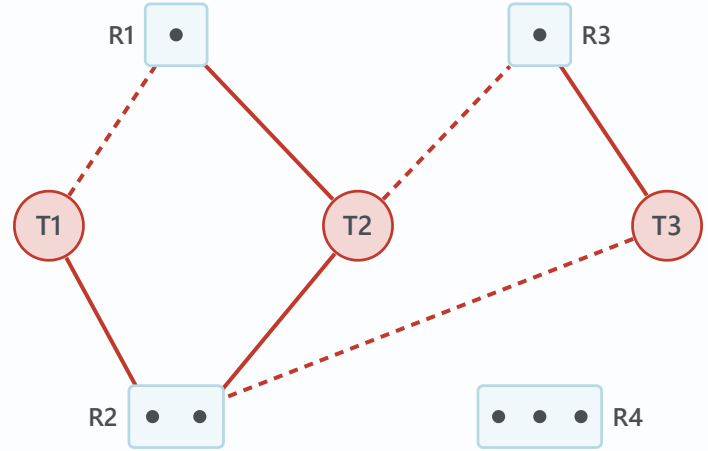


Döngü 1: $T1 \rightarrow R1 \rightarrow T2 \rightarrow R3 \rightarrow T3 \rightarrow R2 \rightarrow T1$.

Adım 3 / 4

Döngü ve Kilitlenme

- Tek bir istek kenarı ($T3 \rightarrow R2$) 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.



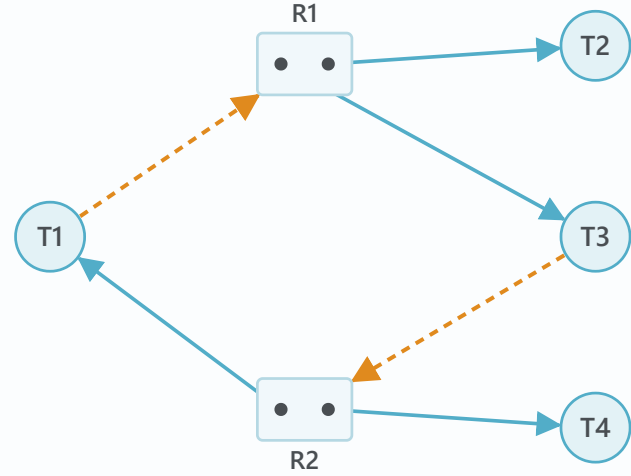
T1, T2, T3 kilitlendi

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

Adım 4 / 4

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.

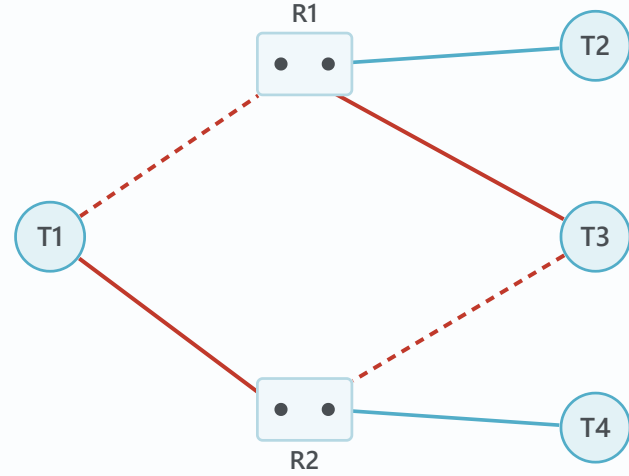


T1 bir R1 ister; R1'in iki örneği T2 ve T3'te. T3 bir R2 ister; R2'nin örnekleri T1 ve T4'te.

Adım 1 / 3

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.

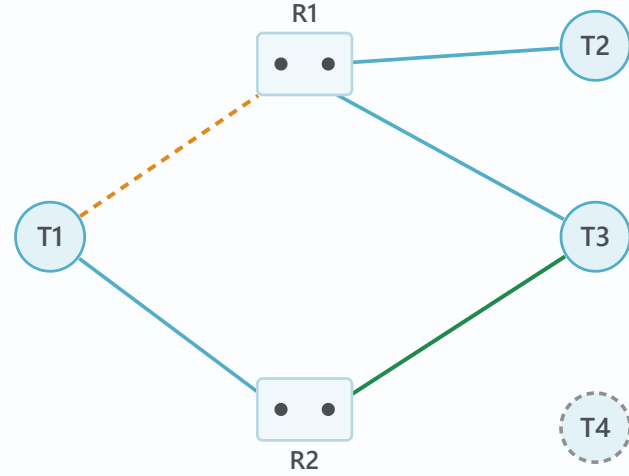


Döngü var: $T1 \rightarrow R1 \rightarrow T3 \rightarrow R2 \rightarrow T1$. Ama T2 ve T4 döngünün *dışında* ve hiçbir şey beklemiyor.

Adım 2 / 3

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**.

Döngü Kuralı

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

Tek örnek

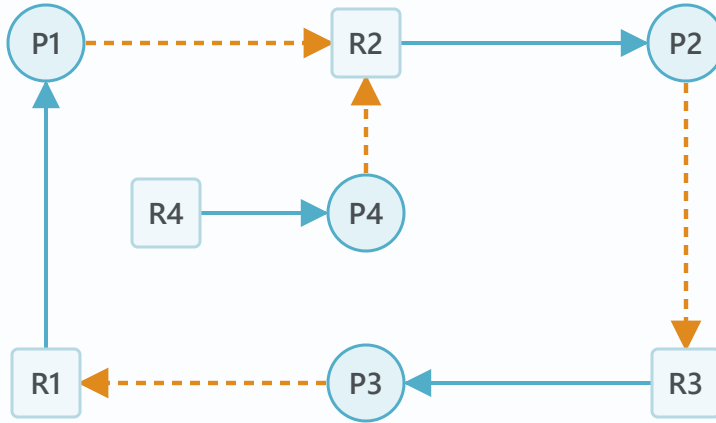
Döngü arama yeterli: DFS ile $O(V + E)$.
Döngüdeki süreçler kilitlenmiştir.

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.

Bekleme (Wait-For) Çizgesi

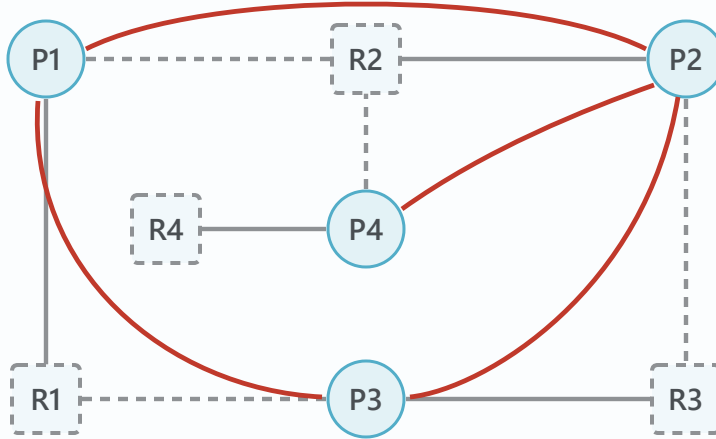


Tek örnekli kaynaklarla bir tahsis çizgesi: P1 R1'i tutar R2'yi ister, P2 R2'yi tutar R3'ü ister, P3 R3'ü tutar R1'i ister, P4 R4'ü tutar R2'yi ister.

Adım 1 / 4

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

Bekleme (Wait-For) Çizgesi

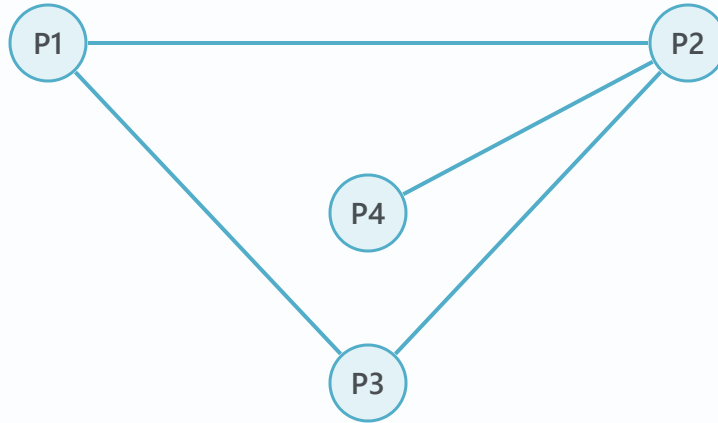


Her $P_i \rightarrow R \rightarrow P_j$ yolu tek bir $P_i \rightarrow P_j$ kenarına daraltılır (kırmızı): " P_i, P_j 'yi bekliyor".

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

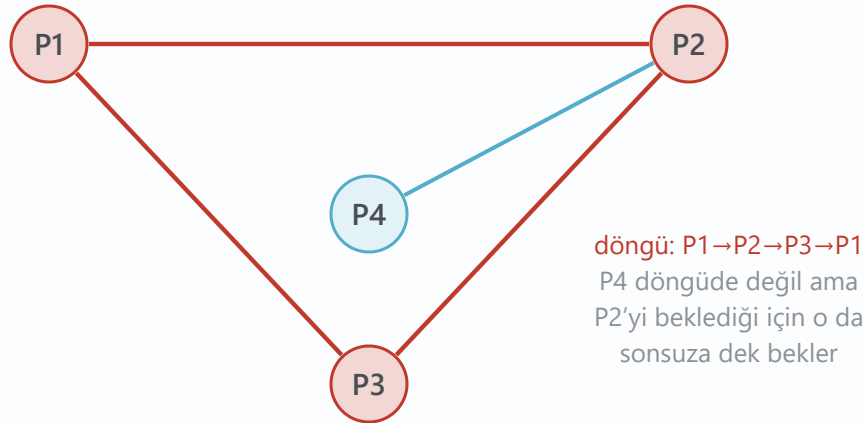


Kaynak düğümleri silinir: **bekleme (wait-for) çizgesi**.

Adım 3 / 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



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

Adım 4 / 4

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

Kilitlenmelerle Başa Çıkma Stratejileri

1. Görmezden gel

"Devekuşu algoritması". Kilitlenme nadir ve önleme maliyeti yüksekse sorunu yok sayıp gerekirse yeniden başlat.

2. Tespit et ve kurtar

Tepkisel. Kilitlenmeye izin ver, periyodik olarak tespit et, sonra kesme, geri sarma ya da sonlandırma ile çöz.

3. Dinamik kaçınma

Önleyici, çalışma anında. Her isteği, sistemi güvensiz duruma sokacaksa ertele (Bankacı). Azami ihtiyaçların önceden bilinmesini ister.

4. Önleme

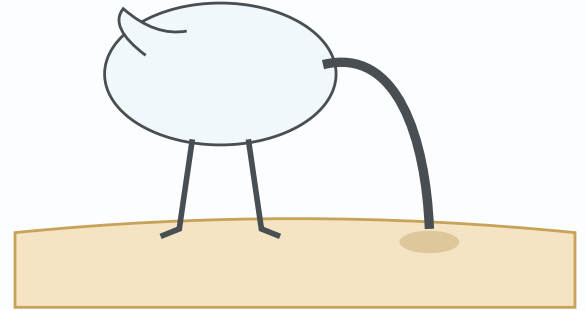
Önleyici, tasarım anında. Dört koşuldan en az birini yapısal olarak imkânsız kıl (ör. kilit sıralaması).

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

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.

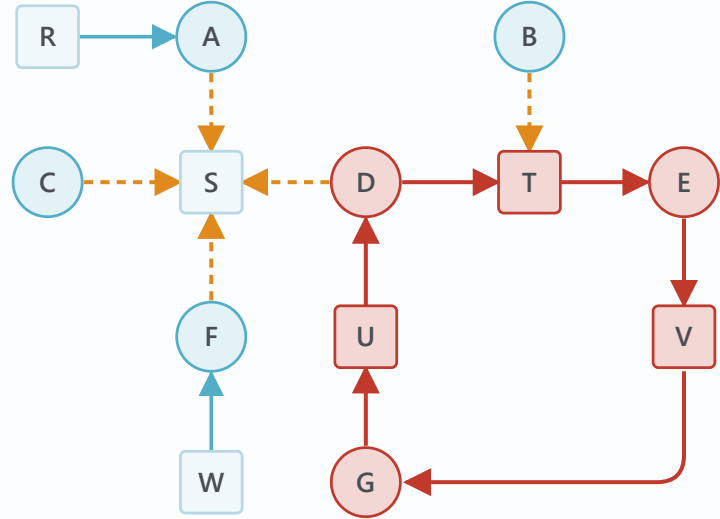
"Kilitlenme mi? Ne kilitlenmesi?"



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: $D \rightarrow T \rightarrow E \rightarrow V \rightarrow G \rightarrow U \rightarrow D$. 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.

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 işaretlessiz say.
3. Geçerli düğümü L'nin sonuna ekle. Düğüm L'de **iki kez** geçiyorsa çizgede döngü vardır: L'deki döngüyü raporla, dur.
4. Geçerli düğümünden çıkan işaretlessiz kenar var mı? Varsa 5'e, yoksa 6'ya git.
5. Rastgele bir işaretlessiz 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, 3'e git.

Özünde

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

Maliyet

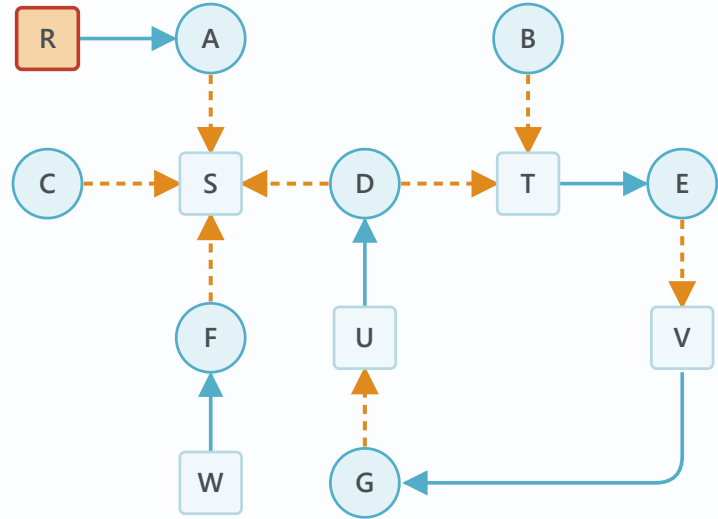
Basit sürüm her düğümünden yeniden başladığı için $O(V \cdot (V+E))$; renkli DFS (beyaz/gri/siyah) tek geçişte $O(V+E)$.

Çıktı

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

Döngü Arama: Adım Adım

- Düğümler alfabetik sırayla başlangıç alınır: R, A, B, ...
- Turuncu düğümler o anki L listesinde; kırmızı çerçeve geçerli düğüm.
- Kenar seçimi "rastgele" olabilir; burada önce S, sonra T seçiliyor.



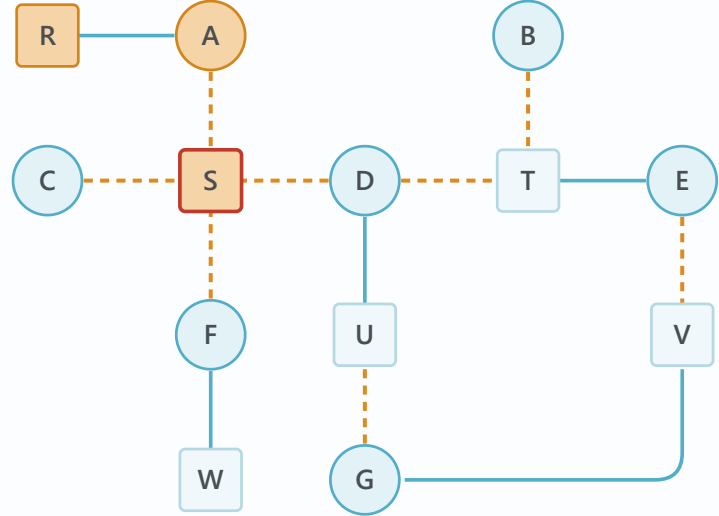
L = [R]

R'den başla. L = [R]. Tek çıkış $R \rightarrow A$.

Adım 1 / 7

Döngü Arama: Adım Adım

- Düğümler alfabetik sırayla başlangıç alınır: R, A, B, ...
- Turuncu düğümler o anki L listesinde; kırmızı çerçeve geçerli düğüm.
- Kenar seçimi "rastgele" olabilir; burada önce S, sonra T seçiliyor.



$L = [R, A, S]$

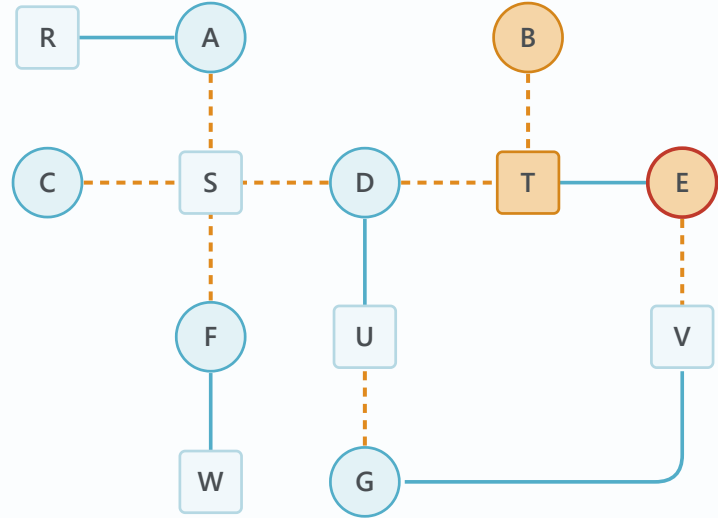
$R \rightarrow A \rightarrow S$. S'den çıkan kenar yok: çıkmaz sokak.

-

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

- Düğümler alfabetik sırayla başlangıç alınır: R, A, B, ...
- Turuncu düğümler o anki L listesinde; kırmızı çerçeve geçerli düğüm.
- Kenar seçimi "rastgele" olabilir; burada önce S, sonra T seçiliyor.

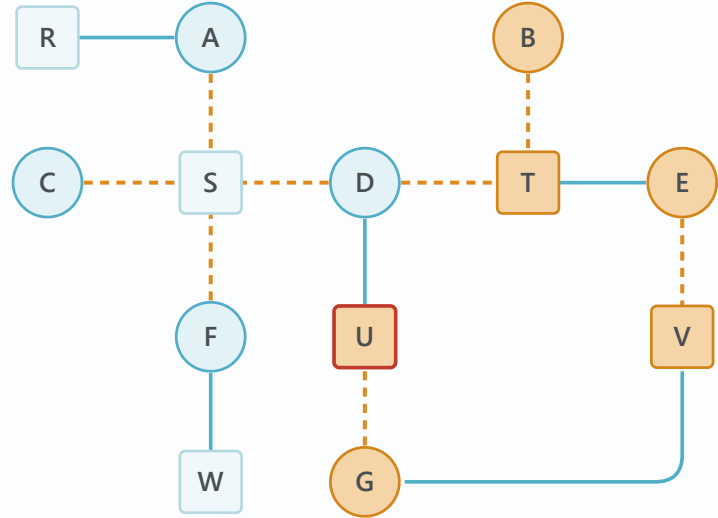


$L = [B, T, E]$

B'den başla: $B \rightarrow T \rightarrow E$.

Döngü Arama: Adım Adım

- Düğümler alfabetik sırayla başlangıç alınır: R, A, B, ...
- Turuncu düğümler o anki L listesinde; kırmızı çerçeve geçerli düğüm.
- Kenar seçimi "rastgele" olabilir; burada önce S, sonra T seçiliyor.

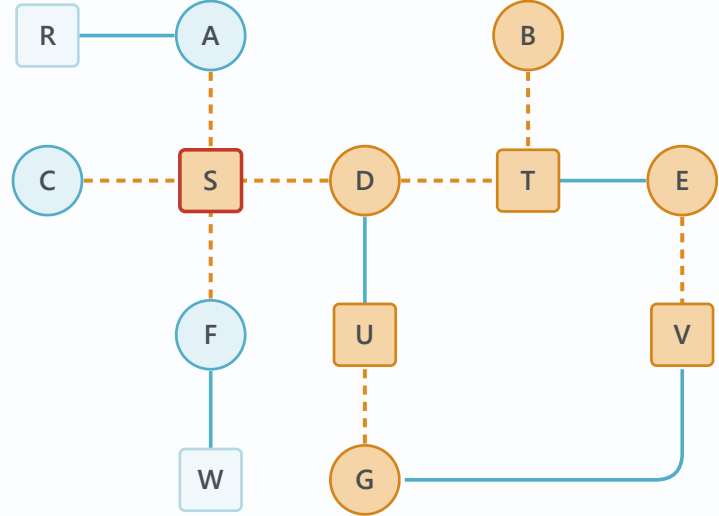


$L = [B, T, E, V, G, U]$

$E \rightarrow V \rightarrow G \rightarrow U$. Hâlâ tekrar yok.

Döngü Arama: Adım Adım

- Düğümler alfabetik sırayla başlangıç alınır: R, A, B, ...
- Turuncu düğümler o anki L listesinde; kırmızı çerçeve geçerli düğüm.
- Kenar seçimi "rastgele" olabilir; burada önce S, sonra T seçiliyor.

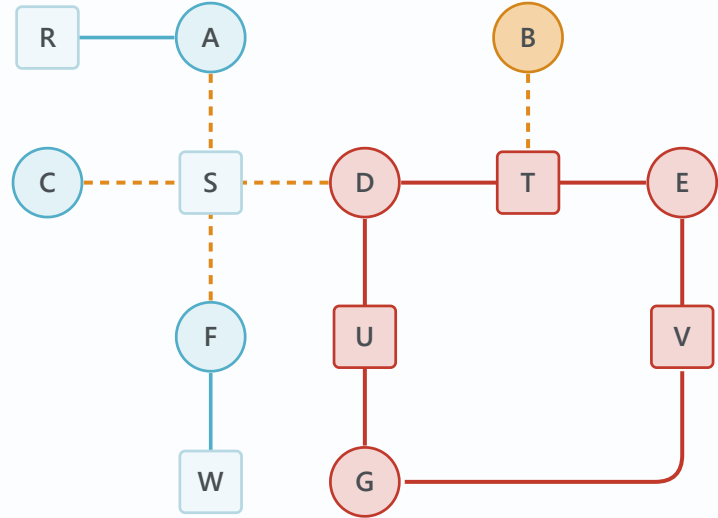


$L = [B, T, E, V, G, U, D, S]$

$U \rightarrow D$. D'den önce S denenir: çıkmaz sokak, S L'den çıkar.

Döngü Arama: Adım Adım

- Düğümler alfabetik sırayla başlangıç alınır:
R, A, B, ...
- Turuncu düğümler o anki L listesinde;
kırmızı çerçeve geçerli düğüm.
- Kenar seçimi "rastgele" olabilir; burada
önce S, sonra T seçiliyor.



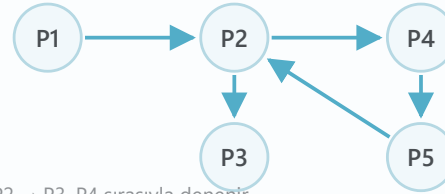
$L = [B, T, E, V, G, U, D, T]$

$D \rightarrow T$: T, L'de **ikinci kez** $\rightarrow$ 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 }
```



beyaz: ziyaret
gri: yolda
siyah: bitti

P2 → P3, P4 sırasıyla denir

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

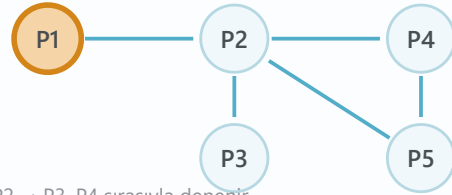
P5 = B

ÇAĞRI YIĞINI

main

Döngü Arama: Renkli DFS Kodu

```
1 enum { BEYAZ, GRI, SIYAH };
2 int renk[N];          /* hepsi BEYAZ */
3 bool dongu(int u) {
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 }
```



beyaz: ziyaret
gri: yolda
siyah: bitti

P2 → P3, P4 sırasıyla denir

P1 griye boyanır: şu anki yolun üzerinde.

Adım 2 / 9

DEĞİŞKENLER

P1 = G

P2 = B

P3 = B

P4 = B

P5 = B

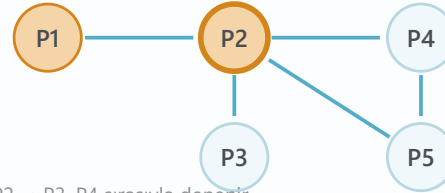
ÇAĞRI YIĞINI

dongu(P1)

main

Döngü Arama: Renkli DFS Kodu

```
1 enum { BEYAZ, GRI, SIYAH };
2 int renk[N];          /* hepsi BEYAZ */
3 bool dongu(int u) {
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 }
```



beyaz: ziyaret
gri: yolda
siyah: bitti

P2 → P3, P4 sırasıyla denir

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

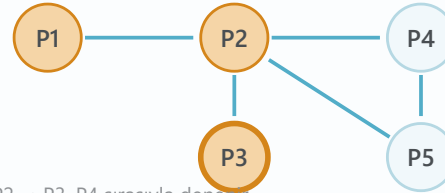
dongu(P2)

dongu(P1)

main

Döngü Arama: Renkli DFS Kodu

```
1 enum { BEYAZ, GRI, SIYAH };
2 int renk[N];          /* hepsi BEYAZ */
3 bool dongu(int u) {
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 }
```



beyaz: ziyaret
gri: yolda
siyah: bitti

P2 → P3, P4 sırasıyla denir

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)

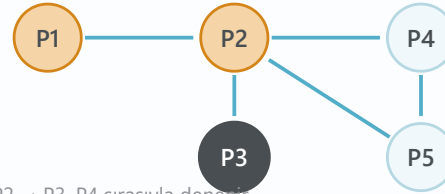
dongu(P2)

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    }
12    renk[u] = SIYAH;    /* bitti */
13    return false;
14 }
```



beyaz: ziyaret
gri: yolda
siyah: bitti

P2 → P3, P4 sırasıyla denemir

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)

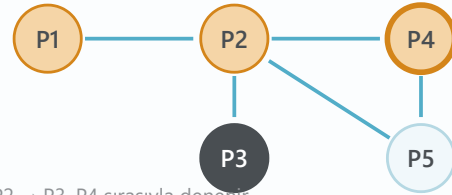
main

Döngü Arama: Renkli DFS Kodu

```

1 enum { BEYAZ, GRI, SIYAH };
2 int renk[N];          /* hepsi BEYAZ */
3 bool dongu(int u) {
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 }

```



beyaz: ziyaret
gri: yolda
siyah: bitti

P2 → P3, P4 sırasıyla deniyor

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)

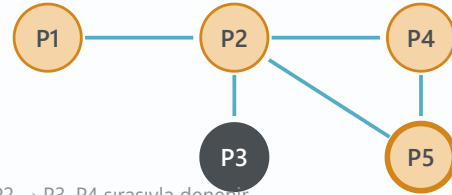
main

Döngü Arama: Renkli DFS Kodu

```

1 enum { BEYAZ, GRI, SIYAH };
2 int renk[N];          /* hepsi BEYAZ */
3 bool dongu(int u) {
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 }

```



beyaz: ziyaret
gri: yolda
siyah: bitti

P2 → P3, P4 sırasıyla deniyor

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)

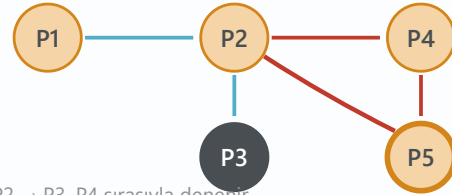
main

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;
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 }

```



beyaz: ziyaret
gri: yolda
siyah: bitti

P2 → P3, P4 sırasıyla deniyor

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)

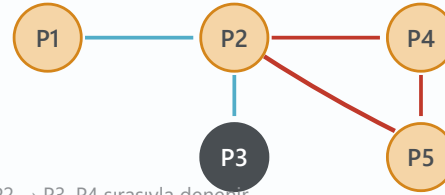
main

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 */
9         if (renk[v] == BEYAZ && dongu(v))
10            return true;
11     }
12     renk[u] = SIYAH;   /* bitti */
13     return false;
14 }

```



beyaz: ziyaret
gri: yolda
siyah: bitti

P2 → P3, P4 sırasıyla deniyor

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

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

Boştaki kaynaklar

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

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

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

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

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

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

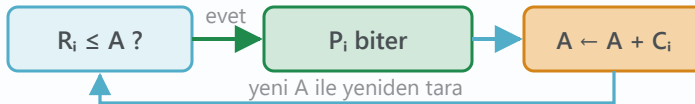
- n süreç, m kaynak türü. C'nin i. satırı P_i'nin *şu an tuttuğu*, R'nin i. satırı P_i'nin *şu an istediği* kaynaklardır.
- Vektör karşılaştırması: $X \leq Y$ ancak ve ancak **her** j için $X_j \leq Y_j$. Ör. (1 0 1 0) $\leq$ (2 1 0 0) **değil**.

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
```

hiçbiri yoksa: dur



- **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: $A \leftarrow A + C_i$ (biten sürecin tuttukları boş havuza döner).

Tespit Örneği

► $A \leftarrow (2 \ 1 \ 0 \ 0)$

► tüm P_i işaretsiz

3 tekrarla:

4 $R_i \leq A$ olan işaretsiz P_i bul

5 bulunamazsa: çık

6 $A \leftarrow A + C_i$

7 P_i '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

$E = (4 \ 2 \ 3 \ 1)$

$A = \begin{bmatrix} 2 & 1 & 0 & 0 \end{bmatrix}$

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

Adım 1 / 9

DEĞİŞKENLER

$A = (2 \ 1 \ 0 \ 0)$

işaretleli = -

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 X

P1: R₁ = (2 0 0 1). Boş CD yok → şimdilik bitemez.

Adım 2 / 9

DEĞİŞKENLER

A = (2 1 0 0)

işaretili = -

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

Tespit Örneği

1 $A \leftarrow (2 \ 1 \ 0 \ 0)$
 2 tüm P_i işaretli
 3 tekrarla:

► $R_i \leq A$ olan işaretli P_i bul

5 bulunamazsa: çık
 6 $A \leftarrow A + C_i$
 7 P_i 'yi işaretli
 8 işaretli kilitli

C (tutulan)

	TP	PI	Tr	CD
P1	0	0	1	0
P2	2	0	0	1
P3	0	1	2	0

R (istek)

	TP	PI	Tr	CD
P1	2	0	0	1
P2	1	0	1	0
P3	2	1	0	0

$E = (4 \ 2 \ 3 \ 1)$

$A = \begin{bmatrix} 2 & 1 & 0 & 0 \end{bmatrix}$

$(1 \ 0 \ 1 \ 0) \leq (2 \ 1 \ 0 \ 0)$? tarayıcı yok X

$P2: R_2 = (1 \ 0 \ 1 \ 0)$. Boş tarayıcı yok X.

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 $A \leftarrow (2\ 1\ 0\ 0)$
 2 tüm P_i işaretli
 3 tekrarla:

► $R_i \leq A$ olan işaretli P_i bul

5 bulunamazsa: çık
 6 $A \leftarrow A + C_i$
 7 P_i 'yi işaretle
 8 işaretlisizler kilitli

C (tutulan)

	TP	PI	TR	CD
P1	0	0	1	0
P2	2	0	0	1
P3	0	1	2	0

R (istek)

	TP	PI	TR	CD
P1	2	0	0	1
P2	1	0	1	0
P3	2	1	0	0

$E = (4\ 2\ 3\ 1)$

$A = \begin{array}{|c|c|c|c|} \hline 2 & 1 & 0 & 0 \\ \hline \end{array}$

$(2\ 1\ 0\ 0) \leq (2\ 1\ 0\ 0) \checkmark$

$P3: R_3 = (2\ 1\ 0\ 0) \leq A \checkmark$. P3 çalışıp bitebilir.

Adım 4 / 9

DEĞİŞKENLER

$A = (2\ 1\ 0\ 0)$

işaretili = -

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
8 işaretsizler kilitli
    
```

C (tutulan)

	TP	PI	Tr	CD
P1	0	0	1	0
P2	2	0	0	1
P3	0	1	2	0

E = (4 2 3 1)

R (istek)

	TP	PI	Tr	CD
P1	2	0	0	1
P2	1	0	1	0
P3	2	1	0	0

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)

işaretili = 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:
```

► $R_i \leq A$ olan işaretsiz P_i bul

5 bulunamazsa: çık

6 $A \leftarrow A + C_i$

7 P_i 'yi işaretle

8 işaretsizler kilitli

C (tutulan)

	TP	PI	TR	CD
P1	0	0	1	0
P2	2	0	0	1
P3	0	1	2	0

R (istek)

	TP	PI	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 X

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

Adım 6 / 9

DEĞİŞKENLER

A = (2 2 2 0)

işaretili = P3

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

Tespit Örneği

1 $A \leftarrow (2\ 1\ 0\ 0)$
 2 tüm P_i işaretsiz
 3 tekrarla:

► $R_i \leq A$ olan işaretsiz P_i bul

5 bulunamazsa: çık

► $A \leftarrow A + C_i$

► P_i 'yi işaretle

8 işaretsizler kilitli

C (tutulan)

	TP	PI	Tr	CD
P1	0	0	1	0
P2	2	0	0	1
P3	0	1	2	0

$E = (4\ 2\ 3\ 1)$

R (istek)

	TP	PI	Tr	CD
P1	2	0	0	1
P2	1	0	1	0
P3	2	1	0	0

A

4	2	2	1
---	---	---	---

$P2: (1\ 0\ 1\ 0) \leq (2\ 2\ 2\ 0) \checkmark \rightarrow \text{biter}, C_2 = (2\ 0\ 0\ 1)$ döner: $A = (4\ 2\ 2\ 1)$.

Adım 7 / 9

DEĞİŞKENLER

$A = (4\ 2\ 2\ 1)$

işaretili = 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:
```

► $R_i \leq A$ olan işaretsiz P_i bul

5 bulunamazsa: çık

► $A \leftarrow A + C_i$

► P_i 'yi işaretle

8 işaretsizler kilitli

C (tutulan)

	T _p	P _i	T _r	C _D
P1	0	0	1	0
P2	2	0	0	1
P3	0	1	2	0

R (istek)

	T _p	P _i	T _r	C _D
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)

işaretili = 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	PI	TR	CD
P1	0	0	1	0
P2	2	0	0	1
P3	0	1	2	0

R (istek)

	TP	PI	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)

işaretili = 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

$$E = (4 \ 2 \ 3 \ 1)$$

R (istek)

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

$$E = (4 \ 2 \ 3 \ 1)$$

R (istek)

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

A	2	1	0	0
---	---	---	---	---

P1 X (CD yok)

P1 X (CD gerekli, boşta yok).

Adım 2 / 4

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

$$E = (4 \ 2 \ 3 \ 1)$$

R (istek)

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

A	2	1	0	0
---	---	---	---	---

P2 X (tarayıcı yok)

P2 X (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

E = (4 2 3 1)

	R (istek)			
	Tp	Pl	Tr	CD
P1	2	0	0	1
P2	1	0	1	0
P3	2	1	0	1

A	2	1	0	0
---	---	---	---	---

P3 X → P1, P2, P3 kilitli

P3 X (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.

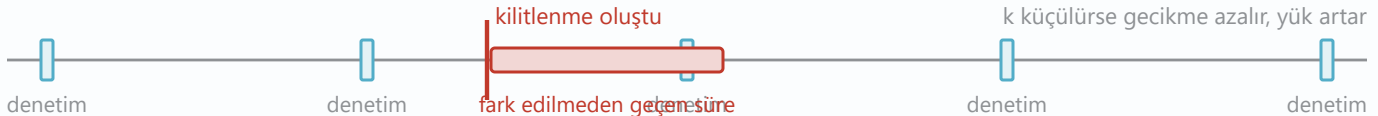
Periyodik

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.



Kilitlenmeden Kurtarma

Kaynak kesme

Bir kaynağı sahibinden geçici olarak alıp başka sürece ver (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)

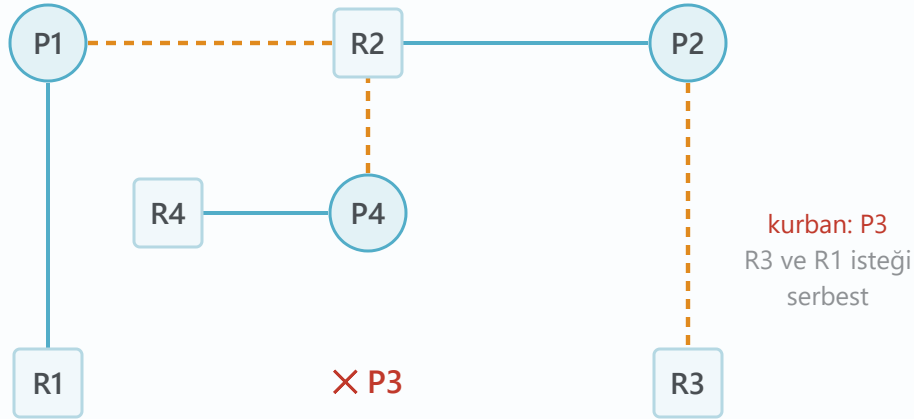
En kaba yöntem. Ya döngüdeki tüm süreçler ya da döngü kırılana kadar *teker teker* (her öldürmeden sonra tespit yeniden çalışır). Döngü dışından bir süreç de seçilebilir.

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 sayısını maliyete ekle.

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



Kurtarma: Adım Adım

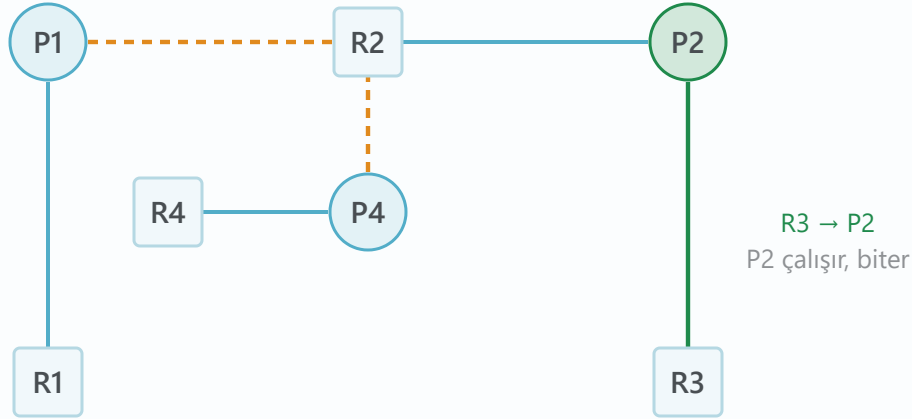


Kurban: en az iş kaybettirecek süreç, P3. Sonlandırılır (ya da denetim noktasına geri sarılır); tuttuğu R3 serbest kalır.

Adım 2 / 4

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

Kurtarma: Adım Adım

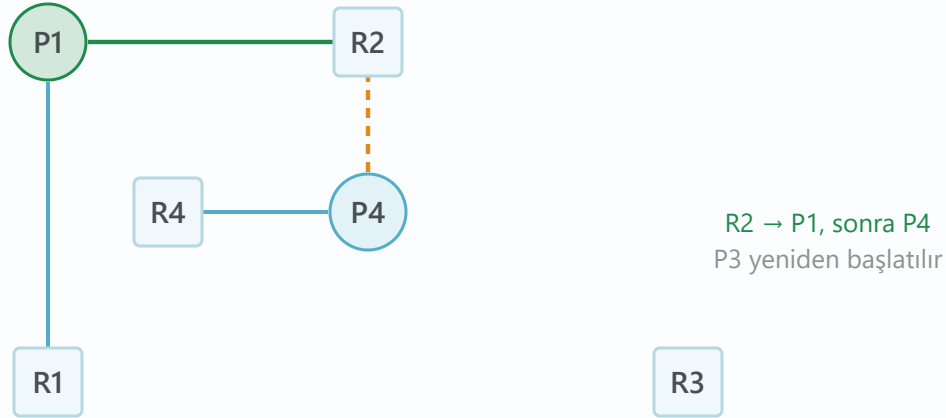


R3 bekleyen P2'ye verilir. P2 işini bitirip R2 ve R3'ü bırakır.

Adım 3 / 4

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

Kurtarma: Adım Adım



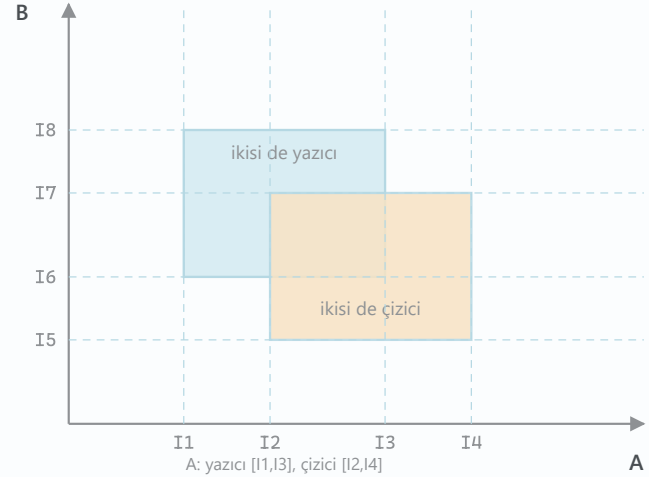
R2 sıradaki bekleyene (P1) verilir; P1 bitince P4. P3 baştan (ya da denetim noktasından) yeniden çalıştırılır.

Adım 4 / 4

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

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**.



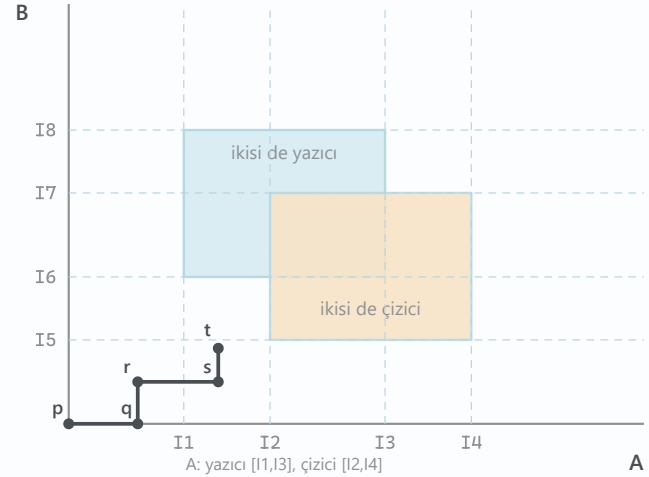
Yatay eksen A'nın, dikey eksen B'nin yürüttüğü komutlar. Tek CPU: yörünge yalnızca **sağa** (A çalışır) ya da **yukarı** (B çalışır) ilerler.

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**.



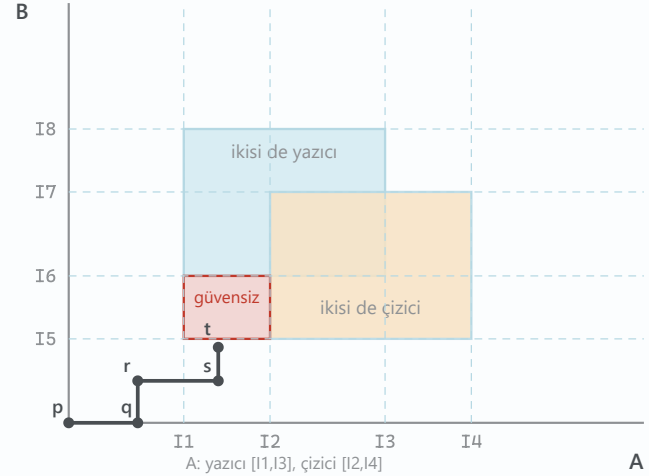
$p \rightarrow q$: A çalışır; $q \rightarrow r$: B çalışır; $r \rightarrow s$: A yazıcıyı alır (I1). $s \rightarrow t$: B ilerler, I5'te çiziciyi isteyecek.

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**.



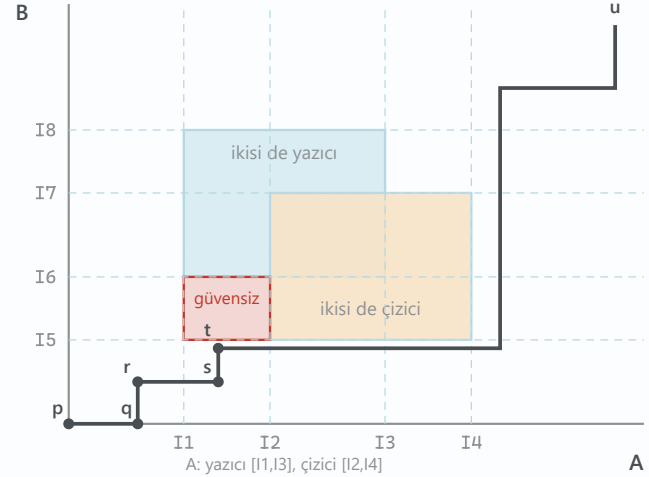
Kırmızı kutuya girilirse A yazıcıyı, B çiziciyi tutar ve ikisi de taralı bölgeye girmek zorundadır: kilitlenme **kaçınılmaz**. Bu bölge **güvensizdir**.

Adım 3 / 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**.



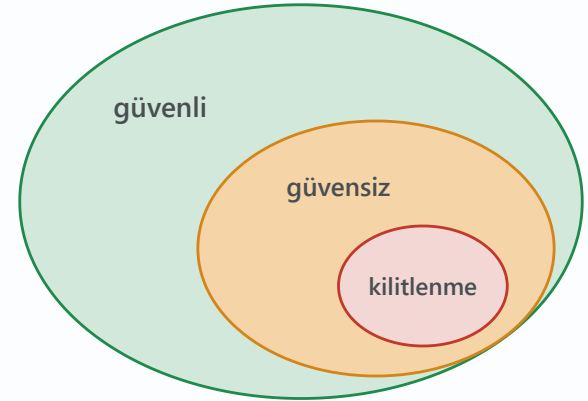
t noktasında İS, B'nin çizici isteğini erteler; A I4'e kadar çalışıp ikisini de bırakır. Sonra B devam eder: kilitlenme yok.

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** $\langle P_1, \dots, P_n \rangle$ vardır.
- P_i 'nin kalan ihtiyacı $\leq$ boştakiler + P_j ($j < i$) tarafından tutulanlar.
- **Güvensiz $\neq$ 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 = $\text{Max} - \text{Has}$.
- 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.

Adım 1 / 5

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.

	Has	Max
A	3	9
B	4	4
C	2	7

Boş: 1

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

Adım 2 / 5

Tanenbaum MOS Şekil 6-9.

Güvenli Durum Örneği

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

	Has	Max
A	3	9
B	0	–
C	2	7

Boş: 5

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

Adım 3 / 5

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.

	Has	Max
A	3	9
B	0	–
C	7	7

Boş: 0

(d) C'nin kalan ihtiyacı $5 \leq 5$: C'ye 5 verilir.

Adım 4 / 5

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 = $\text{Max} - \text{Has}$.
- 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.

Adım 5 / 5

Tanenbaum MOS Şekil 6-9.

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.

	Has	Max
A	4	9
B	2	4
C	2	7

Boş: 2

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

Adım 2 / 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.

	Has	Max
A	4	9
B	4	4
C	2	7

Boş: 0

(c) Yalnızca B bitebilir (ihtiyaç $2 \leq 2$):
B'ye 2 verilir.

Adım 3 / 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.

	Has	Max
A	4	9
B	0	–
C	2	7

Boş: 4

A 5, C 5 istiyor; boş 4

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

Adım 4 / 4

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.

Adım 1 / 3

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 $2 \leq 2 \rightarrow C$ biter (boş 4), sonra D ($3 \leq 4 \rightarrow$ boş 8), A, B. **Güvenli.**

Adım 2 / 3

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.

Adım 3 / 3

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: —

$(7\ 4\ 3) \leq (3\ 3\ 2)$? $\times$

T0: Need $(7\ 4\ 3) \leq (3\ 3\ 2)$? 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:
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

$(1\ 2\ 2) \leq (3\ 3\ 2) \checkmark$ Work += (2 0 0)

T1: $(1\ 2\ 2) \leq (3\ 3\ 2) \checkmark \rightarrow$ 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

$(6\ 0\ 0) \leq (5\ 3\ 2)?$ X

T2: $(6\ 0\ 0) \leq (5\ 3\ 2)$? A yetmez X.

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:
6     ▶ Work ← Work + Allocation[i]
7     ▶ Finish[i] ← true
8 if tüm Finish[i]: GÜVENLİ

```

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

Sıra: T1 → T3

$(0\ 1\ 1) \leq (5\ 3\ 2) \checkmark$ Work += (2 1 1)

T3: $(0\ 1\ 1) \checkmark \rightarrow$ 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:
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		
7	4	5

Sıra: T1 → T3 → T4

$(4\ 3\ 1) \leq (7\ 4\ 3) \checkmark$ Work += (0 0 2)

T4: $(4\ 3\ 1) \checkmark \rightarrow$ 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:
6     ▶ Work ← Work + Allocation[i]
7     ▶ Finish[i] ← true
8 if tüm Finish[i]: GÜVENLİ

```

Allocation				Need				Work		
	A	B	C		A	B	C			
T0	0	1	0	T0	7	4	3	7	5	5
T1	2	0	0	T1	1	2	2			
T2	3	0	2	T2	6	0	0			
T3	2	1	1	T3	0	1	1			
T4	0	0	2	T4	4	3	1			

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).

Adım 7 / 9

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:
6     ▶ Work ← Work + Allocation[i]
7     ▶ Finish[i] ← true
8 if tüm Finish[i]: GÜVENLİ

```

Allocation				Need				Work		
	A	B	C		A	B	C			
T0	0	1	0	T0	7	4	3	10	5	7
T1	2	0	0	T1	1	2	2			
T2	3	0	2	T2	6	0	0			
T3	2	1	1	T3	0	1	1			
T4	0	0	2	T4	4	3	1			

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 alınır.
- Hesaplar: Python betiğiyle doğrulandı.

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

$$\text{Request}_1 = (1 \ 0 \ 2) \leq \text{Need}_1 = (1 \ 2 \ 2) \checkmark$$

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

Adım 1 / 4

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 alınır.
- Hesaplar: Python betiğiyle doğrulandı.

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

$$(1\ 0\ 2) \leq \text{Available} = (3\ 3\ 2) \checkmark$$

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

varmış gibi: Avail (2 3 0), Alloc₁ (3 0 2), Need₁ (0 2 0)

İsteği **vermiş gibi** yeni durumu hesapla.

Adım 3 / 4

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 alınır.
- Hesaplar: Python betiğiyle doğrulandı.

Allocation				Need				Work			
	A	B	C		A	B	C				
T0	0	1	0	T0	7	4	3		10	5	7
T1	3	0	2	T1	0	2	0				
T2	3	0	2	T2	6	0	0				
T3	2	1	1	T3	0	1	1				
T4	0	0	2	T4	4	3	1				

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**.

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

T4: $(3\ 3\ 0) \leq \text{Avail } (2\ 3\ 0)$? $\times \rightarrow$ 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				Need				Avail			
	A	B	C		A	B	C				
T0	0	1	0	T0	7	4	3		2	3	0
T1	3	0	2	T1	0	2	0				
T2	3	0	2	T2	6	0	0				
T3	2	1	1	T3	0	1	1				
T4	0	0	2	T4	4	3	1				

T0: $(0\ 2\ 0) \leq \text{Need}_0 \checkmark$ ve $\leq \text{Avail}\ (2\ 3\ 0) \checkmark$

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

varmış gibi: Avail (2 1 0), Need₀ (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...

Adım 3 / 4

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

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 */
▶ 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):
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 simülasyonu ile 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):
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	—

$(0\ 1\ 1) \leq \text{Available } (3\ 3\ 2)$: 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 simülasyonu ile 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):
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 simülasyonu ile 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):
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 simülasyonu ile 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):
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 (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 simülasyonu ile 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):
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 (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 simülasyonu ile 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):
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 (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 simülasyonu ile 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):
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 (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 simülasyonu ile 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):
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 (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 simülasyonu ile 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):
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 (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 simülasyonu ile doğrulandı.

Çoklu Kaynak: İkinci Örnek

Atanmış					Hâlâ gereken				
	Tp	Pl	Yz	CD		Tp	Pl	Yz	CD
A	3	0	1	1	A	1	1	0	0
B	0	1	0	0	B	0	1	1	2
C	1	1	1	0	C	3	1	0	0
D	1	1	0	1	D	0	0	1	0
E	0	0	0	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 $\leq A$ olan satırı ara" adımları burada güvenlik testidir; aynı mantık tespit algoritmasında *istek* matrisiyle, kaçınmada *kalan ihtiyaç* matrisiyle kullanılır.

Çoklu Kaynak: İkinci Örnek

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

$D (0 \ 0 \ 1 \ 0) \leq A \rightarrow$ biter, $A = (2 \ 1 \ 2 \ 1)$; E, A, B, C de sırayla biter. Durum güvenli.

Adım 2 / 4

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

Çoklu Kaynak: İkinci Örnek

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

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

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

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

Adım 3 / 4

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

Çoklu Kaynak: İkinci Örnek

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

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

sonra E son yazıcıyı ister $\rightarrow A = (1 \ 0 \ 0 \ 0)$, kimse bitemez $\rightarrow$ ERTELE

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

Adım 4 / 4

Tanenbaum MOS Şekil 6-12. "Her satırda $\leq A$ 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: [Deadlock & Banker's Algorithm Simulator](#) (İngilizce) — Allocation, Max, Need ve Available değerlerini girip güvenli diziyi ve istek kontrolünü adım adım izleyin.

ÖNLEME

Önleme: Koşullara Saldırmak

Koşul	Saldırı	Bedeli
Karşılıklı dışlama	Kaynağı paylaşılabilir yap: biriktirme (spooling), salt okunur paylaşım, kilitsiz yapılar	Çoğu kaynak doğası gereği paylaşamaz (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; <code>tryLock</code> + 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**.

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**.



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.



Kesme Yok Koşuluna Saldırı

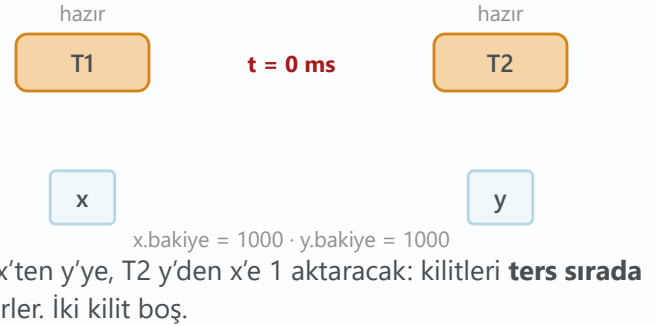
- **Geri alma protokolü:** kaynak tutan bir süreç hemen karşılanamayan bir istek yaparsa, tuttuğu tüm kaynaklar *örtük olarak bırakılır*; süreç hepsini birlikte alabildiğinde yeniden başlar.
- Yalnızca durumu kaydedilip geri yüklenebilen kaynaklara uygundur: CPU yazmaçları, bellek, veritabanı işlemi (abort).
- **Sanallaştırma:** yazıcı biriktirilir, bellek sayfaları.
- Mutex için: **zaman aşımli deneme**. Edinemeyen iş parçacığı elindekini kendisi bırakır ve rastgele bir süre sonra yeniden dener.
- Çalıştırıldı (JDK 21): 2 iş parçacığı $\times$ 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(); }
}
/* çağırın: while (!dene(x, y, 1))
    Thread.sleep(rastgele); */
```

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

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 /* çağırın: while (!dene(x, y, 1))
14     Thread.sleep(rastgele); */
```



Adım 1 / 10

DEĞİŞKENLER

t = 0 ms

x = boş

y = boş

x.bakiye = 1000

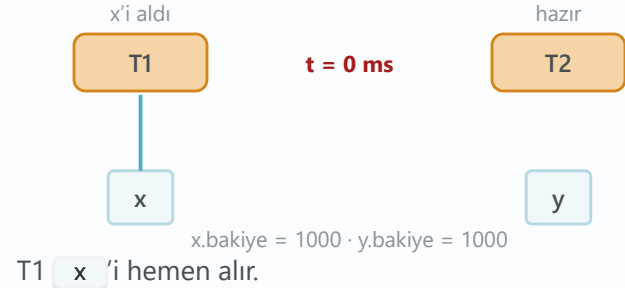
y.bakiye = 1000

ÇIKTI



tryLock ile Geri Alma: Kod İzleme

```
1 boolean dene(Hesap k, Hesap h, int m)
2     throws InterruptedException {
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 /* çağırın: while (!dene(x, y, 1))
14     Thread.sleep(rastgele); */
```



Adım 2 / 10

DEĞİŞKENLER

t = 0 ms

x = T1

y = boş

x.bakiye = 1000

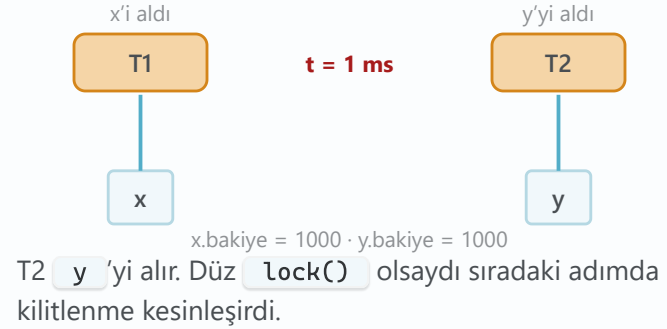
y.bakiye = 1000

ÇIKTI



tryLock ile Geri Alma: Kod İzleme

```
1 boolean dene(Hesap k, Hesap h, int m)
2     throws InterruptedException {
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 /* çağırın: while (!dene(x, y, 1))
14     Thread.sleep(rastgele); */
```



Adım 3 / 10

DEĞİŞKENLER

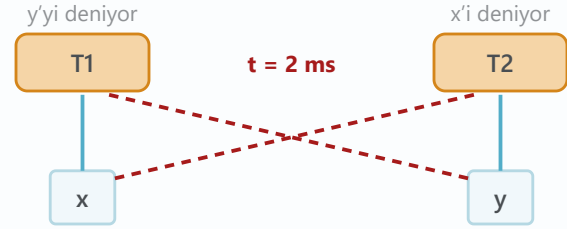
 $t = 1 \text{ ms}$ $x = T1$ $y = T2$ $x.bakiye = 1000$ $y.bakiye = 1000$

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))
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 /* çağırın: while (!dene(x, y, 1))
14     Thread.sleep(rastgele); */
```



x.bakiye = 1000 · y.bakiye = 1000

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

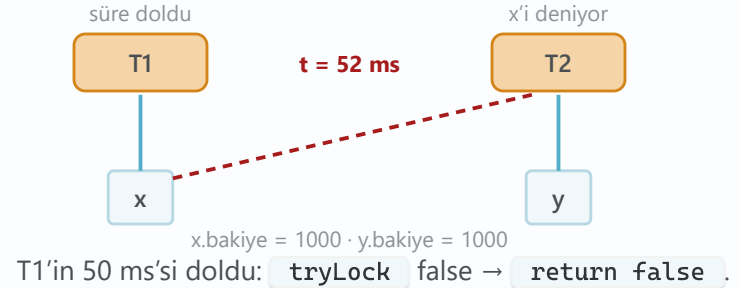
y.bakiye = 1000

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))
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 /* çağırın: while (!dene(x, y, 1))
14     Thread.sleep(rastgele); */
```



Adım 5 / 10

DEĞİŞKENLER

t = 52 ms

x = T1

y = T2

x.bakiye = 1000

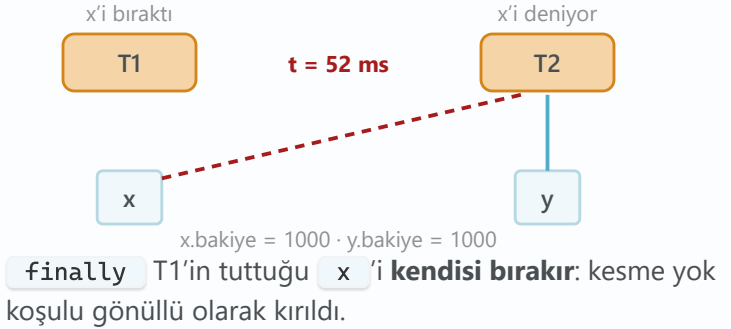
y.bakiye = 1000

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(); }
11    } finally { k.kilit.unlock(); }
12 }
13 /* çağırın: while (!dene(x, y, 1))
14     Thread.sleep(rastgele); */
```



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 {
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 /* çağırın: while (!dene(x, y, 1))
14     Thread.sleep(rastgele); */
```



Adım 7 / 10

DEĞİŞKENLER

t = 52 ms

x = T2

y = T2

x.bakiye = 1000

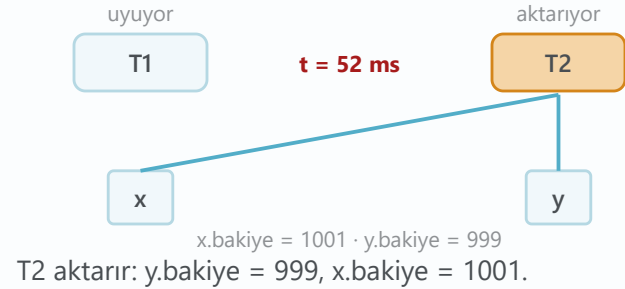
y.bakiye = 1000

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 */
8         ▶ try { k.bakiye -= m; h.bakiye += m;
9             return true; }
10        finally { h.kilit.unlock(); }
11    } finally { k.kilit.unlock(); }
12 }
13 /* çağırın: while (!dene(x, y, 1))
14     Thread.sleep(rastgele); */
```



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; }
10        finally { h.kilit.unlock(); }
11    } finally { k.kilit.unlock(); }
12 }
13 /* çağırın: while (!dene(x, y, 1))
14     Thread.sleep(rastgele); */
```

uyuyor

T1

t = 53 ms

bitti

T2

x

y

x.bakiye = 1001 · y.bakiye = 999

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

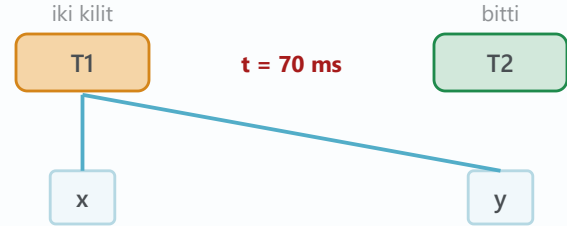
ÇIKTI

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 }
▶ /* çağırın: while (!dene(x, y, 1))
▶     Thread.sleep(rastgele); */
```



$x.bakiye = 1000 \cdot y.bakiye = 1000$

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 sürücü
5	Blu-ray sürücü

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

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



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

Adım 1 / 3

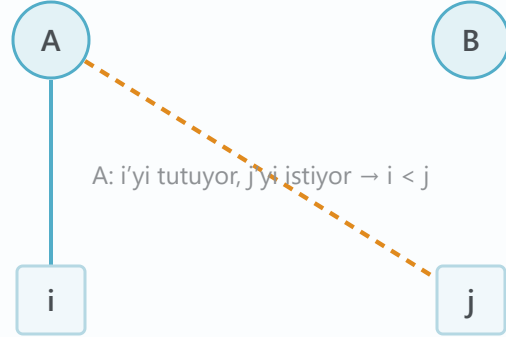
ÖNLEME

Döngüsel Beklemeye Saldırı

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

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

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



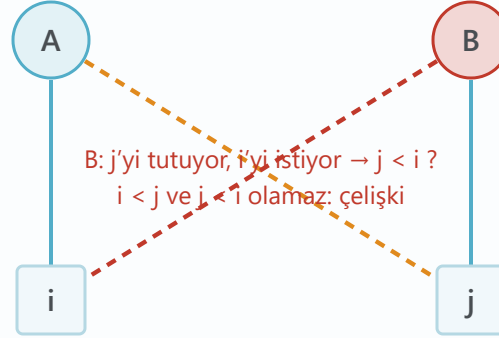
A, i'yi tutup j'yi istiyorsa kurala göre $i < j$.

Adım 2 / 3

Döngüsel Beklemeye Saldırı

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

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.

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 */
```



id 1



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.ilc = ?

T2.ilc = ?

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 {  
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

id 1

T1: ilk = x (1 < 2). x kilidini alır.

T2

y

id 2

Adım 2 / 6

DEĞİŞKENLER

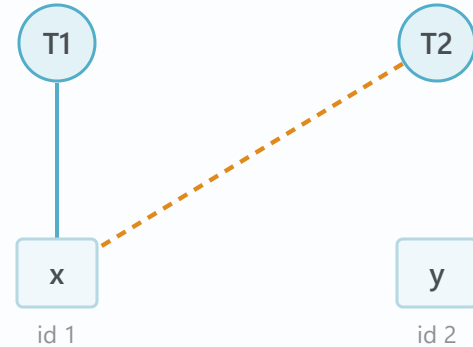
T1.ilk = x

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;  
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 */
```



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

Adım 3 / 6

DEĞİŞKENLER

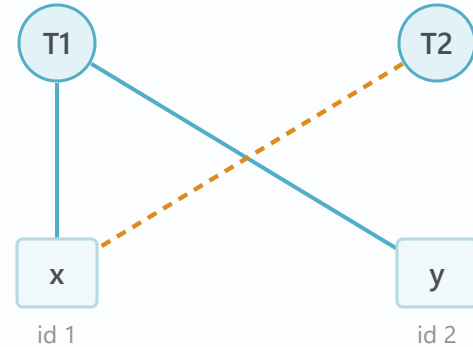
T1.ilc = x

T2.ilc = 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 {  
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 y'yi de alır (kimse tutmuyor) ve aktarımı yapar.

Adım 4 / 6

DEĞİŞKENLER

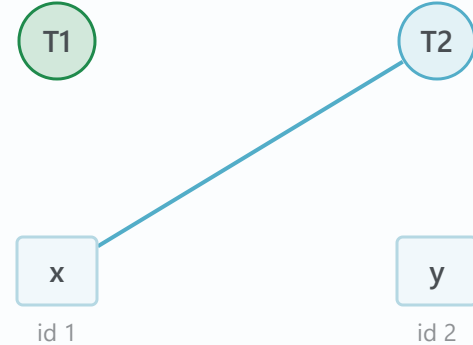
T1.ilc = x

T2.ilc = 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 {  
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 kilitleri bırakır; T2 x'i, sonra y'yi alır. Döngü oluşamaz.

Adım 5 / 6

DEĞİŞKENLER

T1 = **bitti**

T2.ilc = 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 {  
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 */
```



id 1



id 2

Çalıştırıldı (JDK 21): her biri 100 000 ters yönlü aktarım, sonuç `x=1000 y=1000`, kilitlenme yok.

Adım 6 / 6

DEĞİŞKENLER

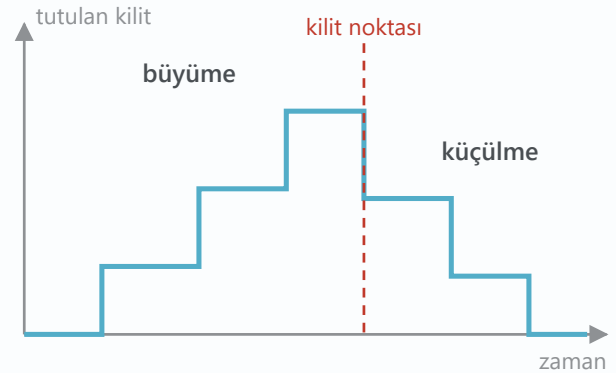
T1 = **bitti**T2 = **bitti**

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

DİĞER SORUNLAR

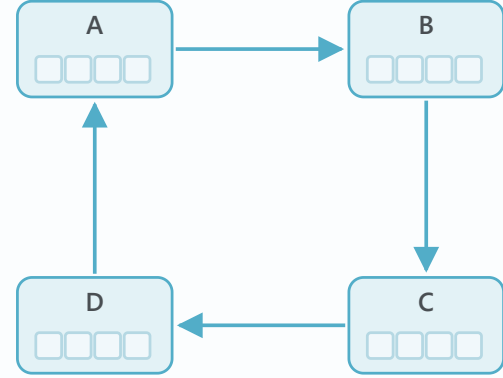
İki Aşamalı Kitleme (2PL)

- 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 kitlemeyi 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**.



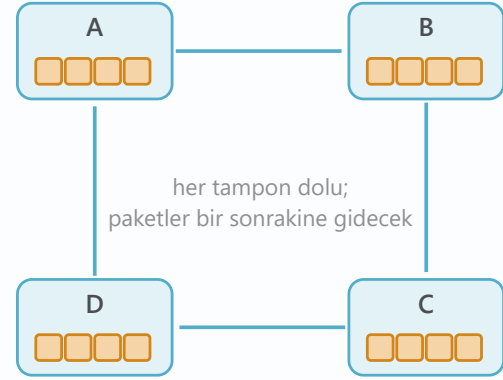
Dört yönlendirici halka biçiminde; her birinin sınırlı tampon belleği var.

Adım 1 / 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**.

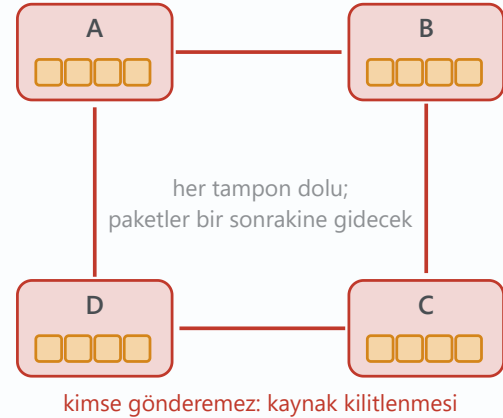


Yoğun trafikte tüm tamponlar, saat yönünde bir sonraki yönlendiriciye gidecek paketlerle dolar.

Adım 2 / 3

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**.



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.

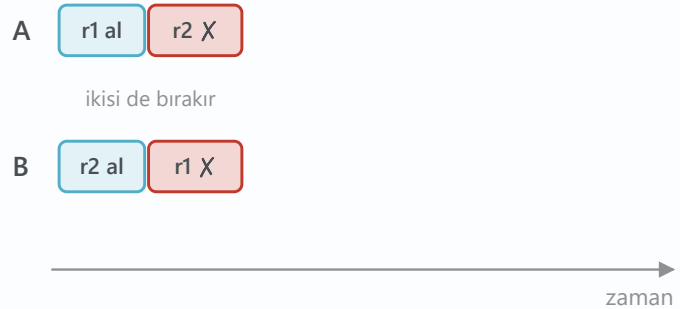
Diğer Sorunlar

Canlı Kilit (Livelock)

- Koridorda karşılaşan iki kişi aynı anda aynı yana kayar: kimse geçemez.
- Kilitlenmede süreçler **uyur**; canlı kilitle **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).

DİĞER SORUNLAR

Canlı Kilit (Livelock)

- Koridorda karşılaşan iki kişi aynı anda aynı yana kayar: kimse geçemez.
- Kilitlenmede süreçler **uyur**; canlı kilitle **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.

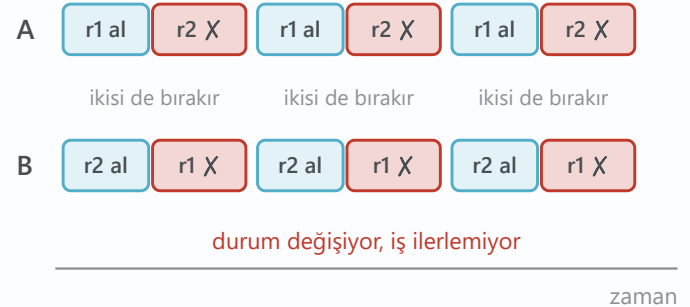
DİĞER SORUNLAR

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**.

DİĞER SORUNLAR

Canlı Kilit (Livelock)

- Koridorda karşılaştan 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, r2 ✓

B

bekle

r2, r1 ✓

rastgele geri çekilme: ritim bozulur

zaman

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

DİĞER SORUNLAR

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**).

yeni küçük işler sürekli geliyor



bekliyor... bekleiyor...

çözüm: FIFO ya da yaşlandırma (aging)

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

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).

Yanılığ: "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 parçasığı iki monitörü (`synchronized` kilidi) **ters sırada** alacak.

Adım 1 / 5

DEĞİŞKENLER

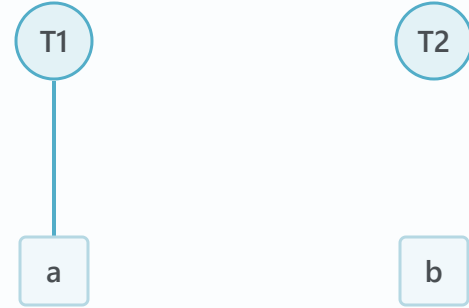
T1 = **RUNNABLE**

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



T1 a'nın monitörüne girer ve uyur.

Adım 2 / 5

DEĞİŞKENLER

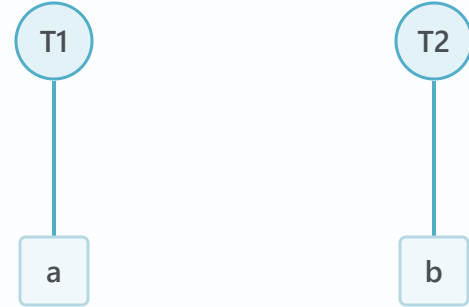
T1 = TIMED_WAITING

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(() -> {
10    ▶ synchronized (b) {
11    ▶     uyu(10);
12
13    synchronized (a) { isle(); }
14 }, "T2");
15 t1.start(); t2.start();
```



Bu sırada T2 b'nin monitörüne girer.

Adım 3 / 5

DEĞİŞKENLER

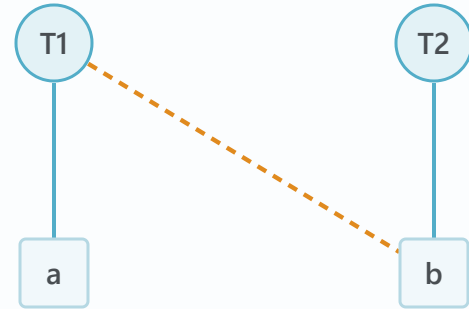
T1 = TIMED_WAITING

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     }
7 }
8 }, "T1");
9 Thread t2 = new Thread(() -> {
10    synchronized (b) { isle(); }
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

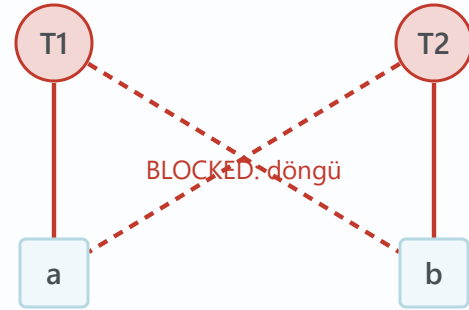
T1 = **BLOCKED**

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 */
6     }
7 }, "T1");
8
9 Thread t2 = new Thread(() -> {
10    synchronized (b) {
11        uyu(10);
12    }
13 }, "T2");
14
15 t1.start(); t2.start();
```



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

Adım 5 / 5

DEĞİŞKENLER

T1 = **BLOCKED**

T2 = **BLOCKED**

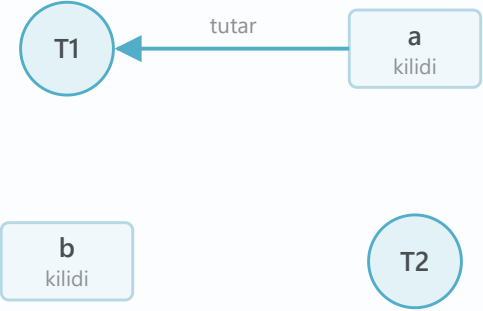
uyu(10) zamanlamayı çok olası kılar ama garanti etmez. **CountDownLatch** ile iki ilk kilit alınana dek bekletilen sürüm JDK 21'de her çalıştırmada kilitlendi.

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. JDK 21'de **Vector.addAll** argümanın **toArray()** 'ini kilitsiz çağırır. Kural: kilit tutarken yabancı yöntem çağırmayın.

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

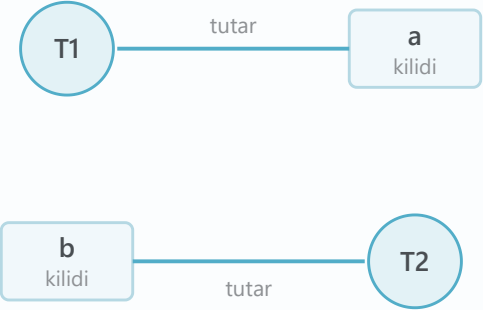
10/10

10/10

Düzeltilmiş sürüm

0/10

0/10



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

Adım 2 / 3

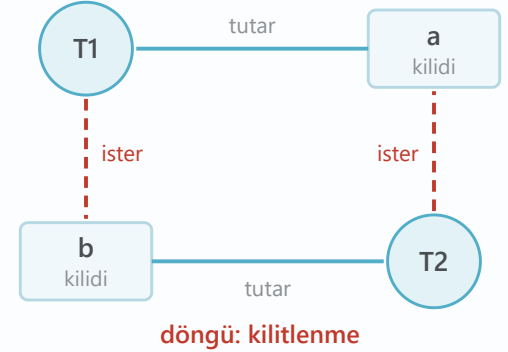
Tablo: 2 iş parçacığı × 200.000 tur, 10 denemede kilitlenen. JDK 21'de **Vector.addAll** argümanın **toArray()** 'ini kilitsiz çağırır. Kural: kilit tutarken yabancı yöntem çağırmayın.

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.

Adım 3 / 3

Tablo: 2 iş parçacığı × 200.000 tur, 10 denemede kilitlenen. JDK 21'de `Vector.addAll` argümanın `toArray()`'ini kilitsiz çağırır. Kural: kilit tutarken yabancı yöntem çağırmayın.

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"

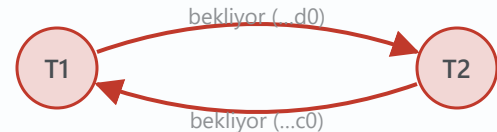
Java stack information for the threads listed above:
"T1":
    at Asili.calis(Asili.java:7)
    - waiting to lock <0x...d0> (a java.lang.Object)
    - locked <0x...c0> (a java.lang.Object)
...
Found 1 deadlock.
```

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

`mmap_lock`

`i_rwsem`

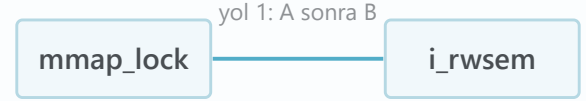
lockdep her kilidi bir **sınıfa** (aynı türdeki tüm örnekler) bağlar.

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



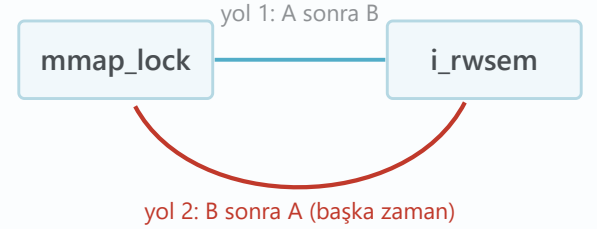
Bir görev A'yı tutarken B'yi alınca "A → B" kenarı kaydedilir.

Adım 2 / 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	<code>lockdep</code> (<code>CONFIG_PROVE_LOCKING</code>): kilit sırası doğrulayıcı; hung task dedektörü
C/C++	ThreadSanitizer: <i>lock-order-inversion</i> ; Valgrind Helgrind
Java	<code>jstack</code> , <code>jcmd</code> , <code>ThreadMXBean</code>
Go	<code>fatal error: all goroutines are asleep - deadlock!</code>
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.

Adım 3 / 3

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

ÖZET

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 $\Leftrightarrow$ kilitlenme.
- Güvensiz durum kilitlenme değildir ama bitiş garantisi yoktur.
- Pratikte en etkili araç: **kilit sıralaması** + zaman aşımli deneme + araçlarla (lockdep, jstack, TSan) denetim.

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 μ s.
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 P_3 'ün isteği $(2\ 1\ 0\ 1)$ olsaydı sonuç ne olurdu?
4. Güvensiz durum ile kilitlenmiş durum arasındaki fark nedir?
5. Silberschatz örneğinde T_1 'in $(1\ 0\ 2)$ isteğinden sonra T_0 $(0\ 2\ 0)$ 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

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](#)