Bölüm 7: Deadlocks Kilitlenmeler

Bölüm 7: Kilitlenmeler

- Deadlock Problemi
- Sistem Modeli
- Deadlock Tarifi
- Deadlock için Çözüm Yöntemleri
- Deadlock Önleme
- Deadlock’tan Kaçınma
- Deadlock Tespiti
- Deadlocktan Kurtulma

Hedefler

- Aynı anda çalışan işlemlerin görevlerini tamamlamasına engel olan kilitlenmelerin açıklanması
- Bir bilgisayar sistemindeki kilitlenmelerin oluşmasını engelleyen veya bu kilitlenmelerden kurtulmayı sağlayan yöntemlerin açıklanması

Deadlock Problemi

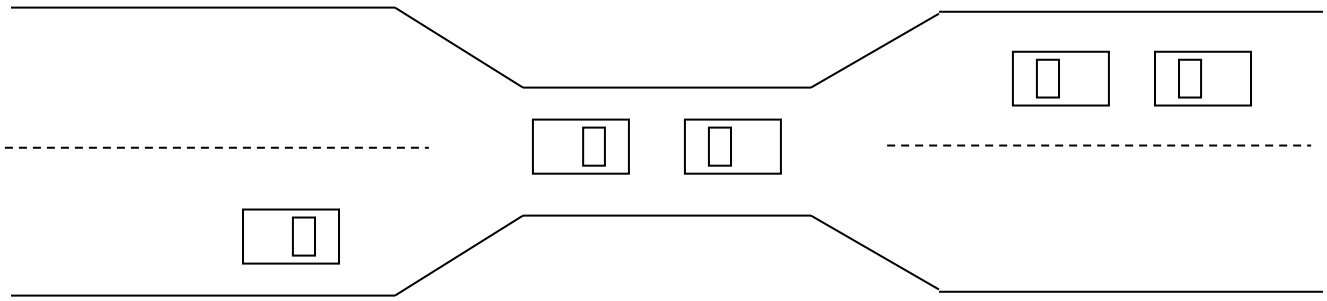
- Her biri bir kaynağı tutan processlerden oluşan bir kümede, aynı işlemlerin bu kümedeki bir başka işleme ait kaynağı elde etmek için beklemesi sonucu bloklanmaları

- Örnek
  - Sistem 2 disk sürücüsüne sahip
  - P_1 ve P_2 işlemlerinin her biri bir diski kullanıyor ve her biri diğer diske ihtiyaç duyuyor

- A ve B semaforları (ilk değerleri 1) ile örnek

P_0	P_1
wait (A);	wait (B);
wait (B);	wait (A);

Köprüden Karşıya Geçme Örneği



- Köprü üzerinde trafik tek yöne akabilir
- Köprü bir kaynak gibi görülebilir
- Eğer kilitlenme olursa, araçlardan biri geriye giderse sorun çözülebilir
- Eğer kilitlenme olursa birden fazla aracın geri çekilmesi gerekebilir
- Açlık (starvation) mümkündür
- Not: Pek çok işletim sistemi kilitlenmelere engel olmaz veya kilitlenmeleri tespit edip çözmeye çalışmaz

Sistem Modeli

- Kaynak (Resource) tipleri $R_1, R_2, \ldots, R_m$

  CPU döngüleri (CPU cycles), hafıza alanı, I/O cihazları

- Her R_i kaynak tipi W_i örneklerine sahiptir

- İşlemler bir kaynağı aşağıdaki şekilde kullanırlar:
  - **istek (request)**
  - **kullanım (use)**
  - **iade etme (release)**

Kilitlenme Tarifi (Deadlock Recipe)

Kilitlenme dört şartın aynı anda sağlanması durumunda ortaya çıkar.

- **Birbirini dışlama (mutual exclusion):** belli bir anda sadece bir işlem, bir kaynağı kullanmalıdır
- **Tutma ve Bekleme (hold and wait):** en az bir kaynağı tutan bir işlem başka işlemler tarafından tutulan ek kaynaklar için beklemelidir
- **Eldekini Bırakmama (no preemption):** bir kaynak ancak o kaynağı tutan işlem, kayangı isteyerek bırakırsa serbest kalmalıdır
- **Dairesel Bekleme (circular wait):** öyle bir bekleyen process kümesi olmalıdır ki $\{P_0, P_1, \ldots, P_n\}$, P_0, P_1 tarafından tutlan bir kaynağı beklemeli, P_1, P_2 tarafından tutlan bir kaynağı beklemeli, …, P_{n-1}, P_n tarafından tutlan bir kaynağı beklemeli ve P_n, P_0 tarafından tutulan bir kaynağı beklemelidir

Kaynak-Tahsis Etme Çizgesi (1/2)

Resource-Allocation Graph

V noktalar kümesi ve E kenarlar kümesi

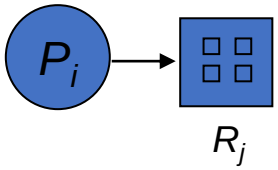
- V iki farklı tipe bölünüyor:
  - $P = \{P_1, P_2, \ldots, P_n\}$, sistemdeki tüm işlemlerin kümesi
  - $R = \{R_1, R_2, \ldots, R_m\}$, sistemdeki tüm kaynakların kümesi
- **İstek kenarı (request edge)** – $P_i \rightarrow R_j$ yönlü kenar
- **Atama kenarı (assignment edge)** – $R_j \rightarrow P_i$ yönlü kenar

Kaynak-Tahsis Çizgesi (2/2)

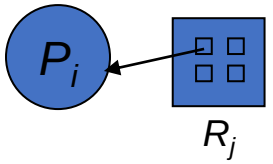
- İşlem

- 4 örneği olan kaynak tipi

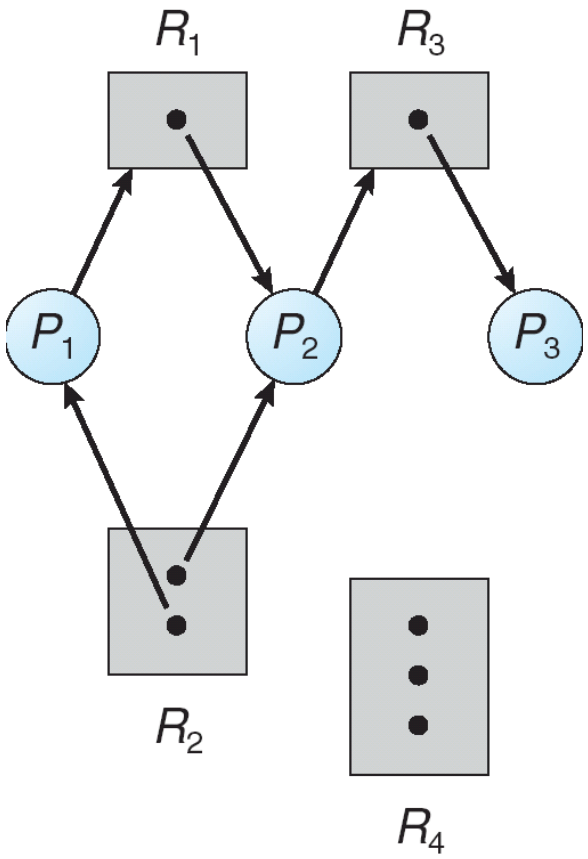
- P_i , R_j örneğini istiyor



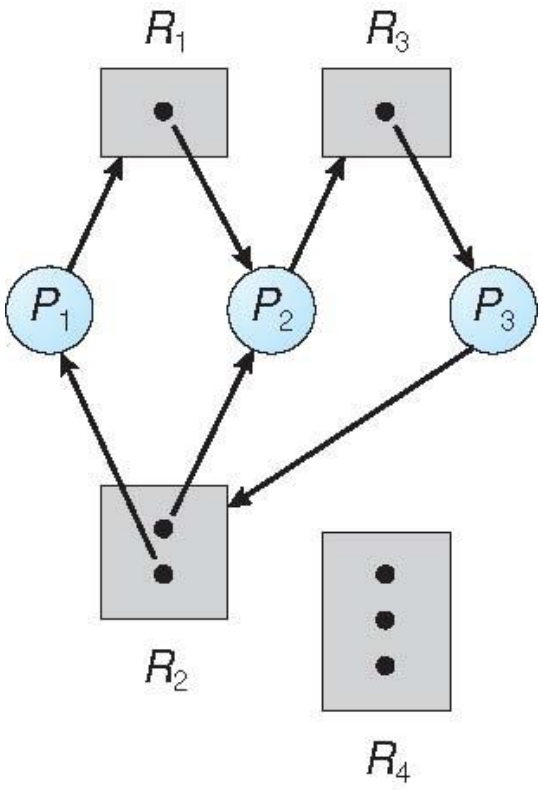
- P_i , R_j örneğini tutuyor



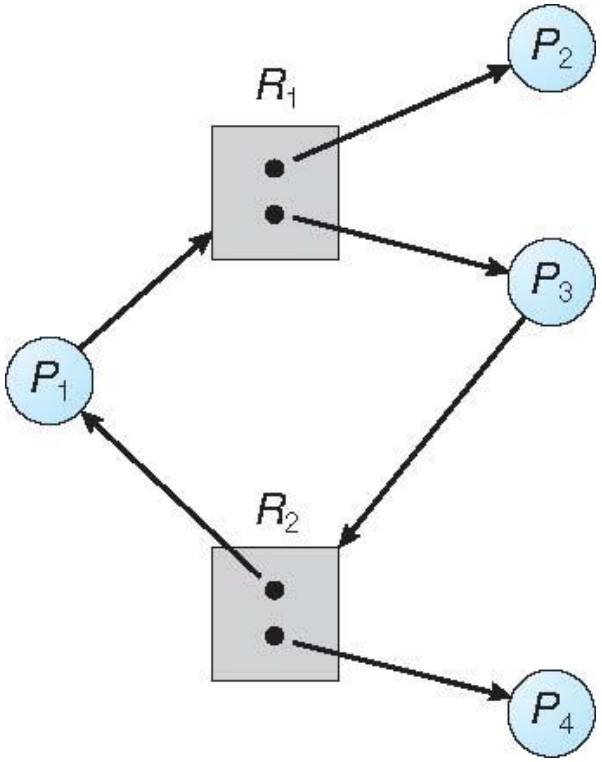
Örnek Kaynak-Ayırım Çizgesi



Kilitlenme İçeren Kaynak-Tahsis Çizgesi



Döngü İçeren Fakat Kilitlenme olmayan Kaynak Tahsisi Çizgesi



Kaynak-Tahsis Çizgesi Temel Bilgileri

- Eğer çizge döngü içermiyorsa $\Rightarrow$ kilitlenme yok
- Eğer çizge döngü içeriyorsa $\Rightarrow$
  - Her bir kaynak tipi için bir örnek varsa, kilitlenme var
  - Her bir kaynak için birden fazla örnek varsa, kilitlenme olma olasılığı var

Java Kilitlenme Örneği (1/2)

```
class A implements Runnable
{
   private Lock first, second;

   public A(Lock first, Lock second) {
      this.first = first;
      this.second = second;
   }

   public void run() {
      try {
         first.lock();
         // do something
            second.lock();
            // do something else
      }
      finally {
         first.unlock();
         second.unlock();
      }
   }
}
```

Thread A

```
class B implements Runnable
{
   private Lock first, second;

   public A(Lock first, Lock second) {
      this.first = first;
      this.second = second;
   }

   public void run() {
      try {
         second.lock();
         // do something
            first.lock();
            // do something else
      }
      finally {
         second.unlock();
         first.unlock();
      }
   }
}
```

Thread B

Java Kilitlenme Örneği (2/2)

```
public static void main(String arg[]) {
   Lock lockX = new ReentrantLock();
   Lock lockY = new ReentrantLock();

   Thread threadA = new Thread(new A(lockX,lockY));
   Thread threadB = new Thread(new B(lockX,lockY));

   threadA.start();
   threadB.start();
}
```

Eğer aşağıdaki senaryo gerçekleşirse kilitlenme mümkün:

threadA -> lockY -> threadB -> lockX -> threadA

Kilitlenmeler İçin Çözüm Yöntemleri

1. **Problemi yok say:** Sistemde kilitlenmelerin hiç bir zaman oluşmayacağını varsay. Pek çok işletim sistemi bu yöntemi kullanır: UNIX ve Java gibi
2. **Kilitlenmeyi Önleme:** Sistemin *asla* kilitilenme durumuna geçmesine izin verme.
3. **Kilitlenmeyi Çözme:** Sistemin kilitlenmesine izin ver. Ardından kilitlenme durumunu algılayıp kilitlenmeyi çöz.

Kilitlenmeyi Önleme (1/3)

- Deadlock Prevention
- Daha önce belirtildiği gibi, kilitlenme olması için dört gerekli şart sağlanmalıdır.
- Bu şartlardan en az birinin sağlanmazsa, kilitlenmenin önüne geçilir.

- **Birbirini dışlama (mutual exclusion)** – paylaşılamaz kaynaklar için şart (örn: yazıcı), paylaşılabilir kaynaklar için gerekli değil (örn: sadece okunabilir dosyalar)
- **Tutma ve Bekleme (hold and wait)** – bir işlem, bir kaynak için istekte bulunduğunda, başka bir kaynağı tutmadığının garantilenmesi gerekir. Örnek bir protokol:
  - İşlemin, ancak sahip olduğu kaynağı iade ettikten sonra yeni kaynak istemesine izin verilebilir
  - Problemler
    - Düşük performanslı kaynak kullanımı
    - Açlığın (starvation) mümkün olması

Kilitlenmeyi Önleme (2/3)

- **Eldekini Bırakmama(no preemption)** –
  - Eğer bazı kaynaklara sahip bir işlem, direk elde edemeyeceği bir kaynağı isterse, tüm sahip olduğu kaynakları elinden alınır ve bekleme moduna alınır
  - Geri alınan kaynaklar, bu kaynakları bekleyen işleme verilir
  - Kaynakları alınan işlem, ancak eski kaynaklarını ve yeni istediği kaynakları alabileceği zaman yeniden başlatılır
  - Bu protokol, genellikle, durumu kolayca kaydedilip eski haline gelebilecek kaynaklar için kullanılır (yazmaçlar ve hafıza alanı gibi)
- **Dairesel Bekleme (circular wait)** – kaynaklara erişimi sıraya koyma

Deadlock Önleme (3/3)

- **Dairesel Bekleme (circular wait)** – kaynaklara erişimi tam anlamıyla sıraya koyarak ve işlemlerin kaynakları bu sıraya uygun şekilde elde etmesini sağlayarak engellenebilir

  F(teyp sürücüsü) = 1

  F(disk sürücüsü) = 5

  F(yazıcı) = 12

- Hem yazıcıyı hem de teyp sürücüsünü elde etmek isteyen işlemci, önce teyp sürücüsünü, ardından yazıcıyı istemelidir

Deadlocktan Kaçınma (1/2)

- Deadlock Avoidance
- Önceki yansılarda anlatılan kilitlenme önleme algoritmaları, kilitlenmenin dört ön şartından en az birinin oluşmasını engelleyerek çalışıyordu
- Bu ise sistemin ve cihazın kullanım performasının düşmesine neden olmaktaydı
- Kilitlenme önlemede alternatif bir yöntem, kaynakların nasıl isteneceğine dair ek **önbilgi** gerektirir
- İhtiyaç duyulan önbilginin miktarına ve tipine göre farklı algoritmalar bulunmaktadır

Deadlock Kaçınma (2/2)

- En basit ve en kullanışlı model, işlemlerin her bir tip kaynaktan en fazla kaç tane ihtiyaç duyabileceğini belirtmesini gerektirir.
- Bu bilgileri elde eden kilitlenme kaçınma algoritması dinamik olarak kaynak atama durumunu inceler ve dairesel-bekleme (circular-wait) durumunun oluşmasına izin vermez.
- Kaynak atama durumu, kullanılabilir ve boştaki kaynak sayısı ile işlemlerin maksimum istekleri ile belirlenir.

Güvenli Durum (Safe State)

- Bir işlem boştaki bir kaynağı istediğinde, sistem kaynağın bu işleme verilmesi durumunun sistemi kilitlenmeye neden olmayacak güvenli durumda bırakıp bırakmayacağına karar vermelidir
- Eğer sistem kaynakları tüm işlemlere kilitlenme olmaksızın belirli bir işlem sırasında sağlayabiliyorsa, sistem güvenli durumdadır
- Sistemdeki <P_1, P_2, …, P_n> işlem sırasını ele alalım
- $j < i$ *iken,* P_i 'nin istediği tüm kaynaklar, boştaki kaynaklar ve tüm P_j işlemlerinin tuttuğu kaynaklar tarafından sağlanıyorsa sistem güvenli durumdadır.
- Bu şart sağlanmazsa sistem güvenilmez (unsafe) durumdadır.
- Her güvenilmez durum kilitlenmeye neden olmak zorunda değildir, ancak kilitlenmeye neden olabilir.

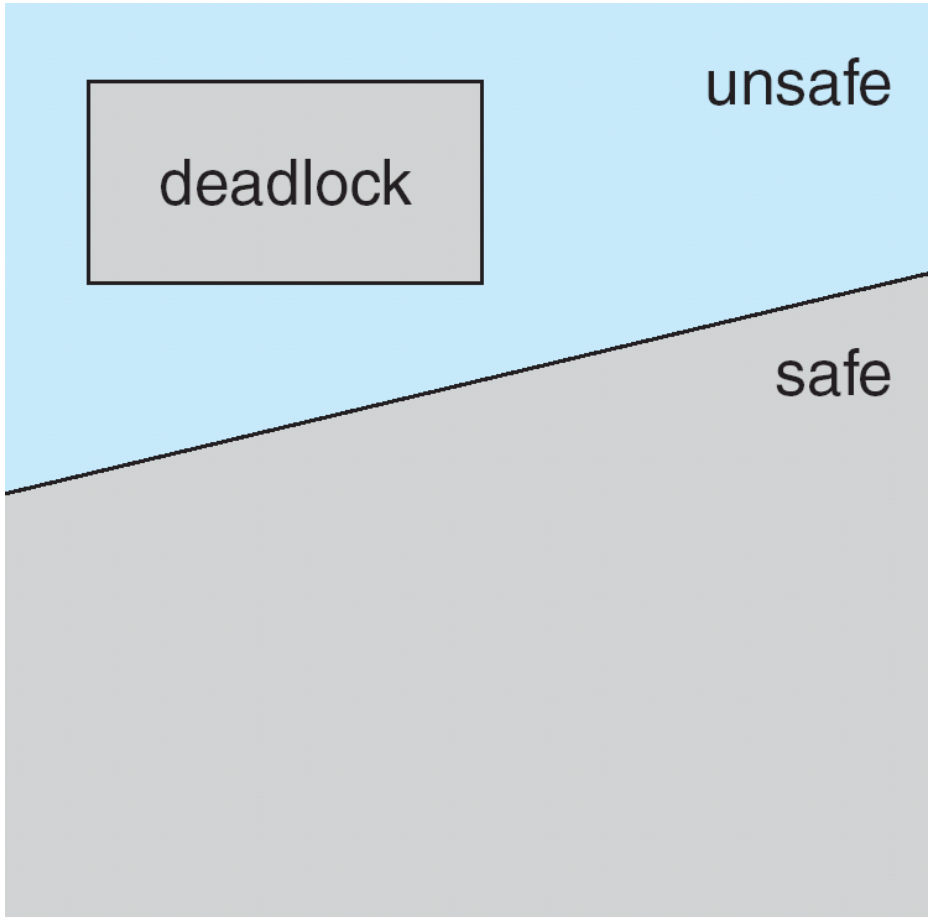
Güvenli Durum (Safe State)

- Genel mantık:
  - Eğer P_i kaynak istekleri o an için karşılanamıyorsa, P_i, P_j işlemlerinin tümünün sonlanmasını bekleyebilir
  - P_j sonlandığında, P_i ihtiyaç duyduğu kaynakları elde edebilir, çalışır ve normal şekilde sonlanır
  - P_j sonlandığında, P_i ihtiyaç duyduğu kaynakları elde edip çalışmaya devam eder
  - ...

Güvenli Durum – Temel Bilgiler

- Eğer sistem güvenli durumdaysa $\Rightarrow$ kilitlenme olmaz
- Eğer sistem güvenilmez durumdaysa $\Rightarrow$ kilitlenme ihtimali vardır
- Kilitlenmeden kaçınma $\Rightarrow$ sistemin hiç bir zaman güvenilmez duruma geçmemesi sağlanarak gerçekleştirilebilir

Safe, Unsafe ve Deadlock Durumları



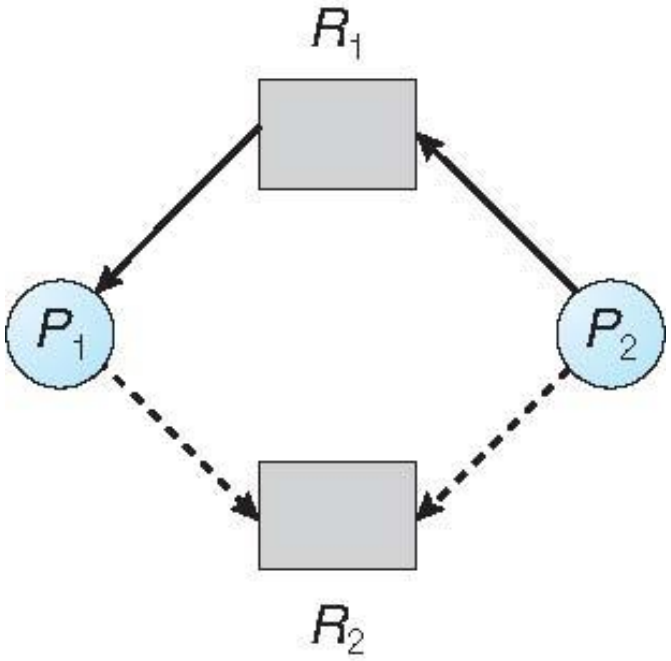
Kilitlenmeden Kaçınma Algoritmaları

- Belirli bir kaynak tipinden bir tane örnek bulunması
  - Kaynak-Tahsis Çizgesi kullanımı
- Belirli bir kaynak tipinden birden fazla örnek bulunması
  - Banker algoritması kullanımı

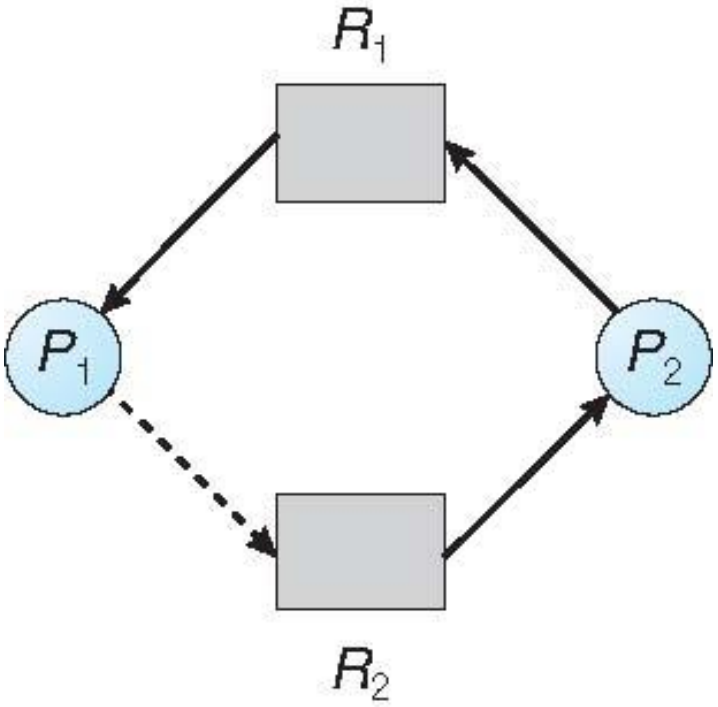
Kaynak-Tahsis Çizgesi Yaklaşımı

- **Niyet kenarı (claim edge)** $P_i \rightarrow R_j$, P_i işleminin ileride bir zamanda R_j kaynağını isteyebileceğini gösteriyor ve çizge üzerinde nokta nokta çizgi (dashed line) şeklinde gösteriliyor
- Niyet kenarı, işlem, kaynağı istediğinde **istek kenarına (request edge)** dönüştürülüyor
- İstek kenarı, kaynak işleme atandığında **atanma kenarına (assignment edge)** dönüştürülüyor
- Bir kaynak işlem tarafından iade edildiğinde, atanma kenarı iddia kenarına dönüştürülüyor
- İşlemler, sistemdeki kaynaklar için önceden niyetlerini bildirmelidir

Kaynak-Ayırım Çizgesi



Kaynak-Ayırım Çizgesi – Güvenilmez Durum



Kaynak-Ayırım Çizgesi Algoritması

- P_i işleminin R_j kaynağını istediğini varsayalım: istek kenarı
- Eğer istek kenarının atanma kenarına dönüşümü, kaynak-ayırım çizgesinde bir döngünün oluşmasına neden olmuyorsa istek karşılanır.
- Aksi halde istek karşılanmaz.

Banker Algoritması

- Belirli bir kaynak tipinden birden fazla örnek bulunması
- Her bir işlem maksimum kullanım tahminini, kullanım niyeti olarak bildiriyor.
- Bir işlem kaynak talebinde bulunduğunda beklemek zorunda kalabilir.
- Bir işlem istediği tüm kaynaklara sahip olduğunda, bu kaynakları sınırlı bir zaman içinde iade etmelidir.

Banker Algoritması için Veri Yapıları

n = işlem sayısı

m = kaynak tipi sayısı

- **Available (uygun)***: m* boyutunda vektör. Eğer Available [*j*] = *k* ise, R_j tipinde *k* örnek kullanıma uygun durumda
- **Max (maksimum)***: n x m* matrisi. Eğer *Max* [*i,j*] = *k* ise, P_i işlemi R_j tipinde kaynaklardan en fazla *k* tanesini isteyebilir
- **Allocation (ayırım)***: n* x *m* matrisi. Eğer Allocation[*i,j*] = *k* ise, P_i işlemi şu anda R_j tipindeki kaynaklardan *k* tanesine sahiptir
- **Need (ihtiyaç)***: n* x *m* matrisi. Eğer *Need*[*i,j*] = *k* ise, P_i işleminin son bulması için R_j kaynak tipinden *k* tanesine daha ihtiyaç duyabilir

Need [*i,j*] = *Max*[*i,j*] – *Allocation* [*i,j*]

Güvenlik (Safety) Algorithm

1. *Work* ve *Finish* vektörleri sırasıyla *m* ve *n* uzunlukta olsun. İlk değer ataması:

   Work = *Available*

   Finish [*i*] = *false, i* = 0, 1, …, *n*- 1

2. Aşağıdaki şartları sağlayan *i* değerini bul:

   (a) *Finish* [*i*] = *false*

   (b) $Need_i \leq Work$

   Böyle bir *i* bulunmazsa, 4. adıma git

3. $Work = Work + Allocation_i$

   Finish[*i*] = *true*

   2. Adıma git

4. Tüm *i* değerleri için *Finish* [*i*] == true ise sistem güvenli durumdadır

P_i İşlemi için Kaynak-İstek Algoritması

Request = P_i işleminin istek vektörü. Eğer $Request_i[j] = k$ ise P_i işlemi R_j kaynak tipinden k tane istiyor

1. Eğer $Request_i \leq Need_i$ ise 2. adıma git. Değilse hata ver çünkü işlem belirlemiş olduğu maksimum istek sayısını aştı

2. Eğer $Request_i \leq Available$ ise, 3. adıma git. Değilse, P_i beklemeli, çünkü yeterli kaynak yok

3. Kaynak-atama durumunu aşağıdaki gibi değiştirerek P_i'nin istediği kaynakları P_i'ye verir gibi yap:

   $Available = Available - Request;$

   $Allocation_i = Allocation_i + Request_i;$

   $Need_i = Need_i - Request_i;$

   - *Eğer güvenli ise* $\Rightarrow$ *kaynaklar* P_i'ye verilir
   - *Güvenli değilse* $\Rightarrow$ P_i beklemelidir ve kaynak-atama durumu eski haline getirilmelidir

Banker Algoritması Örneği

- P_0 ... P_4; 5 işlem

  3 kaynak tipi:

  A (10 örnek), *B* (5 örnek), and *C* (7 örnek)

  T_0 anındaki sistem durumu:

	Allocation	*Max*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Örnek (devamı)

- *Need* matrisinin içeriği (*Max – Allocation)* olarak tanımlanmıştır:

	Need
	A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

- Sistem güvenli durumdadır çünkü < P_1, P_3, P_4, P_2, P_0> sırası güvenlilik kriterini sağlamaktadır

Örnek: P_1 İsteği (1,0,2)

- P_1 , (1,0,2) isteğinde bulunmuş olsun
- Request $\leq$ Available şartının kontrolü: (1,0,2) $\leq$ (3,3,2) $\Rightarrow$ true
- İstek karşılandıktan sonra, kaynak-atama durumunun güncel hali:

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 1	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- Güvenlik çalıştırıldığında < P_1, P_3, P_4, P_0, P_2> sırasının güvenlik şartını sağladığı görülür

Örnek: Diğer İstekler

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 1	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- P_4 ‘ün (3,3,0) isteği karşılanabilir mi?
  - Yeterli kaynak yok
- P_0 ‘ın (0,2,0) isteği karşılanabilir mi?
  - Kaynaklar yeterli fakat yeni sistem durumu güvenli değil

Kilitlenme Tespiti ve Kurtarma

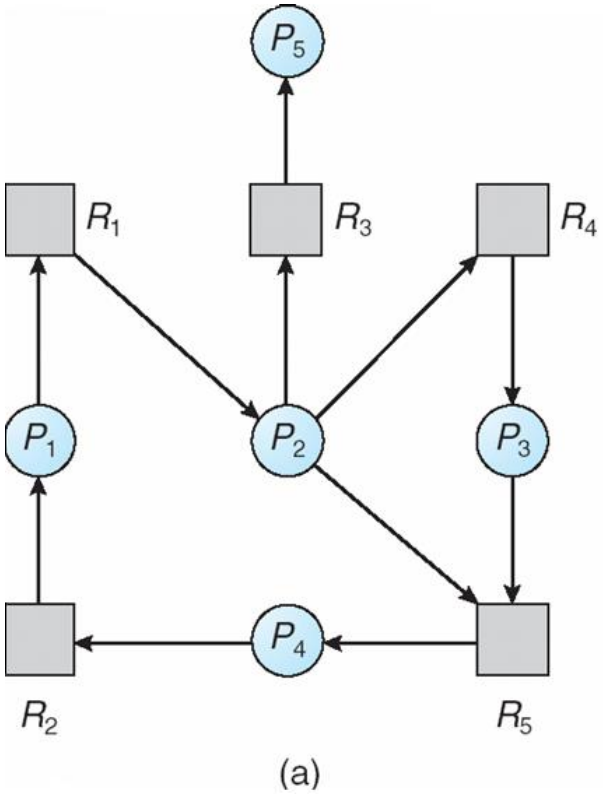
Kitilenmeyi önleme ve kilitlenmeden kaçınma çözümlerini kullanmayan sistemlerde:

- Sistemin kilitlenme durumuna girmesine izin verilir
- Kilitlenme tespit algoritması
- Kilitlenmeden kurtarma mekanizması

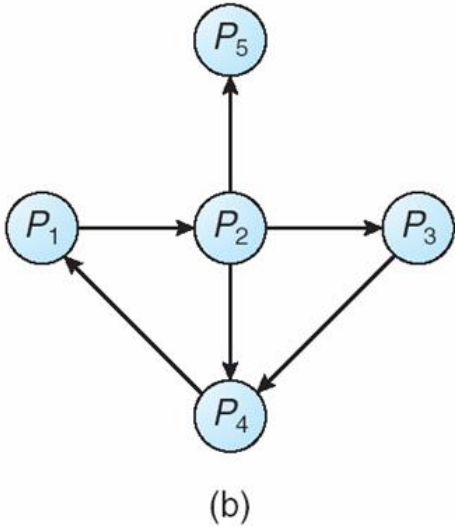
Tüm Kaynak Tipleri için Birer Örnek

- Bekleme çizgesini (*wait-for* graph) düzenli olarak göncelle
  - Çizge nodları işlemlerdir
  - *Eğer* P_i , P_j 'yi bekliyorsa $P_i \rightarrow P_j$
- Belirli zaman aralıklarında, bekleme çizgesinde döngü arayan algoritmayı çalıştır. Eğer döngü varsa, kilitlenme vardır
- Bir çizgede döngü tespit eden algoritma n^2 işlem gerektirir
- Burada *n,* çizgedeki nodların sayısıdır

Kaynak-Ayırım Çizgesi ve Bekleme Çizgesi



Kaynak-Ayırım Çizgesi



Karşılık gelen Bekleme Çizgesi

Kaynak Tiplerinden Birden Fazla Örnek Bulunması

- **Available (uygun)***: m* uzunluğunda vektör. Her bir tip kaynak için kaynaktan kaç tanesinin boşta ve kullanılabilir olduğunu belirtiyor
- **Allocation (ayırım)***: n* x *m* boyutlu matris. Her bir tip kaynağın her bir işlem tarafından kadarının kullanıldığını belirtir.
- **Request (istek)***: n* x *m* boyutlu matris. Her bir işlemin o anki isteklerini gösterir. Eğer *Request* [*i,j*] = *k* ise, P_i işlemi R_j tipinde kaynaktan *k* tane daha istiyor demektir

Kilitlenme Tespit Algoritması

1. *Work* ve *Finish* vektörleri sırasıyla *m* ve *n* uzunluğunda olsun. .
   İlk değer ataması:
   (a) *Work* = *Available*
   (b) *i* = 1,2, …, *n*, eğer $Allocation_i \neq 0$ ise, *Finish*[i] = false; değilse, *Finish*[i] = *true*

2. Aşağıdaki şartları sağlayan *i* indeksini bul:
   (a) *Finish*[*i*] == *false*
   (b) $Request_i \leq Work$

   Eğer bu şartları sağlayan *i* değerini bulamazsan, 4. adıma git

Kilitlenme Tespit Algoritması (Devam)

3. *Work* = *Work* + *Allocation*$_i$
   Finish[*i*] = *true*
   2. adıma git

4. Eğer herhangi bir *i değeri için*, $1 \leq i \leq n$, *Finish*[*i*] == false, şartı sağlanıyorsa sistem kilitlenmiştir. Dahası *Finish*[*i*] == *false* ise, P_i işlemi kilitlenmiştir

Bu algoritma sistemin kilitlenme durumunda olup olmadığını bulmak için O(m x $n^{2)}$ işlem gerektirir

Tespit Algoritması Örneği

- P_0 ... P_4; 5 işlem,
  Üç kaynak tipi: A (7 örnek), *B* (2 örnek), ve *C* (6 örnek)

- T_0 anına sistem durumu:

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- <P_0, P_2, P_3, P_1, P_4> sırası ile çalıştırılırlarsa will tüm *i* değerleri için *Finish*[*i*] = true olur

Tespit Algoritması Örneği (Devam)

- P_2 ‘nin C tipinde ek bir kaynak daha istediğini düşünelim

	Request
	A B C
P_0	0 0 0
P_1	2 0 1
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

- Sistemin durumu?
  - P_0’ın bırakığı kaynaklar geri iade edilir ama diğer işlemlerin isteklerini karşılayacak miktarda kaynak bulunmamaktadır
  - P_1, P_2, P_3, ve P_4 işlemlerinin dahil olduğu bir kilitlenme oluşur

Tespit Algoritması Kullanımı

- Tespit algoritmasının ne zaman ve hangi sıklıkta çalıştırılacağı aşağıdaki faktörlere bağlıdır:
  - Kilitlenme hangi sıklıkta gerçekleşir?
  - Kaç tane işlemin durumu geriye alınmalıdır?
    - Her bir ayrık döngü (disjoint cycle) için bir tane
- Eğer tespit algoritması rastgele çağırılırsa, kaynak çizgesinde pek çok döngü bulunur ve kilitlenmiş pek çok işlem arasında hangi işlemlerin kilitlenmeye neden olduğunu tespit edemeyiz

Kilitlenmeden Kurtarma: İşlemin Sonlandırılması

- Tüm kilitlenen işlemleri sonlandır
- Kilitlenme döngüsü ortadan kalkana kadar, işlemleri teker teker sonlandır
- İşlemleri hangi sırada sonlandırmalıyız?
  - İşlemin önceliğine göre
  - İşlem ne kadar süredir çalışıyor ve sonlanması için ne kadar süreye ihtiyacı var?
  - İşlemin kullandığı kaynaklar?
  - İşlemin tamamlanması için ne kadar kaynağa ihtiyacı var?
  - Kaç tane işlemin sonlandırılması gerekiyor?
  - Işlem interktif mi yoksa sıralı (batch) işlem mi?

Kilitlenmeden Kurtarma: Kaynak İadesi

- Kurban (victim) seçimi – hangi işleme ait hangi kaynaklar geri alınmalıdır
- Geriye sarma (rollback) – Bir işlemden bir kaynağı geri aldığımızda ne yapacağız? Açık bir şekilde işlem çalışmaya normal şekilde çalışmaya devam edemez.
  - İşlemi durduruğ yeniden başlatmak
- Açlık (starvation) – bazı işlemler her zaman kurban olarak seçilebilir
  - geriye alınma sayısı seçimin bir parametresi olarak kullanılabilir