İşletim Sistemleri
Deadlocks

Dr. Öğr. Üyesi Ertan Bütün

Bu dersin içeriği hazırlanırken Operating System Concepts (Silberschatz, Galvin and Gagne) kitabı ve
Prof. Dr. M. Ali Akcayol'un (Gazi Üniversitesi Bilgisayar Mühendisliği Bölümü) ders sunumlarından faydalanılmıştır.

https://codex.cs.yale.edu/avi/os-book/OS9/slide-dir/index.html

http://w3.gazi.edu.tr/~akcayol/BMOS.htm

Konular

- Sistem modeli
- Deadlock tanımı ve özellikleri
- Deadlock yönetimi için metotlar
- Deadlock önleme
- Deadlock’tan kaçınma
- Deadlock algılama
- Deadlock’tan kurtulma

Sistem modeli

- Multiprogramming ortamlarda çok sayıda process/thread sınırlı kaynağı kullanmak için yarışır.
- Bir process bir kaynağa istek yaptığında, kaynak dolu ise bekleme durumuna geçer.
- Bazen bekleyen process bir daha asla durumunu değiştiremez, çünkü talep ettiği kaynaklar **başka kaynakları bekleyen process'ler** tarafından tutulmaktadır. Bu duruma deadlock (kilitlenme) denir.
- İşletim sistemleri genellikle deadlock önleme mekanizmalarını sağlamazlar. Program geliştiricilerin deadlock oluşmayacak şekilde program geliştirmesi gereklidir.

Sistem modeli

- Bir sistemdeki kaynaklar (resources), CPU cycle, dosyalar, I/O cihazları (yazıcı, DVD sürücü) gibi farklı türdedir ($R_1, R_2, \ldots, R_m$).
- Her türün (R_i) bir veya daha fazla örneği (instances, W_i) olabilir.
- Bir process bir kaynağı aşağıdaki sırayla kullanır:
  - Request: Process kaynağa istek yapar, kaynak kullanılabilir değilse bekler.
  - Use: Kaynak kullanılabilir ise process kaynak üzerindeki işlemini gerçekleştirir.
  - Release: Process kaynağı serbest bırakır.

Sistem modeli

- Kaynakların talep edilmesi (request) ve serbest bırakılması (release) sistem çağrıları şeklinde olabilir.
  - Örneğin cihaz için request() ve release(), dosya için open() ve close(), hafıza için allocate() ve free() sistem çağrıları şeklinde olabilir.
- Benzer şekilde request ve release işlemlerinin process senkronizasyonundaki karşılıkları,
  - semaforlar için wait() ve signal(),
  - mutex locks için acquire() ve release() şeklinde olabilir.

Konular

- Sistem modeli
- Deadlock tanımı ve özellikleri
- Deadlock yönetimi için metotlar
- Deadlock önleme
- Deadlock’tan kaçınma
- Deadlock algılama
- Deadlock’tan kurtulma

Deadlock'ın Karakteristiği

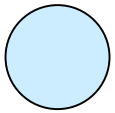
- Bir deadlock durumunda, process'ler hiçbir zaman sonlanamaz, sistem kaynakları atanmış durumdadır ve başka işler başlatılamaz.
- Aşağıdaki **4 durum aynı anda oluştuğunda** deadlock ortaya çıkabilir:
  1. Mutual exclusion: Paylaşılamaz bir modda en az bir kaynak tutulmalıdır; yani bir seferde yalnızca bir process, kaynağı kullanabilir.
  2. Hold and wait: Bir process bir kaynağı tutarken, başka bir kaynağı da bekler durumunda olmalıdır. Beklediği kaynak başka bir process tarafından kullanılır durumda olmalıdır.
  3. No preemption: Kaynaklar zorla process'in elinden alınamaz, yani bir kaynak yalnızca onu tutan process, görevini tamamladıktan sonra gönüllü olarak serbest bırakılabilir.
  4. Circular wait: Birbirlerini dairesel sırada bekleyen bir process kümesi {P0, P1, …, Pn} olmalıdır. P0 process'i P1'i, P1 process'i P2'yi, …, Pn process'i de P0'ı beklemelidir.
- Her durum birbirinden tamamen bağımsız değildir, circular wait oluştuğunda hold and wait durumu da vardır.

Kaynak Tahsisi Grafı (Resource-Allocation Graph)

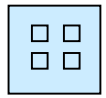
- Deadlock'lar tek yönlü graf (directed graph) kullanılarak (system resource-allocation graph) tanımlanabilir.
- Graf üzerinde düğümler V (vertices) ile kenarlar E (edge) ile gösterilir.
- V kümesi iki kısma ayrılır:
  - P = {P1, P2, …, Pn} aktif process'leri gösterir.
  - R = {R1, R2, …, Rm} sistemdeki tüm kaynakları gösterir.
- Bir Pi process'inden Rj kaynağına çizilen kenar, Pi -> Rj şeklinde gösterilir.
  - Pi -> Rj kenarı ile Pi process'inin Rj kaynağına istek yaptığı ve beklediği ifade edilir.
  - Rj -> Pi kenarı ile de Rj kaynağının Pi process'ine atandığı ifade edilir.
  - Pi -> Rj istek kenarı (request edge), Rj -> Pi atama kenarı (assignment edge) olarak adlandırılır.

Kaynak Tahsisi Grafı

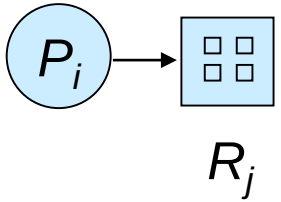
- Graf üzerinde her bir process daire ile, her bir kaynak dikdörtgenle gösterilir.



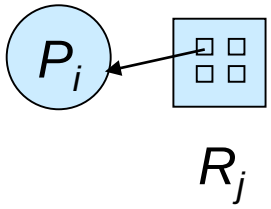
- Bir kaynaktan birden fazla örnek varsa (birden fazla CD WR) dikdörtgen içerisinde her örnek ayrı nokta ile gösterilir.



- Bir process, bir kaynaktan bir örneğe istek yaparsa istek kenarı çizilir.

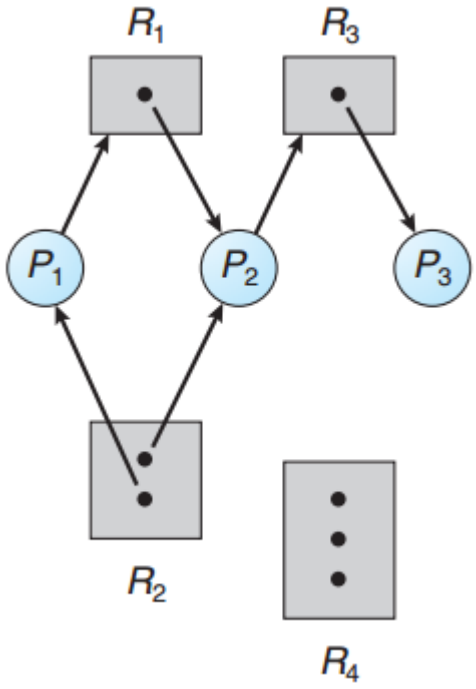


- Bir process'in yaptığı istek karşılanırsa atama kenarına dönüştürülür. Kaynak serbest bırakıldığında ise atama kenarı silinerek kaynak serbest bırakılır.



Kaynak Tahsisi Graf Örneği

- Kaynak türleri
  - R1 ve R3'ten bir, R2'den iki ve R4'ten üç kaynak örneği
- Process durumları
  - P1 process'i, R2 kaynak türünün bir örneğini tutuyor ve R1 kaynak türünün bir örneğini bekliyor.
  - P2 process'i, bir R1 örneğini ve bir R2 örneğini tutuyor ve bir R3 örneğini bekliyor.
  - P3 process'i, bir R3 örneğini tutuyor.
- Bir kaynak tahsis grafının tanımı göz önüne alındığında, eğer grafta kenarlar(edge) döngü oluşturmuyorsa, sistemdeki hiçbir process'in deadlock olmadığını gösterir.
- Graf bir döngü içeriyorsa, bir deadlock olabilir.



Deadlock İçeren Bir Kaynak Tahsisi Grafiği

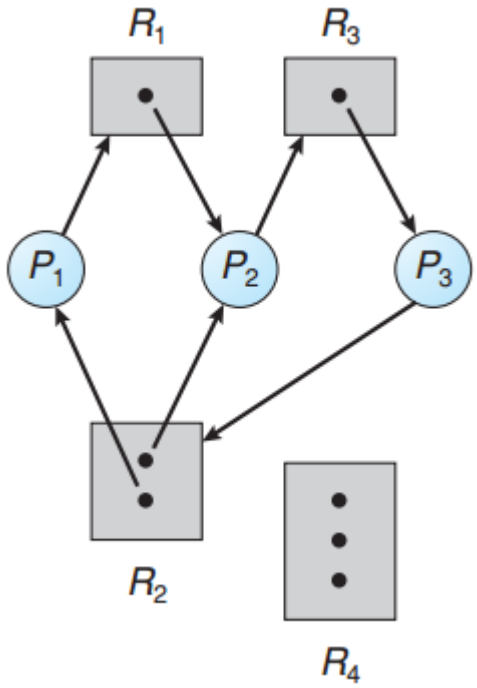
- Döngü yalnızca her biri tek bir örneğe sahip olan bir dizi kaynak türü içeriyorsa:
  - Bir deadlock oluşmuştur. Döngüde yer alan her process deadlock olmuştur.
  - Bu durumda, grafikteki bir döngü, deadlock'ın varlığı için hem gerekli hem de yeterli bir koşuldur.
- Her kaynak türünün birkaç örneği varsa:
  - Bir döngünün mutlaka bir deadlock oluşturacağı anlamına gelmez.
  - Bu durumda graftaki bir döngü, deadlock'ın varlığı için gerekli ancak yeterli bir koşul değildir.

Deadlock İçeren Bir Kaynak Tahsisi Grafiği

- Bunu bir örnekle göstermek için önceki graf örneğinde P3, R2den bir kaynak örneği istesin.
- Bu örnekte sistemde en az iki döngü vardır:

$$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$$
$$P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$$

- P1, P2 ve P3 process'leri deadlock olmuştur.
  - P2, P3 tarafından tutulan R3 kaynağını bekliyor.
  - P3, P1 veya P2'nin R2 kaynağını serbest bırakmasını bekliyor.
  - Ayrıca, P1, P2'nin R1 kaynağını serbest bırakmasını bekliyor.

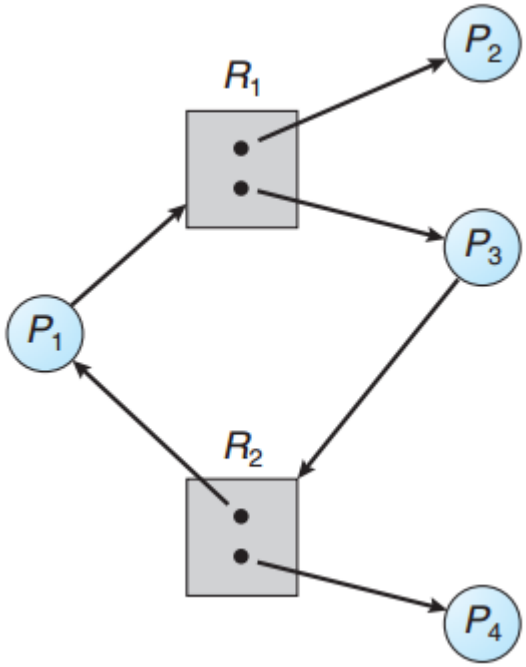


Döngü İçeren Ancak Deadlock İçermeyen Bir Graf

- Buradaki örnekte bir döngü vardır:

$$P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$$

  - Ancak deadlock yoktur,
  - P4, R2 türünden kullandığı kaynağı serbest bırakırsa P3 kullanabilir, döngü kırılabilir.



Temel Bilgiler

- Eğer grafta döngü yoksa $\Rightarrow$ deadlock yoktur.
- Eğer grafta döngü varsa
  - Her kaynak türünün sadece bir örneği varsa deadlock olur.
  - Her kaynak türünün birkaç örneği varsa deadlock olma ihtimali vardır.

Konular

- Sistem modeli
- Deadlock tanımı ve özellikleri
- Deadlock yönetimi için metotlar
- Deadlock önleme
- Deadlock'tan kaçınma
- Deadlock algılama
- Deadlock'tan kurtulma

Deadlock Yönetimi İçin Metotlar

- Deadlock problemi için 3 farklı yol izlenebilir:
  1. Deadlock'lardan kaçınmak veya deadlock'ları önlemek için protokol kullanılabilir. Bu durumda sistem hiçbir zaman deadlock durumuna düşmez.
  2. Sistemin deadlock durumuna düşmesine izin verilir, deadlock algılanır ve çözülür.
  3. Deadlock problemi tamamen gözardı edilir ve sistemde deadlock hiçbir zaman olmayacak gibi davranılır.
- Üçüncü durum işletim sistemleri tarafından yaygın kullanılır (Linux, Windows). Linux ve Windows işletim sistemleri, deadlock yönetimini uygulama geliştiricilere bırakır.

Deadlock Yönetimi İçin Metotlar

- Bir sistemde hiçbir zaman deadlock olmamasını garanti etmek için, deadlock-prevention veya deadlock-avoidance yöntemleri kullanılabilir.
- Deadlock prevention, kaynak isteklerini sınırlandırarak deadlock oluşmasını önler.
- Deadlock avoidance, bir kaynağa istek yapan process'in yanı sıra, hangi zaman aralığında kullanacağını da bilmek ister.
- Deadlock avoidance, kaynak isteği yapan process'in beklemesine veya kaynağın atanmasına karar verebilir.
- Bir sistem, deadlock prevention veya deadlock avoidance yöntemlerini kullanmazsa deadlock oluşabilir.
  - Bu sistemler, deadlock olup olmadığını kontrol eden bir algoritma ve deadlock oluştuğunda çözümünü sağlayan bir algoritma sağlamalıdır.

Deadlock Yönetimi İçin Metotlar

- Deadlock'ları tespit etmek ve ortadan kaldırmak için algoritmalar olmadığında, sistemin deadlock bir durumda olduğu ancak ne olduğunun fark edilemediği bir duruma ulaşabilir.
  - Bu durumda, tespit edilmeyen deadlock, sistemin performansının düşmesine neden olur.
  - Çünkü kaynaklar çalıştırılamayan process'ler tarafından tutulmaktadır ve daha fazla process kaynaklar için talepte bulundukça deadlock durumuna girecektir.
  - Sonunda, sistem çalışmayı durduracak ve manuel olarak yeniden başlatılması gerekecektir.
- Deadlock algılama ve çözümleme yöntemleri çoğu işletim sisteminde kullanılmaz. Çünkü bu ekstra maliyet gerektirir.
- Bazı sistemler, başka durumlar için kullandığı yöntemleri deadlock yönetiminde de kullanırlar.

Konular

- Sistem modeli
- Deadlock tanımı ve özellikleri
- Deadlock yönetimi için metotlar
- Deadlock önleme
- Deadlock'tan kaçınma
- Deadlock algılama
- Deadlock'tan kurtulma

Deadlock Önleme (Deadlock Prevention)

- Deadlock oluşması için 4 durumun (mutual exclusion, hold and wait, nopreemption, circular wait) gerçekleşmesi gereklidir.
- Bu koşullardan en az birinin gerçekleşmeyeceğinden emin olunarak bir deadlock oluşmasını önlenebilir. Şimdi bu yaklaşım doğrultusunda dört koşul ayrı ayrı incelenecektir.
- Mutual exclusion – en azından bir kaynak paylaşılamaz (eş zamanlı erişilemez) durumdadır.
  - Paylaşılabilir kaynaklar deadlock oluşturmaz.
  - Örneğin Read-only dosyalar paylaşılabilir kaynaklardır ve deadlock oluşturmaz.
  - Ancak genel olarak, mutual exclusion koşulu yadsınarak deadlock'lar önlemez, çünkü bazı kaynaklar özünde paylaşılamazdır. Örneğin, bir mutex lock birkaç process tarafından aynı anda paylaşılamaz.

Hold and Wait - Deadlock Önleme

- Bir sistemde hold and wait durumunun oluşmaması için, bir process bir kaynağa istek yaptığında başka bir kaynağı tutmaması gerekir. Bunun için iki ayrı protokol çözümü:

1. Her process'in yürütülmeye başlamadan önce **tüm kaynaklarını** talep ettiği ve bunların process'e tahsis edildiği bir protokol kullanılabilir.
   - Bu, process için kaynak isteyen sistem çağrılarının diğer tüm sistem çağrılarından önce olmasını zorunlu kılarak sağlanabilir.
2. Başka bir protokolde ise, eğer bir process **kaynak kullanmıyorsa** yeni kaynak için istek yapabilir.
   - Bir process birden çok kaynağı talep edebilir ve bunları kullanabilir. Herhangi bir ek kaynak talep etmeden önce, halihazırda kendisine tahsis edilmiş tüm kaynakları serbest bırakması gerekir.

Hold and Wait - Deadlock Önleme

- Bu iki protokol arasındaki farkı göstermek için, verileri bir DVD sürücüsünden diskteki bir dosyaya kopyalayan, bu kopyalanan verileri dosyada sıralayan ve ardından sonuçları bir yazıcıda yazdıran örnek bir process olsun.
- Birinci protokol için process'in başlangıçta DVD sürücüsünü, disk dosyasını ve yazıcıyı istemesi gerekir.
  - Yazıcıya yalnızca sonunda ihtiyaç duysa bile, tüm yürütme süresi boyunca yazıcıyı tutacaktır. **(verimsiz kullanım)**
- İkinci protokolde ise process'in başlangıçta yalnızca DVD sürücüsünü ve disk dosyasını istemesine izin verilir. DVD sürücüsünden verileri diske kopyalar ve ardından hem DVD sürücüsünü hem de disk dosyasını serbest bırakır.
  - Process daha sonra disk dosyasını ve yazıcıyı istemelidir. Disk dosyasını yazıcıya kopyaladıktan sonra, bu iki kaynağı serbest bırakır ve sona erer.
  - **Starvation mümkündür**. Birkaç popüler kaynağa ihtiyaç duyan **bir process, süresiz olarak beklemek zorunda kalabilir**, çünkü ihtiyaç duyduğu kaynaklardan en az biri her zaman başka bir process'te olabilir.

No Preemption - Deadlock Önleme

- Deadlock için gerekli üçüncü koşul, önceden tahsis edilmiş olan kaynakların preempted olmamasıdır, yani bir process bir kaynağı kullanırken henüz işini bitirmeden bu kaynak, başka bir process'e tahsis edilemez (preempted olamaz).
- Bu koşulun geçerli olmadığından emin olmak için iki farklı protokol kullanabilir.
- Birinci protokol:
  - Bir process bazı kaynakları tutuyorsa ve kendisine hemen tahsis edilemeyen başka bir kaynak talep ederse process'in beklemesi gerekir, bu durumda process'in elinde tuttuğu tüm kaynaklar preempted yapılır, başka bir deyişle bu kaynaklar dolaylı olarak serbest bırakılır.
  - Bırakılan kaynaklar, process'in beklediği kaynaklar listesine eklenir.
  - Process, yalnızca eski kaynaklarının yanı sıra talep ettiği yenilerini de geri kazanabildiğinde yeniden başlatılacaktır.

No Preemption - Deadlock Önleme

- İkinci protokol:
  - Bir process bir kaynağa istek yaptığında kullanılabilir olup olmadığı kontrol edilir. Uygunsa tahsis edilir.
  - İstek yapılan kaynak başka process tarafından tutuluyorsa, tutan process'in başka bir kaynağı bekleyip beklemediği kontrol edilir.
    - Kaynağı tutan process başka bir kaynağı bekliyorsa, istek yapılan kaynak alınır (preempt) yeni istek yapan process'e atanır.
    - Kaynağı tutan process başka bir kaynağı beklemiyorsa, istek yapan process bekletilir.

Circular Wait - Deadlock Önleme

- Circular wait durumunun önlenmesi için aşağıdaki protokol uygulanabilir:
- Her kaynak türüne farklı bir tam sayı atanarak tüm kaynak türleri sıralanır,
  - Bu, iki kaynağı karşılaştırmamızı ve sıralamamızda birinin diğerinden önce gelip gelmediğini belirlememizi sağlar.
- Bir process kaynak isteğini ancak artan sırada yapabilir.
  - Bir process, başlangıçta bir kaynağı isteyebilir. Ardından yapacağı istekler artan sırada olmak zorundadır.
  - Bir processe *Ri* kaynağı tahsis edildikten sonra bu process ancak *F(Rj) > F(Ri)* (*F*, kaynağın unique numarasını döndüren fonksiyondur) koşuluna uyan *Rj* kaynak türlerini isteyebilir.
    - Alternatif bir protokol olarak, *Ri* kaynak türünün bir örneğini talep eden bir process'in, *F (Rj) ≥ F (Ri)* olacak şekilde herhangi bir *Rj* kaynağını serbest bırakmasını isteyebiliriz.

Circular Wait - Deadlock Önleme

- Bu iki protokol kullanılırsa, circular wait koşulu gerçekleşemez. Bunun ispatı şu şekildedir:
- Circular wait'e dahil olan process'ler kümesi {P0, P1, ..., Pn} olsun, burada Pi, Pi+1 process'i tarafından tutulan bir Ri kaynağını beklemektedir. (Dairesel olduğundan Pn de P0 tarafından tutulan bir Rn kaynağını beklemektedir).
- *Pi+1* process'i, *Ri+1* kaynağını talep ederken *Ri* kaynağını tuttuğundan, tüm *i* için *F(Ri) < F(Ri+1)* olmalıdır.
  - Ancak bu durum şu anlama gelir: *F(R0) < F(R1) <... < F(Rn) < F(R0)*
  - Bu protokole göre *F(Rn) < F(R0)* olamayacağı için circular wait koşulu gerçekleşemez.

Konular

- Sistem modeli
- Deadlock tanımı ve özellikleri
- Deadlock yönetimi için metotlar
- Deadlock önleme
- Deadlock’tan kaçınma
- Deadlock algılama
- Deadlock’tan kurtulma

Deadlock'tan Kaçınma (Deadlock Avoidance)

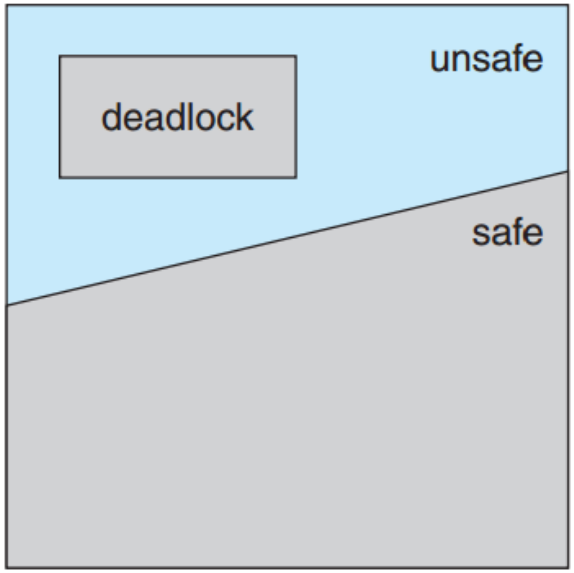
- Deadlock önleme algoritmaları, isteklerin nasıl yapılacağını sınırlayarak deadlock'ları önler. Sınırlar, deadlock için gerekli koşullardan en az birinin oluşmasını engeller.
  - Bununla birlikte, bu şekilde deadlock'ları önlemek cihazlardan az faydalanılmasına ve sistem throughput'unun düşmesine neden olur.
- Diğer bir yöntem olan Deadlock'tan Kaçınma yönteminde, deadlock oluşumunu engellemek için kaynakların nasıl istendiğinin bilinmesi gerekir. Her process, ihtiyaç duyabileceği her türden maksimum kaynak sayısını bildirir.
- Bu ön bilgi göz önüne alınarak circular wait koşulunun asla var olamayacağından emin olmak için kaynak tahsis durumu dinamik olarak incelenir.

Safe State - Deadlock'tan Kaçınma

- Eğer bir sistem, kaynakları process'lere belirli bir sırada (safe sequence) maksimum ihtiyaçları kadar atayabiliyorsa ve deadlock oluşmuyorsa bu durum safe state olarak adlandırılır.
- Eğer bir sistemde safe sequence varsa ancak sistem safe state olur.
- Bir dizi <P1, P2, …, Pn> process ancak aşağıdaki durumları sağlıyorsa mevcut kaynak atama durumu safe sequence olur:
  - Pi process'inin tüm kaynak istekleri mevcut boş kaynaklarla ve tüm Pj'ler (j<i) tarafından tutulan dolu kaynaklarla karşılanabiliyorsa,
  - Bu durumda, Pi'nin ihtiyaç duyduğu kaynaklar hemen mevcut değilse, Pi tüm Pj'lerin tamamlanmasını bekler.
  - Tüm Pj'ler tamamlandığında Pi, ihtiyaç duyduğu tüm kaynakları elde edebilir, belirlenen görevi tamamlayabilir, tahsis edilen kaynaklarını iade edebilir ve sonlanabilir.
  - Pi sona erdiğinde, Pi+1 gerekli kaynakları elde edebilir ve bu böyle devam eder. Böyle bir sıra yoksa, sistem durumunun unsafe olduğu söylenir.

Temel Bilgiler- Deadlock'tan Kaçınma

- Eğer bir sistem safe state ise ⇒ deadlock yoktur
- Eğer bir sistem unsafe state ise ⇒ deadlock olma ihtimali vardır.
- Deadlock'tan Kaçınma ⇒ Bir sistemin asla unsafe state'e girmeyeceğinden emin olur.
  - Safe state olduğu sürece, işletim sistemi unsafe state ve deadlock durumlardan kaçınabilir. Unsafe state bir durumda işletim sistemi, process'lerin bir deadlock meydana gelecek şekilde kaynakları istemesini engelleyemez.



Kaynak Tahsisi Graf Algoritması- Deadlock'tan Kaçınma

- Her kaynak türünün **yalnızca bir örneğini içeren** bir kaynak tahsis sistemi varsa, deadlock'ı önlemek için bu bölümde tanımlanan **kaynak tahsisi grafının** bir formunu kullanabilir.
- Daha önce açıklanan istek ve atama kenarlarına ek olarak, niyet kenarı (claim edge) adı verilen yeni bir kenar türü eklenir.
- Pi -> Rj niyet kenarı, Pi process'inin Rj kaynağını ileriki zamanda isteyebileceğini gösterir. Niyet kenarı graf üzerinde noktalı çizgiyle gösterilir.
- Pi process'i Rj kaynağını istediğinde niyet kenarı istek kenarına dönüştürülür. Rj kaynağı Pi process'i tarafından serbest bırakıldığında, atama kenarı niyet kenarına dönüştürülür.
- Process'ler, sistemdeki kaynaklar için önceden niyetlerini bildirmelidir. Yani Pi process'i çalışmaya başlamadan önce tüm niyet kenarları graf üzerinde zaten görüntülenmelidir.

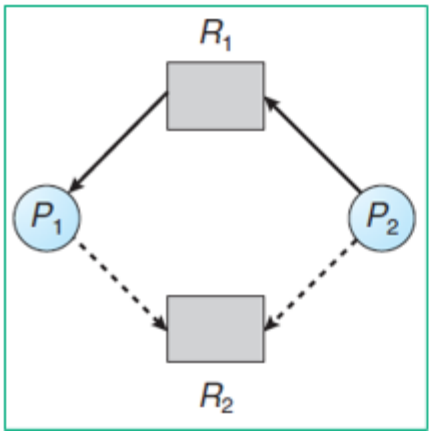
Kaynak Tahsisi Graf Algoritması- Deadlock'tan Kaçınma

- Pi process'i Rj kaynağını istesin. Bu isteğe yalnızca Pi → Rj istek kenarının bir Rj → Pi atama kenarına dönüştürülmesi, kaynak tahsis grafında bir döngü oluşturmuyorsa izin verilebilir.
- Döngü oluşmuyorsa, kaynağın tahsisi sistemi güvenli bir durumda bırakacaktır.
- Bir döngü bulunursa, kaynağın tahsisi sistemi güvenli olmayan bir duruma sokar. Bu durumda, Pi process'i, isteklerinin karşılanması için beklemek zorunda kalacaktır.

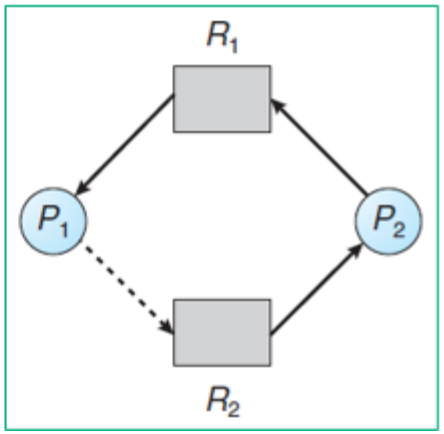
Kaynak Tahsisi Graf Algoritması- Deadlock'tan Kaçınma

- Şekil1'de P2'nin R2 istediğini varsayalım. R2 şu anda boşta olmasına rağmen, bu eylem grafta ileride bir döngü oluşturabileceğinden (Şekil 2) P2'ye tahsis edilmez.
- Bir döngü, sistemin unsafe state olduğunu gösterir. P1 R2'yi isterse ve P2, R1'i isterse, bir deadlock meydana gelir.

Şekil1



Şekil2



(güvenli olmayan durum, unsafe state)

Banker Algoritması- Deadlock'tan Kaçınma

- Kaynak Tahsisi (Resource-allocation) algoritması aynı kaynaktan **birden fazla** olan sistemlerde uygulanamaz.
- Aynı kaynaktan birden fazla olan sistemlerde banker algoritması (banker's algorithm) kullanılabilir.
- Banka sisteminde, bir banka parasını hiçbir zaman tüm müşterilerinin ihtiyaçlarını karşılayamayacak şekilde tahsis etmez.
- Sisteme yeni bir process girdiğinde, ihtiyaç duyabileceği her kaynak türünün maksimum örnek sayısını bildirmesi gerekir. Bu sayı, sistemdeki toplam kaynak sayısını geçemez.
- Bir kullanıcı bir dizi kaynak talep ettiğinde, sistem bu kaynakların tahsis etmenin sistemi safe state'te bırakıp bırakmayacağını belirlemelidir.
  - Safe state'te bırakacaksa kaynaklar tahsis edilir, aksi takdirde process başka bir process'in yeterli kaynakları serbest bırakmasını beklemelidir.

Banker Algoritması- Deadlock'tan Kaçınma

- Banker algoritması aşağıdaki veri yapılarını kullanır:
  (**n** process sayısını, **m** kaynak türü sayısını gösterir)
  - Available: **m** uzunluğunda bir vektördür. Her bir kaynak türü için kaç adet boşta kaynak olduğunu tutar. **Available(j) = k** ise, sistemdeki Rj kaynağı türünden **k** adet boştadır.
  - Max: **n * m** matristir. Her process'in her kaynak türü için maksimum kaynak talebini tutar. **Max[i][j] = k** ise Pi process'i Rj kaynak türünden en fazla **k** tane isteyebilir.

	Alloc	Max	Need	Avail
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	**7 4 3**	3 3 2
P_1	2 0 0	3 2 2	**1 2 2**	
P_2	3 0 2	9 0 2	**6 0 0**	
P_3	2 1 1	2 2 2	**0 1 1**	
P_4	0 0 2	4 3 3	**4 3 1**	

Banker Algoritması- Deadlock'tan Kaçınma

- **(n** process sayısını, **m** kaynak türü sayısını gösterir)
  - Allocation: **n * m** matristir. Her process'in her kaynak türü için kullanmakta olduğu kaynak miktarını tutar. **Allocation[i][j] = k** ise, Pi process'i Rj kaynak türünden **k** adet kullanmaktadır.
  - Need: **n * m** matristir. Her process'in kalan kaynak ihtiyacını gösterir. **Need[i][j] = k** ise Pi process'i Rj kaynağından **k** adet daha kullanabilir. **Need[i][j] = Max[i][j] – Allocation[i][j]** olur.

	Alloc	Max	Need	Avail
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	7 4 3	3 3 2
P_1	2 0 0	3 2 2	1 2 2	
P_2	3 0 2	9 0 2	6 0 0	
P_3	2 1 1	2 2 2	0 1 1	
P_4	0 0 2	4 3 3	4 3 1	

Banker Algoritması için **Güvenlik Algoritması (Safety Algorithm)**

- Güvenlik algoritması bir sistemin güvenli durumda olup olmadığını bulmak için kullanılır. Algoritmanın adımları:

1. ***Work*** ve ***Finish*** sırasıyla **m** (kaynak türü sayısı) ve **n** (process sayısı) uzunluğunda vektörler olsun ve başlangıç değerleri şöyle olsun:

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

2. Aşağıdaki iki koşulu sağlayan bir ***i*** process'i bul

   a) Finish* [*i*] = *false
   b) $\boldsymbol{Need_i \leq Work}$ ***// i***'nin ihtiyaç sayısı boşta olanlardan küçük mü
   Böyle bir i process'i varsa 3. adıma git
   Böyle bir i process'i yoksa 4. adıma git

3. ***Work = Work +*** $\boldsymbol{Allocation_i}$ ***// i***'nin kullandıkları boş olanlara ilave edilir
   Finish*[*i*] = *true
   2. adıma git

4. ***Finish* [*i*] == *true*** tüm ***i***'ler için sağlanıyorsa sistem güvenli durumdadır.

Pi Process'i için **Kaynak İstek (Resource-Request) Algoritması**

- Kaynak İstek Algoritması ise isteklerin onaylamanın güvenli durumu bozup bozmayacağını anlamak için kullanılıyor.
- P_i process'i için istek vektörü $Request_i$ olsun. $Request_i[j] = k$ ise, P_i process'i, R_j kaynak türünün k tane örneğini ister. P_i tarafından kaynak talebi yapıldığında algoritmanın adımları:
  1. $Request_i \leq Need_i$ ise 2. adıma git, değilse maksimum talebi aştığı için hata ver
  2. $Request_i \leq Available$ ise 3. adıma git, değilse kaynaklar boş olmadığından P_i beklemelidir.
  3. Durumu aşağıdaki gibi değiştirerek istenen kaynakları P_i'ye **tahsis ediyormuş gibi** yapın:

     $Available = Available - Request_i;$
     $Allocation_i = Allocation_i + Request_i;$
     $Need_i = Need_i - Request_i;$

     Durum **güvenli ise** P_i'ye istediği kaynaklar tahsis edilir, değilse P_i, $Request_i$ için beklemelidir, kaynak tahsisi durumu eski haline döndürülür

Banker Algoritması- Örnek

- Sistemde P_0 dan P_4 e, 5 tane process

  Üç kaynak türü:

  A (10 örnek), B (5 örnek), ve C (7 örnek)

- T_0 anında sistem görüntüsü aşağıdadır.

	Alloc	**Max**	**Need**	**Avail**
	A B C	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		

Banker Algoritması- Örnek

- ***Need*** matris içeriği ***Max – Allocation*** ile bulunur:

	Alloc	**Max**	**Need**	**Work**
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	**7 4 3**	3 3 2
P_1	2 0 0	3 2 2	**1 2 2**	
P_2	3 0 2	9 0 2	**6 0 0**	
P_3	2 1 1	2 2 2	**0 1 1**	
P_4	0 0 2	4 3 3	**4 3 1**	

Banker Algoritması- Örnek

- **Güvenlik Algoritması (Safety Algorithm)** ile sistemin güvenli olup olmadığına bakılıyor.

	Alloc	Max	Need	Work
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	7 4 3	3 3 2
P_1	2 0 0	3 2 2	1 2 2	
P_2	3 0 2	9 0 2	6 0 0	
P_3	2 1 1	2 2 2	0 1 1	
P_4	0 0 2	4 3 3	4 3 1	

P_0'ın ihtiyacı şu an karşılanamıyor.

Finish[0]=false
Finish[1]=false
Finish[2]=false
Finish[3]=false
Finish[4]=false

Banker Algoritması- Örnek

	Alloc	Max	Need	Work
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	**7 4 3**	3 3 2
P_1	2 0 0	3 2 2	**1 2 2**	
P_2	3 0 2	9 0 2	**6 0 0**	
P_3	2 1 1	2 2 2	**0 1 1**	
P_4	0 0 2	4 3 3	**4 3 1**	

P_1'in ihtiyacı şu an karşılanabiliyor

safe sequence <P_1,

Finish[0]=false
Finish[1]=true
Finish[2]=false
Finish[3]=false
Finish[4]=false

Banker Algoritması- Örnek

	Alloc	Max	Need	Work
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	**7 4 3**	5 3 2
P_1	0 0 0	0 0 0	0 0 0	
P_2	3 0 2	9 0 2	**6 0 0**	
P_3	2 1 1	2 2 2	**0 1 1**	
P_4	0 0 2	4 3 3	**4 3 1**	

Work güncellenir

Banker Algoritması- Örnek

	Alloc	Max	Need	Work
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	**7 4 3**	5 3 2
P_1	0 0 0	0 0 0	0 0 0	
P_2	3 0 2	9 0 2	**6 0 0**	
P_3	2 1 1	2 2 2	**0 1 1**	
P_4	0 0 2	4 3 3	**4 3 1**	

P_2'ın ihtiyacı şu an karşılanamıyor.

Banker Algoritması- Örnek

	Alloc	Max	Need	Work
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	**7 4 3**	5 3 2
P_1	0 0 0	0 0 0	0 0 0	
P_2	3 0 2	9 0 2	**6 0 0**	
P_3	2 1 1	2 2 2	**0 1 1**	
P_4	0 0 2	4 3 3	**4 3 1**	

safe sequence <P_1, P_3,

P_3'ün ihtiyacı şu an karşılanabiliyor.

Finish[0]=false
Finish[1]=true
Finish[2]=false
Finish[3]=true
Finish[4]=false

Banker Algoritması- Örnek

	Alloc	Max	Need	Work
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	**7 4 3**	7 4 3
P_1	0 0 0	0 0 0	0 0 0	
P_2	3 0 2	9 0 2	**6 0 0**	
P_3	0 0 0	0 0 0	0 0 0	
P_4	0 0 2	4 3 3	**4 3 1**	

Work güncellenir

Banker Algoritması- Örnek

	Alloc	Max	Need	Work
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	**7 4 3**	7 4 3
P_1	0 0 0	0 0 0	0 0 0	
P_2	3 0 2	9 0 2	**6 0 0**	
P_3	0 0 0	0 0 0	0 0 0	
P_4	0 0 2	4 3 3	**4 3 1**	

Finish[0]=false
Finish[1]=true
Finish[2]=false
Finish[3]=true
Finish[4]=true

safe sequence <P_1, P_3, P_4,

P_4'ün ihtiyacı şu an karşılanabiliyor.

Banker Algoritması- Örnek

	Alloc	Max	Need	Work
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	**7 4 3**	7 4 5
P_1	0 0 0	0 0 0	0 0 0	
P_2	3 0 2	9 0 2	**6 0 0**	
P_3	0 0 0	0 0 0	0 0 0	
P_4	0 0 0	0 0 0	0 0 0	

Work güncellenir

Banker Algoritması- Örnek

	Alloc	Max	Need	Work
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	7 4 3	7 4 5
P_1	0 0 0	0 0 0	0 0 0	
P_2	3 0 2	9 0 2	6 0 0	
P_3	0 0 0	0 0 0	0 0 0	
P_4	0 0 0	0 0 0	0 0 0	

P_1'in ihtiyacı şu an karşılanabiliyor.

safe sequence $<P_1, P_3, P_4, P_0,$

Finish[0]=true
Finish[1]=true
Finish[2]=false
Finish[3]=true
Finish[4]=true

Banker Algoritması- Örnek

	Alloc	Max	Need	Work
	A B C	A B C	A B C	A B C
P_0	0 0 0	0 0 0	0 0 0	7 5 5
P_1	0 0 0	0 0 0	0 0 0	
P_2	3 0 2	9 0 2	**6 0 0**	
P_3	0 0 0	0 0 0	0 0 0	
P_4	0 0 0	0 0 0	0 0 0	

Work güncellenir

Banker Algoritması- Örnek

	Alloc	Max	Need	Work
	A B C	A B C	A B C	A B C
P_0	0 0 0	0 0 0	0 0 0	7 5 5
P_1	0 0 0	0 0 0	0 0 0	
P_2	3 0 2	9 0 2	**6 0 0**	
P_3	0 0 0	0 0 0	0 0 0	
P_4	0 0 0	0 0 0	0 0 0	

P_2'ın ihtiyacı şu an karşılanabiliyor.

safe sequence <P_1, P_3, P_4, P_0, P_2>

Finish[0]=true
Finish[1]=true
Finish[2]=true
Finish[3]=true
Finish[4]=true

Banker Algoritması- Örnek

- safe sequence <P_1, P_3, P_4, P_0, P_2>
- ***Finish* [*i*] == *true*** tüm ***i***'ler için sağlandığından sistem güvenli durumdadır.

	Alloc	**Max**	**Need**	**Work**
	A B C	A B C	A B C	A B C
P_0	0 0 0	0 0 0	0 0 0	10 5 7
P_1	0 0 0	0 0 0	0 0 0	
P_2	0 0 0	0 0 0	0 0 0	
P_3	0 0 0	0 0 0	0 0 0	
P_4	0 0 0	0 0 0	0 0 0	

Finish[0]=true
Finish[1]=true
Finish[2]=true
Finish[3]=true
Finish[4]=true

Banker Algoritması- Örnek

- P_1 process'i ***Request*$_1$ = (1, 0, 2)** isteğinde bulunursa, bu isteğin onaylanıp onaylanmayacağına karar vermek için Kaynak İstek (Resource-Request) Algoritmasını 1., 2. ve 3. adımlarını uygulayın ve safe sequence'i bulmaya çalışın, safe sequence ortaya çıkıyorsa ***Request*$_1$** isteğini yerine getirin ve yeni durumun tablosunu gösterin.

Başlangıçta

	Allocation	*Max*	*Available*
	A B C	*A B C*	*A B C*
T_0	0 1 0	7 5 3	3 3 2
T_1	2 0 0	3 2 2	
T_2	3 0 2	9 0 2	
T_3	2 1 1	2 2 2	
T_4	0 0 2	4 3 3	

Banker Algoritması- Örnek

- P_1 process'i ***Request*$_1$ = (1, 0, 2)** isteğinde bulunursa, bu isteğin onaylanıp onaylanmayacağına karar vermek için Kaynak İstek (Resource-Request) Algoritması kullanılır:

  ***1. Request*$_1$ ≤ *Need*$_1$ (*Max*$_1$ - *Allocation*$_1$)** ⇒ *(1,0,2) ≤ (1,2,2) = true*

  2. Request*$_1$ ≤ *Avaliable ⇒ (1, 0, 2) ≤ (3, 3, 2) = true

- 1. ve 2. kontrolün sonucu true'dur, yani sistemde bu talebe cevap verilebilecek yeteri kadar müsait kaynak mevcuttur.

- Daha sonra bu talebin yerine getirildiği varsayıldığında oluşacak yeni duruma bakılır:

Başlangıçta

	Allocation	*Max*	*Available*
	A B C	*A B C*	*A B C*
T_0	0 1 0	7 5 3	3 3 2
T_1	2 0 0	3 2 2	
T_2	3 0 2	9 0 2	
T_3	2 1 1	2 2 2	
T_4	0 0 2	4 3 3	

***Request*$_1$ talebi yerine getirilirse**

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
T_0	0 1 0	7 4 3	2 3 0
T_1	3 0 2	0 2 0	
T_2	3 0 2	6 0 0	
T_3	2 1 1	0 1 1	
T_4	0 0 2	4 3 1	

Banker Algoritması- Örnek

- Bu yeni durumunun güvenli olup olmadığına bakılır. Bunun için güvenlik algoritması çalıştırılarak <P1, P3, P4, P0, P2> safe sequence elde edilerek sistemin güvenli olduğu belirlenir.
  - Bu yüzden P_1 process'inin isteği **derhal kabul edilebilir**.

Banker Algoritması- Örnek

- P4, ***Request*$_2$ = (3, 3, 0)** isteği için Kaynak İstek (Resource-Request) Algoritması uygulayın ve bu isteğe nasıl cevap verilir, isteğe olumlu cevap verilirse yeni durumun tablosunu oluşturun.

***Request*$_1$ talebi yerine getirildi.**

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
T_0	0 1 0	7 4 3	2 3 0
T_1	3 0 2	0 2 0	
T_2	3 0 2	6 0 0	
T_3	2 1 1	0 1 1	
T_4	0 0 2	4 3 1	

Banker Algoritması- Örnek

- P4, ***Request*$_2$ = (3, 3, 0)** isteği için Kaynak İstek (Resource-Request) Algoritmasının 2. adımını sağlamıyor o yüzden talep reddedilir.

  ***1. Request*$_2$ ≤ *Need*$_1$ (*Max*$_1$ - *Allocation*$_1$)** ⇒ *(3,3,0) ≤ (4,3,1) = true*

  2. Request*$_2$ ≤ *Avaliable ⇒ (3, 3, 0) ≤ (2, 3, 0) = false

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
T_0	0 1 0	7 4 3	2 3 0
T_1	3 0 2	0 2 0	
T_2	3 0 2	6 0 0	
T_3	2 1 1	0 1 1	
T_4	0 0 2	4 3 1	

Banker Algoritması- Örnek

- P0, ***Request*$_3$ = (0, 2, 0)** isteği için Kaynak İstek (Resource-Request) Algoritması uygulayın ve bu isteğe nasıl cevap verilir, isteğe olumlu cevap verilirse yeni durumun tablosunu oluşturun.

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
T_0	0 1 0	7 4 3	2 3 0
T_1	3 0 2	0 2 0	
T_2	3 0 2	6 0 0	
T_3	2 1 1	0 1 1	
T_4	0 0 2	4 3 1	

Banker Algoritması- Örnek

- P0, ***Request*$_3$ = (0, 2, 0)** isteğinde bulunduğunda kaynaklar kullanılabilir olsa bile, Kaynak İstek (Resource-Request) Algoritmasının uygulandığında ortaya çıkacak durum unsafe olur, dolayısıyla kabul edilemez

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
T_0	0 1 0	7 4 3	2 3 0
T_1	3 0 2	0 2 0	
T_2	3 0 2	6 0 0	
T_3	2 1 1	0 1 1	
T_4	0 0 2	4 3 1	

Konular

- Sistem modeli
- Deadlock tanımı ve özellikleri
- Deadlock yönetimi için metotlar
- Deadlock önleme
- Deadlock’tan kaçınma
- Deadlock algılama
- Deadlock’tan kurtulma

Deadlock Algılama

- Bir sistem, bir deadlock önleme ya da bir deadlock'tan kaçınma algoritması kullanmazsa bir deadlock durumu ortaya çıkabilir.
- Bu tür bir sistem, deadlock probleminin çözümü için aşağıdakileri sağlayabilir:
  - Sistemin deadlock durumunda olup olmadığına karar verecek bir algoritma
  - Deadlock durumunun çözülmesi için bir algoritma
- Deadlock algılama algoritmaları, her kaynaktan bir tane olması veya her kaynaktan birden fazla olması durumuna göre farklılık gösterir.

Deadlock Algılama

Her kaynaktan bir örnek olması

- Tüm kaynakların tek bir örneği varsa, o zaman kaynak tahsisi (Resource-allocation) grafının bir formunu kullanan wait-for grafı adı verilen bir deadlock algılama algoritması tanımlanabilir.
  - Wait-for grafında, kaynak düğümleri kaldırılarak sadece process düğümleri bırakılır.
  - P_i **den** P_j ye doğru bir kenar ($P_i \rightarrow P_j$), P_i process'inin P_j process'ini beklediğini gösterir.
- Wait-for grafında döngü varsa deadlock vardır.
  - Deadlock tespiti için, wait-for grafında bir döngü arayan bir algoritmanın periyodik olarak çalıştırılması gerekir.

Deadlock Algılama

Her kaynaktan bir örnek olması

- Şekilde bir kaynak tahsisi grafı ve buna karşılık gelen wait-for grafı verilmiştir.

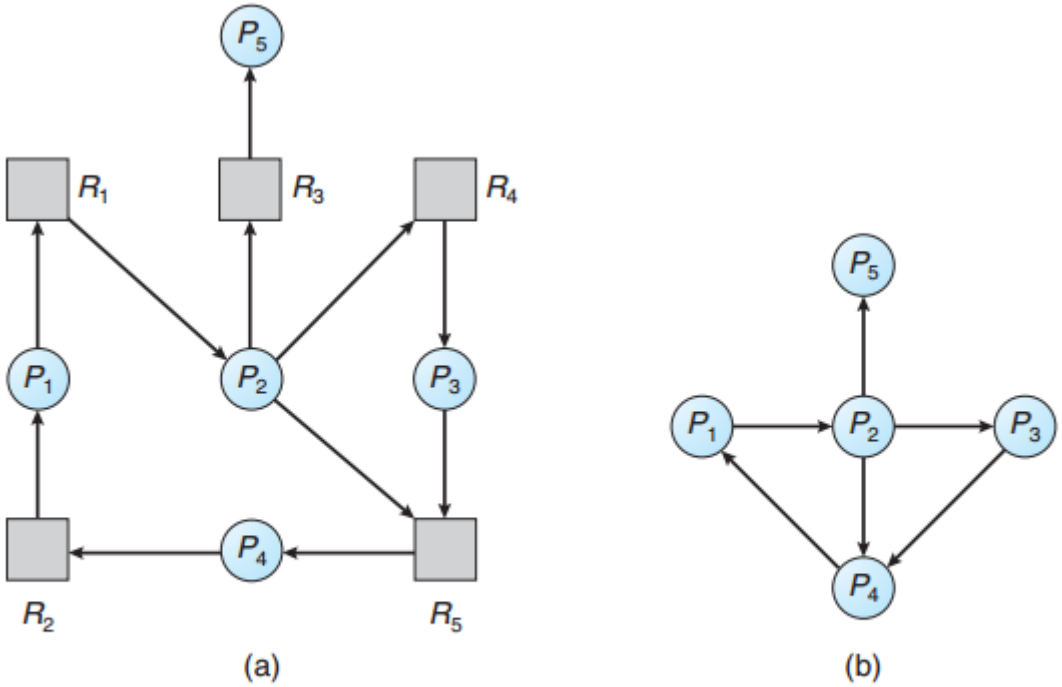


Figure 7.9 (a) Resource-allocation graph. (b) Corresponding wait-for graph.

Deadlock Algılama

Her kaynaktan birden fazla örnek olması

- Bu durumda kullanılacak algoritma için banker algoritmasına benzer birkaç veri yapısı kullanır:
  - **Available**: Her kaynak türü için boştaki kaynak sayısını tutan ***m*** uzunluğunda bir vektördür.
  - **Allocation**: ***n*** **x** ***m*** matristir. Her process için mevcut atanmış kaynak sayısını tutar.
  - **Request**: ***n*** **x** ***m*** matristir, her process'in mevcut istek sayısını tutar.
    - ***Request[i][j] = k*** ise P_i process'i R_j kaynağından ***k*** tane daha istemektedir.

Deadlock Algılama

Her kaynaktan birden fazla örnek olması durumunda algoritmanın adımları:

1. ***Work*** ve ***Finish*** sırasıyla m (kaynak türü sayısı) ve n (process sayısı) uzunluğunda vektörler olsun ve şöyle başlangıç değerleri şöyle olsun:

   Work = Available
   i* = 0, 1, …, *n*- 1 için** **Allocation$_i$ ≠ 0** **ise *Finish* [*i*] = *false
   değilse Finish* [*i*] = *true

2. Aşağıdaki iki koşulu sağlayan bir ***i*** process'i bul

   a) Finish* [*i*] = *false
   b) Request$_i$ ≤ Work // i'nin ihtiyaç sayısı boşta olanlardan küçük mü
   Böyle bir ***i*** process'i varsa 3. adıma git
   Böyle bir ***i*** process'i yoksa 4. adıma git

3. ***Work = Work + Allocation$_i$ // i'***nin kullandıkları boş olanlara ilave edilir
   Finish*[*i*] = *true
   2. adıma git

4. ***Finish*[*i*] == *false*** bazı ***i***'ler için sağlanıyorsa sistem deadlock durumundadır.

Deadlock Algılama- Örnek

Her kaynaktan birden fazla örnek olması

- Sistemde P_0 dan P_4 e, 5 tane process

  Üç kaynak türü: ***A*** (7 örnek), ***B*** (2 örnek), ve ***C*** (6 örnek) olsun.
- T_0 anında sistem görüntüsü şekilde erilmiştir.
- T_0 anında sistemde deadlock var mıdır?

	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	

Deadlock Algılama- Örnek

Her kaynaktan birden fazla örnek olması

- Sistemde P_0 dan P_4 e, 5 tane process

  Üç kaynak türü: ***A*** (7 örnek), ***B*** (2 örnek), ve ***C*** (6 örnek) olsun.

- T_0 anında sistem görüntüsü şekilde erilmiştir.

- T_0 anında sistemde deadlock yoktur. Çünkü algoritma yürütüldüğünde tüm ***i*** ler için ***Finish[i] == true*** değerini veren $< P_0, P_2, P_3, P_4, P_1 >$ safe sequence elde edilir.

	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	

Deadlock Algılama- Örnek

- T_0 anında sistem görüntüsü şekilde erilmiştir, T_1 anında ***P_2, (0, 0, 1)*** isteğini yaparsa sistemde deadlock oluşur mu?
  - Tabloyu yeni isteğe göre güncelleyin ve Deadlock Algılama algoritmasını çalıştırın.

	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	

Deadlock Algılama- Örnek

- ***P_2, (0, 0, 1)*** isteğini yaparsa ***Request*** matrisi şekildeki gibi olur.
- Bu istek için algoritma yürütüldüğünde, algoritmanın 2. adımında ***$Request_i \leq Work$*** koşulunu sağlayan sadece ***P_0*** process'idir, ***P_0*** process'i tarafından tutulan kaynaklar algoritmanın 3. adımında geri alınır,
- Diğer process'lerin hiç biri 2. adımı sağlamaz, mevcut kaynakların sayısı diğer process'lerin isteklerini yerine getirmek için yeterli değildir. ***P_0*** hariç diğer process'lerin hepsinin Finish[i] durumu false olur.
  - Dolayısıyla, ***P_1, P_2, P_3 ve P_4*** process'lerinden oluşan bir deadlock mevcuttur.

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

Deadlock Algılama Algoritmalarının Kullanımı

- Deadlock Algılama algoritması ne zaman çalıştırmalıdır? Bunun cevabı iki faktöre bağlıdır:
  1. Ne sıklıkta bir deadlock meydana gelebilir?
  2. Deadlock meydana geldiğinde kaç process deadlock'tan etkilenecek?
- Sık sık deadlock meydana gelirse, algılama algoritması sık sık çalıştırılmalıdır.
- Ancak, her kaynak talebi için deadloock algılama algoritmasını çağırmak, hesaplama süresinde önemli bir ek yüke neden olacaktır.
  - Daha ucuz bir alternatif: basitçe algoritmayı tanımlı belli aralıklarda çağırmaktır - örneğin, saatte bir kez veya CPU kullanımı yüzde 40'ın altına düştüğünde.

Deadlock Algılama Algoritmalarının Kullanımı

- Deadlock, yalnızca bazı process'ler hemen verilemeyen bir talepte bulunduğunda ortaya çıkar. Bu talep, bir bekleyen process'ler zincirini tamamlayan son talep olabilir:
  - Dolayısıyla bir tahsis talebinin **hemen kabul edilemediği her seferde**, deadloock algılama algoritmasını çalıştırabiliriz.
  - Bu durumda, yalnızca deadlock process'leri değil, aynı zamanda deadlock'a "neden olan" belirli process'i de belirleyebiliriz.
- Deadlock Algılama algoritması, zaman içinde rastgele noktalarda çalıştırılırsa, kaynak grafı birçok döngü içerebilir.
  - Bu durumda, genellikle deadlock olan birçok process'ten hangisinin deadlock'a "neden olduğunu" söyleyemeyiz.

Konular

- Sistem modeli
- Deadlock tanımı ve özellikleri
- Deadlock yönetimi için metotlar
- Deadlock önleme
- Deadlock’tan kaçınma
- Deadlock algılama
- Deadlock’tan kurtulma

Deadlock'tan Kurtulma

- Bir algılama algoritması bir deadlock olduğunu belirlediğinde, birkaç alternatif mevcuttur:
  - Operatöre bir deadlock oluştuğu bildirilir ve operatörün deadlock'la manuel olarak ilgilenmesine izin verilir.
  - Sistemin deadlock durumundan otomatik olarak kurtulmasına izin verilir.
- Bir deadlock'tan kurtulmak için iki seçenek vardır:
  1. Döngüsel beklemeyi bozmak için bir veya daha fazla process'in iptal edilmesi/sonlandırılmasıdır (abort process).
  2. Deadlock olmuş process'lerin baz kaynakları kullanmasının kesintiye (preempt edilmesi) uğratılmasıdır.

Process'lerin Sonlandırılması - Deadlock'tan Kurtulma

- Bir process'i sonlandırarak deadlock'ları ortadan kaldırmak için iki yöntemden birini kullanılır. Her iki yöntemde de sistem, sonlandırılan process'lere tahsis edilen tüm kaynakları geri alır.
  1. Tüm deadlock durumundaki process'ler sonlandırılır. Process'lerin sonuçları kaybolabilir.
  2. Deadlock döngüsü ortadan kalkıncaya kadar her adımda bir deadlock process sonlandırılır. Her process sonlandığında deadlock cycle kontrolü yapılması gereklidir.
- Bir process'in abort edilmesi sonucunda tutarsızlıklar ortaya çıkabilir. Örneğin,
  - Process bir dosyayı güncellemenin ortasındaysa, process'i sonlandırmak o dosyayı yanlış bir durumda bırakabilir.
  - Process bir muteks lock'ı tutarken paylaşılan verileri güncellemenin ortasındaysa sistem, lock'ın durumunu kullanılabilir olarak geri yüklemelidir ancak paylaşılan verilerin bütünlüğü ile ilgili hiçbir garanti verilemeyecektir.

Process'lerin Sonlandırılması - Deadlock'tan Kurtulma

- Kısmi sonlandırma yöntemi kullanılıyorsa, deadlock olmuş hangi process'in (veya process'lerin) sonlandırılması gerektiğine karar verilmelidir. Bu karar verme, CPU Scheduling kararlarına benzer bir karardır.
- Hangi process'in (veya process'lerin) sonlandırılacağı sorusu ekonomik bir sorundur; minimum maliyete neden olacak process'ler iptal edilmelidir.

Process'lerin Sonlandırılması - Deadlock'tan Kurtulma

- Minimum maliyet kesin bir kriter değildir. Aşağıdakiler dahil birçok faktör hangi process'in seçileceğini etkileyebilir:
  1. Process'in önceliğinin ne olduğu
  2. Process'in ne kadar süredir yürütüldüğü ve belirlenen görevi tamamlaması için process'in daha ne kadar süre yürütüleceği
  3. Process'in kaç tane ve ne tür kaynak kullandığı (örneğin, kaynakların öncelikli olup olmadığı)
  4. Process'in tamamlanması için daha kaç kaynağa ihtiyacı olduğu
  5. Kaç process'in sonlandırılması gerektiği
  6. Process'in etkileşimli (interactive) mi yoksa toplu (batch) mu olduğu

Kaynakların Preemption Edilmesi - Deadlock'tan Kurtulma

- Kaynakların preemption edilerek deadlock'ları ortadan kaldırmak için, deadlock döngüsü bozulana kadar process'lerin kullandığı bazı kaynaklar sırayla ellerinden alınıp diğer process'lere verilir.
- Deadlock'larla başa çıkmak için preempiton gerekiyorsa, üç konunun ele alınması gerekir:
  1. Selecting a victim: Process sonlandırmada olduğu gibi maliyeti en aza indirmek için preemption sırası belirlenmelidir.
  2. Kaynağı elinden alınan process güvenli bir duruma geri alınmalı ve bu durumdan yeniden başlatılmalıdır. *
  3. Kaynağı elinden alınacak process'in seçimi maliyet faktörüne göre yapıldığından aynı process sürekli kurban olarak seçilebilir bu da starvation'a neden olabilir. *