İşletim Sistemleri

Binnur Kurt
binnur.kurt@ieee.org

İstanbul Teknik Üniversitesi
Bilgisayar Mühendisliği Bölümü

Version 0.0.2

About the Lecturer

- **BSc**

  İTÜ, Computer Engineering Department, 1995
- **MSc**

  İTÜ, Computer Engineering Department, 1997
- Areas of Interest
  - Digital Image and Video Analysis and Processing
  - Real-Time Computer Vision Systems
  - Multimedia: Indexing and Retrieval
  - Software Engineering
  - OO Analysis and Design

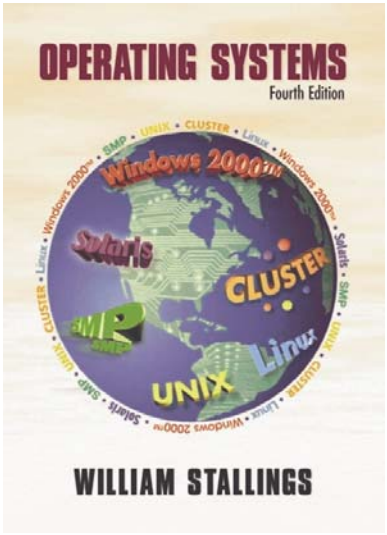
Önemli Bilgiler

- **Dersin**
  - Gün ve Saati
    - 18:30-21:30 Cuma
  - Adresi
    - http://www.cs.itu.edu.tr/~kurt/Courses/os
  - E-posta
    - kurt@ce.itu.edu.tr

Notlandırma

- 3 Ödev (30%)
- Yıliçi Sınavı (30%)
- Final Sınavı (40%)

Kaynakça



Tell me and I forget.
Show me and I remember.
Let me do and I understand.

—Chinese Proverb

İçerik

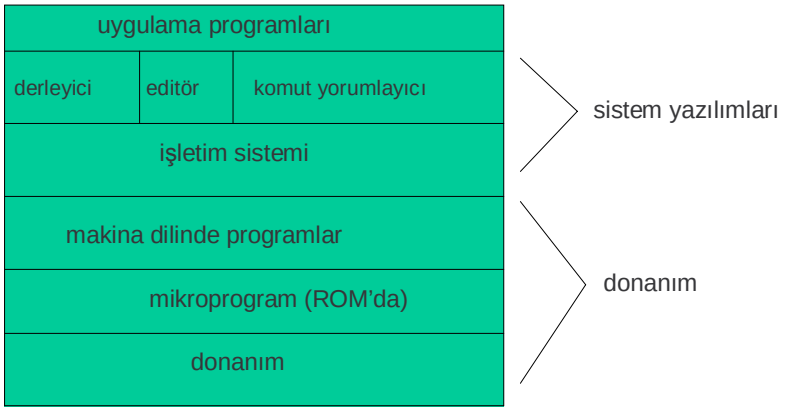
1. Giriş.
2. Prosesler ve Proses Kontrolü.
3. İplikler
4. Prosesler Arası İletişim
5. Ölümcül Kilitlenme
6. İş Sıralama
7. Bellek Yönetimi
8. Giriş/Çıkış Yönetimi
9. Dosya Sistemi

1 GİRİŞ

İşletim Sistemi

- donanımı kullanılabilir yapan yazılım
  - bilgisayar kaynaklarını:
    - denetler,
    - paylaştırır
- üzerinde program geliştirme ve çalıştırma ortamı
- çekirdek (kernel) = işletim sistemi

Bilgisayar Sistemi



İşletim Sistemi

- güncel işletim sistemleri doğrudan donanıma erişmeyi engeller
  - kullanıcı modu × çekirdek modu
- donanımın doğrudan kullanımının zorluklarını gizler
- kullanıcı ve donanım arasında arayüz
  - sistem çağrıları

Sistem Çağrıları

- kullanıcı programların
  - işletim sistemi ile etkileşimi ve
  - işletim sisteminden iş isteği için
- her sistem çağrısına karşılık kütüphane rutini
- kullanıcı program kütüphane rutinini kullanır

İşletim Sisteminin Temel Görevleri

- kaynak paylaşımı
- görüntü makina sağlanması

Kaynak Paylaşımı

- kullanıcılar arasında paylaşım
- güvenlik
  - kullanıcıları birbirinden yalıtır
- paylaşılan kaynaklar:
  - işlemci
  - bellek
  - G / Ç birimleri
  - veriler

Kaynak Paylaşımı

- amaçlar:
  - kaynakların kullanım oranını yükseltmek (utilization)
  - bilgisayar sisteminin kullanılabilirliğini arttırmak (availability)

Kaynak Paylaşımı

- verdiği hizmetler:
  - kullanıcı arayüzünün tanımlanması
    - sistem çağrıları
  - çok kullanıcılı sistemlerde donanımın paylaştırılması ve kullanımın düzenlenmesi
    - kaynaklar için yarışı önlemek
    - birbirini dışlayan kullanım
  - kullanıcıların veri paylaşımını sağlamak (paylaşılan bellek bölgeleri)
  - kaynak paylaşımının sıralanması (scheduling)
  - G/Ç işlemlerinin düzenlenmesi
  - hata durumlarından geri dönüş

Kaynak Paylaşımı

- örnek:
  - yazıcı paylaşılamaz; bir kullanıcının işi bitince diğeri kullanabilir
  - ekranda paylaşım mümkün

Görüntü Makina Sağlanması

- donanımın kullanılabilir hale getirilmesi
- kullanıcı tek başına kullanıyormuş gibi
  - kaynak paylaşımı kullanıcıya şeffaf
- görüntü makinanın özellikleri fiziksel makinadan farklı olabilir:
  - G/Ç
  - bellek
  - dosya sistemi
  - koruma ve hata kotarma
  - program etkileşimi
  - program denetimi

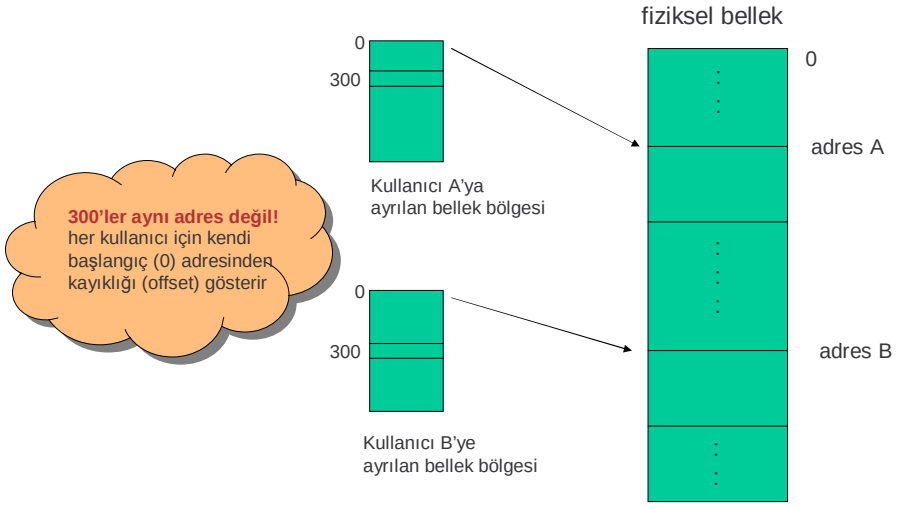
Görüntü Makina Sağlanması

- G/Ç
  - donanıma yakın programlama gerekir
  - işletim sistemi kullanımı kolaylaştırır
    - aygıt sürücüler
  - örnek: diskten / disketten okuma

Görüntü Makina Sağlanması

- Bellek
  - fiziksel bellekten farklı kapasitede görüntü makina
    - disk de kullanılarak daha büyük
    - kullanıcılar arasında paylaştırılarak daha küçük
      - her kullanıcı kendine ayrılan bellek alanını görür

Görüntü Makina Sağlanması



Görüntü Makina Sağlanması

- Dosya sistemi
  - program ve verilerin uzun vadeli saklanması için
  - disk üzerinde
  - bilgilere erişimde fiziksel adresler yerine simgeler kullanımı
    - isimlendirme
      - UNIX işletim sisteminde herşey dosya

Görüntü Makina Sağlanması

- Koruma ve hata kotarma
  - çok kullanıcılı sistemlerde kullanıcıların birbirlerinin hatalarından etkilenmemesi

Görüntü Makina Sağlanması

- Program etkileşimi
  - çalışma anında programların etkileşmesi
    - örneğin birinin ürettiği çıkış diğerine giriş verisi olabilir

Görüntü Makina Sağlanması

- Program denetimi
  - kullanıcıya yüksek düzeyli bir komut kümesi
    - kabuk (shell) komutları
      - kabuk: komut yorumlayıcı
      - kabuk işletim sistemi içinde değil
      - ama sistem çağrılarını yoğun kullanır

İşletim Sistemi Türleri

- Anaçatı işletim sistemleri (mainframe)
- Sunucu (server) işletim sistemleri
- Çok işlemcili işletim sistemleri
- Kişisel bilgisayar işletim sistemleri
- Gerçek zamanlı (real-time) işletim sistemleri
- Gömülü (embedded) işletim sistemleri
- Akıllı-kart (smart card) işletim sistemleri

Anaçatı İşletim Sistemleri

- yoğun G/Ç işlemi gerektiren çok sayıda görev çalıştırmaya yönelik
- üç temel hizmet:
  - batch modda çalışma
    - etkileşimsiz, rutin işler
    - örneğin bir sigorta şirketindeki sigorta tazminatı isteklerinin işlenmesi
  - birim-iş (transaction) işleme
    - çok sayıda küçük birimler halinde gelen isteklere yanıt
    - örneğin havayollarında rezervasyon sistemi
  - zaman paylaşımlı çalışma
    - birden fazla uzaktan bağlı kullanıcının sistemde iş çalıştırması
      - örnek: veri tabanı sorgulaması
  - Örnek: OS/390

Sunucu İşletim Sistemleri

- sunucular üzerinde çalışır
  - büyük kaynak kapasiteli kişisel bilgisayarlar
  - iş istasyonları
  - anaçatı sistemler
- bilgisayar ağı üzerinden çok sayıda kullanıcıya hizmet
  - donanım ve yazılım paylaştırma
  - örneğin: yazıcı hizmeti, dosya paylaştırma, web erişimi
- örnek: UNIX, Windows 2000

Çok İşlemcili İşletim Sistemleri

- birden fazla işlemcili bilgisayar sistemleri
- işlem gücünü arttırma
- işlemcilerin bağlantı türüne göre:
  - paralel sistemler
  - birbirine bağlı, birden fazla bilgisayardan oluşan sistemler
  - çok işlemcili sistemler
- özel işletim sistemi gerek
  - temelde sunucu işletim sistemlerine benzer tasarım hedefleri
  - işlemciler arası bağlaşım ve iletişim için ek özellikler

Kişisel Bilgisayar İşletim Sistemleri

- kullanıcıya etkin ve kolay kullanılır bir arayüz sunma amaçlı
- genellikle ofis uygulamalarına yönelik
- örnek:
  - Windows 98, 2000, XP
  - Macintosh
  - Linux

Gerçek Zamanlı İşletim Sistemleri

- zaman kısıtları önem kazanır
- endüstriyel kontrol sistemleri
  - toplanan verilerin sisteme verilerek bir yanıt üretilmesi (geri-besleme)
- iki tip:
  - katı-gerçek-zamanlı (hard real-time)
    - zaman kısıtlarına uyulması zorunlu
    - örneğin: araba üretim bandındaki üretim robotları
  - gevşek-gerçek-zamanlı (soft-real-time)
    - bazı zaman kısıtlarına uyulmaması mümkün
    - örneğin: çoğulortam sistemleri
- örnek: VxWorks ve QNX

Gömülü İşletim Sistemleri

- avuç-içi bilgisayarlar ve gömülü sistemler
- kısıtlı işlevler
- özel amaçlı
- örneğin: TV, mikrodalga fırın, cep telefonları, ...
- bazı sistemlerde boyut, bellek ve güç harcama kısıtları var
- örnek: PalmOS, Windows CE

Akıllı-Kart İşletim Sistemleri

- en küçük işletim sistemi türü
- kredi kartı boyutlarında, üzerinde işlemci olan kartlar üzerinde
- çok sıkı işlemci ve bellek kısıtları var
- bazıları tek işleve yönelik (örneğin elektronik ödemeler)
- bazıları birden fazla işlev içerebilir
- çoğunlukla özel firmalar trafından geliştirilen özel sistemler
- bazıları JAVA tabanlı (JVM var)
  - küçük JAVA programları (applet) yüklenip çalıştırılır
  - bazı kartlar birden fazla program (applet) çalıştırabilir
    - çoklu-programlama, iş sıralama ve kaynak yönetimi ve koruması

Temel İşletim Sistemi Yapıları

- Monolitik
- Katmanlı
- Sanal Makinalar
- Dış-çekirdek (exo-kernel)
- Sunucu-İstemci Modeli
- Modüler

Monolitik İşletim Sistemleri

- genel bir yapı yok
- işlevlerin tamamı işletim sistemi içinde
- işlevleri gerçekleyen tüm prosedürler
  - aynı seviyede
  - birbirleri ile etkileşimli çalışabilir
- büyük

Modüler Çekirdekli İşletim Sistemleri

- çekirdek minimal
- servisler gerektikçe çalışma anında modül olarak çekirdeğe eklenir
  - örneğin aygıt sürücüler
- küçük çekirdek yapısı
- daha yavaş
- örnek: LINUX

Katmanlı Yapılı İşletim Sistemleri

- işletim sistemi katmanlı
  - hiyerarşik
- örnek: THE işletim sistemi

5	operatör
4	kullanıcı programları
3	G/Ç yönetimi
2	operatör-proses iletişimi
1	bellek ve tambur yönetimi
0	işlemci paylaştırma ve çoklu-programlama

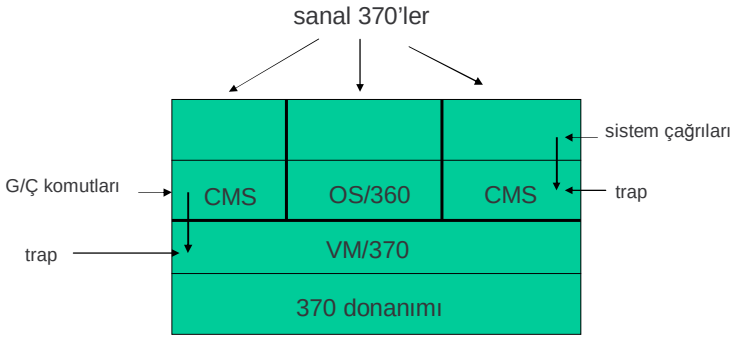
• katman 0 işlemciyi prosesler arası paylaştırır (iş sıralama)
• katman 1 bellek yönetimini yapar (bellek ve tambur arası)
• ...

Her katman altındakinin yaptıklarıyla ilgilenmez. Örnek: 2. katmandaki işlemler için prosesin bellek veya tamburda olması önemli değil.

Sanal Makina

- VM/370

- VM donanım üzerinde koşar
- çoklu programlama yapar
- birden fazla sanal makina sunar
- sanal makinaların her biri donanımın birebir kopyası
- her sanal makinada farklı işletim sistemi olabilir



Dış-Çekirdek (Exo-Kernel)

- MIT'de geliştirilmiş
- sanal makina benzeri
  - sistemin bir kopyasını sunar
  - fark: her sanal makinaya kaynakların birer alt kümesini tahsis eder
    - dönüşüm gerekmez; her makinaya ayrılan kaynakların başı-sonu belli
- dış çekirdek var
  - görevi: sanal makinaların kendilerine ayrılan kaynaklar dışına çıkmamasını kontrol eder
- her sanal makinada farklı işletim sistemi olabilir

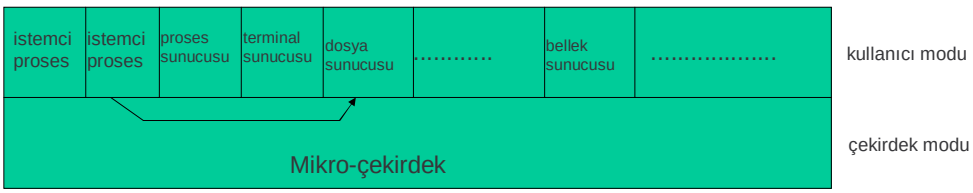
Sunucu-İstemci Modeli

- çekirdek minimal (mikro-çekirdek)
- işletim sisteminin çoğu kullanıcı modunda
- sunumcular ve istemci prosesler var
  - örneğin dosya okuma işlemi
    - istemci proses sunucudan ister
    - sunucu işlemi yürütür
    - yanıtı istemciye verir
- çekirdek sunucu ve istemciler arası iletişimi yönetir

Sunucu-İstemci Modeli

- sunucular kullanıcı modunda
  - dosya sunucusu
  - proses sunucusu
  - terminal sunucusu
  - bellek sunucusu
- işletim sisemi alt birimlerden oluştuğundan:
  - yönetimi kolay
  - bir birimdeki hata tüm sistemi çökertmez (birimler donanıma doğrudan ulaşamaz)
  - gerçeklemede sorunlar: özellikle G/Ç aygıtlarının yönetiminin tamamen kullanıcı düzeyinde yapılması mümkün değil
- dağıtık sistemlerde kullanılmaya çok elverişli yapı

Sunucu-İstemci Modeli



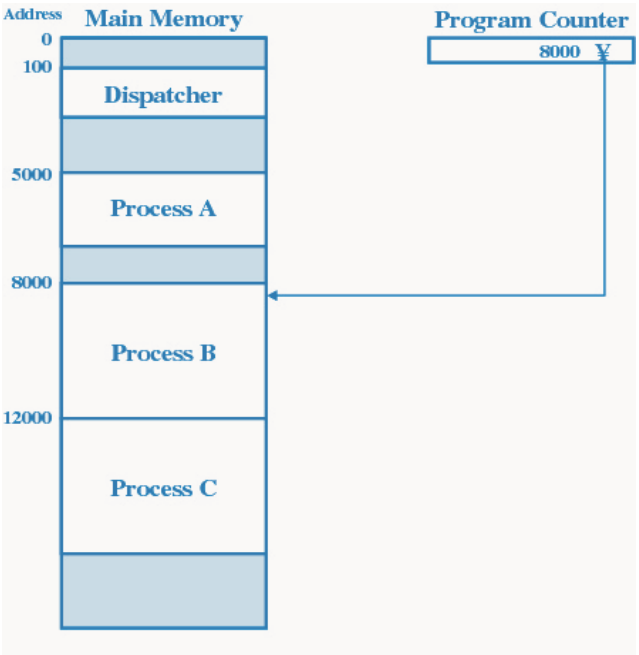
2 PROSESLER

Proses

- Bir işlevi gerçeklemek üzere ardışıl bir program parçasının yürütülmesiyle ortaya çıkan işlemler dizisi
  $\Rightarrow$ Programın koşmakta olan hali
- Aynı programa ilişkin birden fazla proses olabilir.
- Görev (Task) de denir
- Text, veri ve yığın alanları vardır.

Proses

- Bazı sistem çağrıları ile sistem kaynaklarını kullanırlar.
- Birbirleri ve dış dünya ile haberleşirler.
- Davranışını karakterize edebilmek için proses için yürütülen komutların sırası gözlenebilir: prosesin izi (trace)
- Prosesin ömrü: yaratılması ve sonlanması arasında geçen süre



(a) Trace of Process A	(b) Trace of Process B	(c) Trace of Process C
5000	8000	12000
5001	8001	12001
5002	8002	12002
5003	8003	12003
5004		12004
5005		12005
5006		12006
5007		12007
5008		12008
5009		12009
5010		12010
5011		12011

5000 = Starting address of program of Process A
8000 = Starting address of program of Process B
12000 = Starting address of program of Process C

1	5000
2	5001
3	5002
4	5003
5	5004
6	5005
----------	Time out
7	100
8	101
9	102
10	103
11	104
12	105
13	8000
14	8001
15	8002
16	8003
----------	I/O request
17	100
18	101
19	102
20	103
21	104
22	105
23	12000
24	12001
25	12002
26	12003
27	12004
28	12005
----------	Time out
29	100
30	101
31	102
32	103
33	104
34	105
35	5006
36	5007
37	5008
38	5009
39	5010
40	5011
----------	Time out
41	100
42	101
43	102
44	103
45	104
46	105
47	12006
48	12007
49	12008
50	12009
51	12010
52	12011
----------	Time out

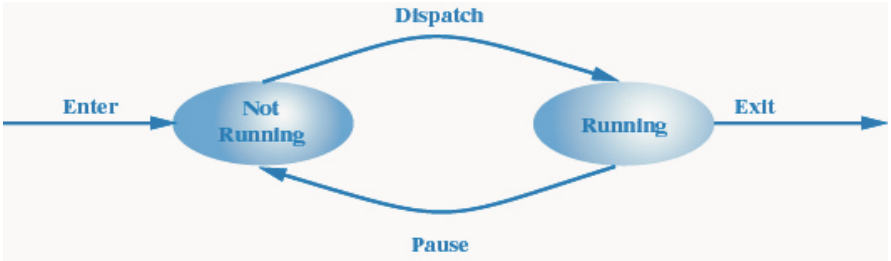
100 = Starting address of dispatcher program

Proses

- Proseslerin işlemciye sahip olma sıraları kestirilemez ⇒ program kodunda zamanlamaya dayalı işlem olmamalı

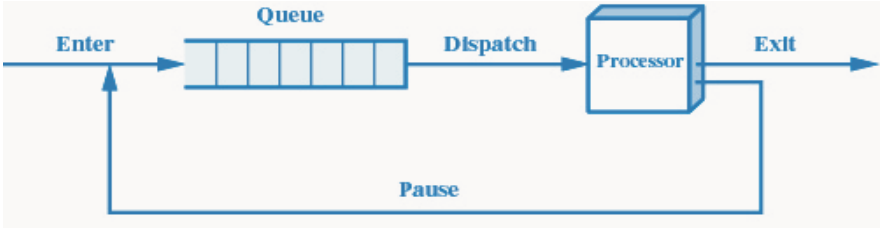
İki Durumlu Proses Modeli

- Proses iki durumdan birinde olabilir:
  - Koşuyor
  - Koşmuyor



Proses Kuyruğu

O anda çalışmayan proses sırasını bir kuyrukta bekler:

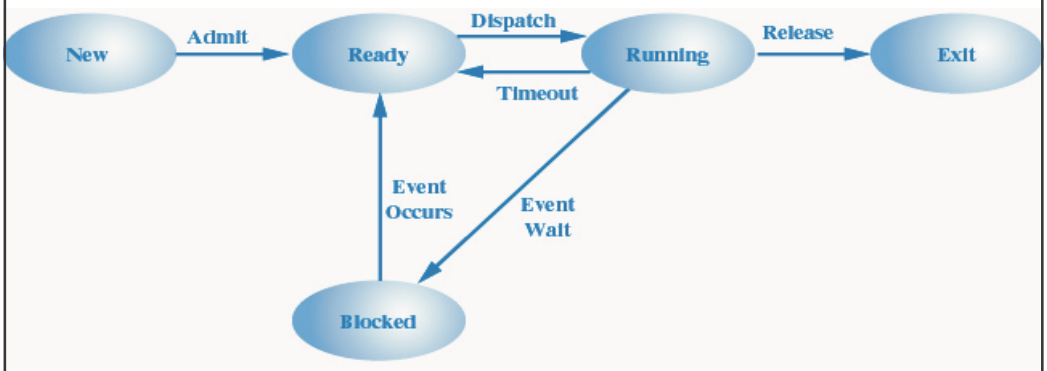


Proses

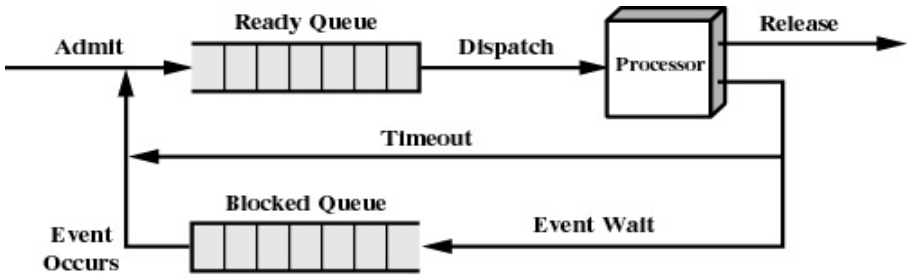
- Koşmuyor
  - çalışmaya hazır
- Bloke
  - G/Ç bekliyor
- Kuyrukta en uzun süre beklemiş prosesin çalıştırılmak üzere seçilmesi doğru olmaz
  - Bloke olabilir

Beş-Durumlu Model

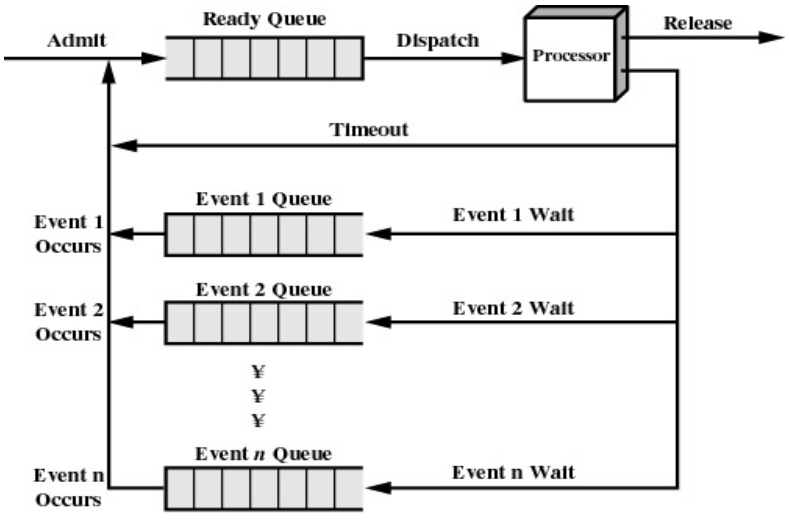
- Koşuyor
- Hazır
- Bloke
- Yeni
- Sonlanıyor



İki Kuyruk



Çoklu Kuyruk



Proses Yaratma

Ne zaman yaratılır?

- Kullanıcı sisteme girmiş
- Bir servis sunmak için
  - örneğin yazıcıdan çıktı
- Bir başka proses yaratmış

Proses Sonlandırma

Ne zaman sonlanır?

- Kullanıcı sistemden çıkmış
- Uygulama sonlandırılmış
- Hata durumu oluşmuş

Prosesin Askıya Alınma Nedenleri

- Swap işlemi
- Hatalı durum oluşması
- Etkileşimli kullanıcı isteği
  - Örneğin hata ayıklama (debug) için
- Ayrılan sürenin dolması (quantum)
- Anne proses tarafından

İşletim Sistemi Kontrol Yapıları

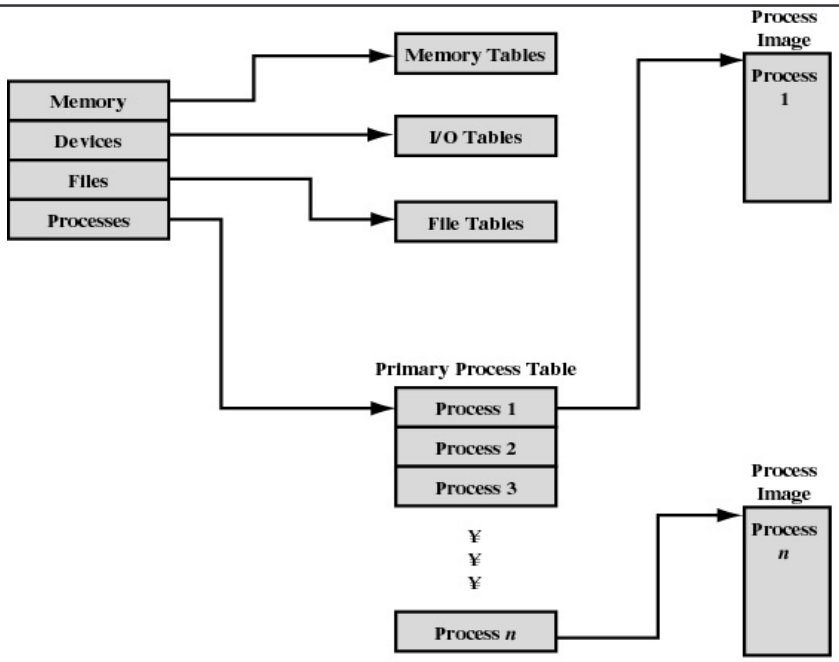
- Her proses ve kaynak ile ilgili durum bilgilerinin tutulması gerekir
  - İşletim sistemi tarafından yönetilen her varlık için tablolar tutulur
    - G/Ç Tabloları
    - Bellek Tabloları
    - Dosya Tabloları
    - Proses Tabloları

Proses Tablosu

- Prosesin bileşenleri
- Yönetilmesi için gerekli özellikleri
  - Kimlik numarası
  - Durumu
  - Bellekteki yeri

Prosesin Bileşenleri

- Proses birden fazla programdan oluşabilir
  - Yerel ve global değişkenler
  - Sabitler
  - Yığın
- Proses Kontrol Bloğu
  - Nitelikler (attributes)
- Prosesin görüntüsü
  - Program, veri, yığın ve niteliklerin tamamı



Proses Kontrol Bloğu

- Proses Kimlik Bilgileri
  - Kimlik Bilgileri
    - Prosesin kimlik numarası
    - Prosesin annesinin kimlik numarası
    - Sahibin kullanıcı kimlik bilgisi

Proses Kontrol Bloğu

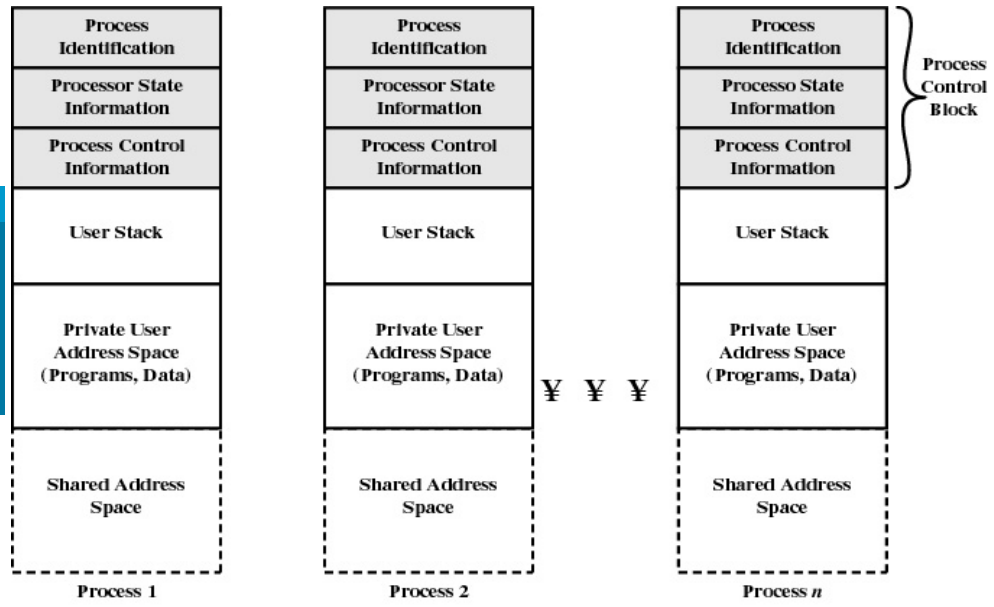
- İşlemci Durum Bilgisi
  - Kullanıcıya açık saklayıcılar
    - İşlemcinin makina dili kullanılarak erişilebilen saklayıcıları.
  - Kontrol ve Durum saklayıcıları
    - Program sayacı
    - Durum saklayıcısı
    - Yığın işaretçileri
    - Program durum sözcüğü (çalışma modu biti var)

Proses Kontrol Bloğu

- Proses Kontrol Bilgileri
  - İş sıralama ve durum bilgileri
    - Prosesin durumu
    - Önceliği
    - İş sıralama ile ilgili bilgiler (Hangi bilgiler olduğu kullanılan iş sıralama algoritmasına bağlı. Örneğin: bekleme süresi, daha önce koştuğu süre)
    - Çalışmak için beklediği olay
  - Veri Yapıları
    - Prosesler örneğin bir çevrel kuyruk yapısında birbirlerine bağlı olabilir (örneğin aynı kaynağı bekleyen eş öncelikli prosesler).
    - Prosesler arasında anne-çocuk ilişkisi olabilir

Proses Kontrol Bloğu

- Prosesler arası haberleşme ile ilgili bilgiler
  - Bazı bayrak, sinyal ve mesajlar proses kontrol bloğunda tutulabilir.
- Proses Ayrıcalıkları
  - Bellek erişimi, kullanılabilecek komutlar ve sistem kaynak ve servislerinin kullanımı ile ilgili haklar
- Bellek yönetimi
  - Prosese ayrılmış sanal bellek bölgesinin adresi
- Kaynak kullanımı
  - Prosesin kullandığı kaynaklar: örneğin açık dosyalar
  - Prosesin önceki işlemci ve diğer kaynakları kullanımına ilişkin bilgiler



Çalışma Modları

- Kullanıcı modu
  - Düşük haklar ve ayrıcalıklar
  - Kullanıcı programları genel olarak bu modda çalışır
- Sistem modu / çekirdek modu
  - Yüksek haklar ve ayrıcalıklar
  - İşletim sistemi çekirdeği prosesleri bu modda çalışır

Proses Yaratılması

- Proses kimlik bilgisi atanır: sistemde tek
- Proses için bellekte yer ayrılır
- Proses kontrol bloğuna ilk değerler yüklenir
- Gerekli bağlantılar yapılır: Örneğin iş sıralama için kullanılan bağlantılı listeye yeni proses kaydı eklenir.
- Gerekli veri yapıları yaratılır veya genişletilir: Örneğin istatistik tutma ile ilgili

Prosesler Arası Geçiş Durumu

- Saat kesmesi
  - proses kendisine ayrılan zaman dilimi kadar çalışmıştır
- G/Ç kesmesi
- Bellek hatası
  - erişilen bellek bölgesi ana bellekte yoktur
- Hata durumu
- Sistem çağrısı

Proseslerin Durum Değiştirmesi

- İşlemci bağlamının saklanması (program sayacı ve diğer saklayıcılar dahil
- O anda koşmakta olan prosesin proses kontrol bloğunun güncellenmesi
- Prosese ilişkin proses kontrol bloğunun uygun kuyruğa yerleştirilmesi: hazır / bloke
- Koşacak yeni prosesin belirlenmesi

Proseslerin Durum Değiştirmesi

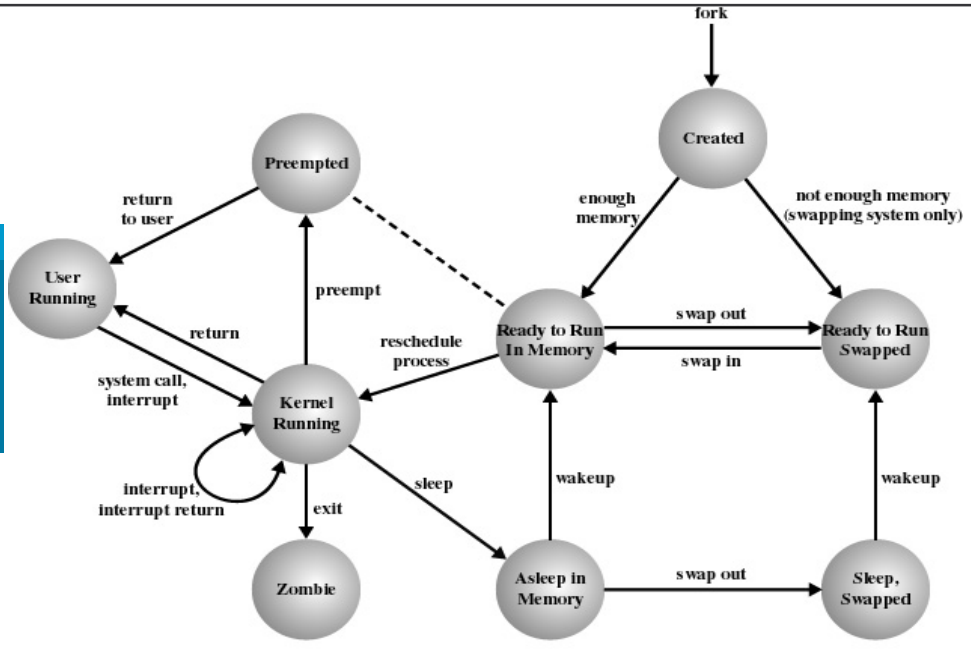
- Seçilen prosesin proses kontrol bloğunun güncellenmesi
- Bellek yönetimi ile ilgili bilgilerin güncellenmesi
- Seçilen prosesin bağlamının yüklenmesi

UNIX'te Proses Durumları

- Kullanıcı modunda koşuyor
- Çekirdek modunda koşuyor
- Bellekte ve koşmaya hazır
- Bellekte uyuyor
- İkincil bellekte ve koşmaya hazır
- İkincil bellekte uyuyor

UNIX'te Proses Durumları

- Pre-empt olmuş (çekirdek modundan kullanıcı moduna dönerken iş sıralayıcı prosesi kesip yerine bir başka prosesi çalışacak şekilde belirlemiş)
- Yaratılmış ama koşmaya hazır değil
- Zombie (proses sonlanmış ancak anne prosesin kullanabilmesi için bazı kayıtları hala tutulmakta, ilgili kaynaklar henüz geri verilmemiş)



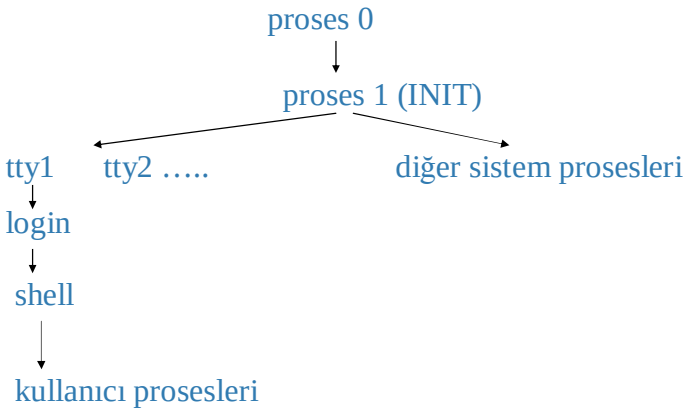
UNIX'de Proses Yaratma

- fork sistem çağrısı ile yaratılır
  - çağrıyı yapan proses: anne proses
  - Yaratılan proses: çocuk proses
- sentaksı pid=fork()
  - Her iki proses de aynı bağlama sahip
  - Anne prosese çocuğun kimlik değeri döner
  - Çocuk prosese 0 değeri döner
- **0** numaralı prosesi açılışta çekirdek yaratılır; fork ile yaratılmayan tek prosestir

UNIX'de Proses Yaratma

- fork sistem çağrısı yapıldığında çekirdeğin yürüttüğü işlemler:
  - proses tablosunda (varsa) yer ayırılır (maksimum proses sayısı belli)
  - çocuk prosese yeni bir kimlik numarası atanır (sistemde tek)
  - Anne prosesin bağlamının kopyası çıkarılır.
  - Dosya erişimi ile ilgili sayaçları düzenler
  - anneye çocuğun kimliğini, çocuğa da 0 değerini döndürür

UNIX'de fork Sistem Çağrısı ile Proses Yaratılma Hiyerarşisi



UNIX'de Proses Sonlanması

- exit sistem çağrısı ile
- sentaksı: exit(status)
  - "status" değeri anne prosese aktarılır
- Tüm kaynakları geri verilir
- Dosya erişim sayaçları düzenlenir
- Proses tablosu kaydı silinir
- Annesi sonlanan proseslerin annesi olarak init prosesi (1 numaralı proses) atanır

Örnek Program Kodu - 1

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int f;

int main (void)
{
  printf("\n Program calisiyor: PID=%d \n",
            getpid());
  f=fork();
```


Örnek Program Kodu - 2

```
if (f==0)  /*cocuk*/
  {
    printf("\nBen cocuk. Kimlik= %d\n", getpid());
    printf("Annemin kimliği=%d\n", getppid());
    sleep(2);
    exit(0):
  }
  else /* anne */
  {
    printf("\nBen anne. Kimlik= %d\n", getpid());
    printf("Annemin kimliği=%d\n", getppid());
    printf("Cocugumun kimliği=%d\n", f);
    sleep(2);
    exit(0):
  }
  return(0);
}
```

3 İPLİKLER

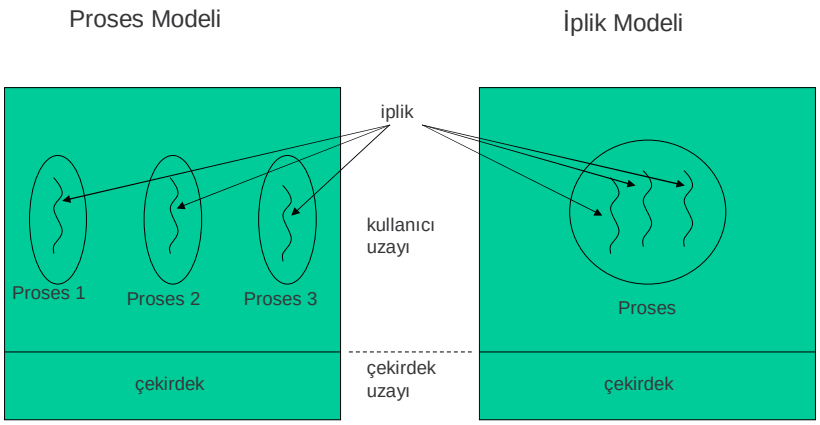
Giriş

- geleneksel işletim sistemlerinde her prosesin
  - özel adres uzayı ve
  - tek akış kontrolü var.
- aynı adres uzayında birden fazla akış kontrolü gerekebilir
  - aynı adres uzayında çalışan paralel prosesler gibi

İplik Modeli

- iplik = hafif proses
- aynı adres uzayını paylaşan paralel prosesler benzeri
- aynı proses ortamında birden fazla işlem yürütme imkanı
- iplikler tüm kaynakları paylaşır:
  - adres uzayı, bellek, açık dosyalar, ...
- çoklu iplikli çalışma
  - iplikler sıra ile koşar

İplik Modeli



İplik Modeli

- iplikler prosesler gibi birbirinden bağımsız değil:
  - adres uzayı paylaşır
    - global değişkenleri de paylaşırlar
    - birbirlerinin yığınını değiştirebilir
    - koruma yok çünkü:
      - mümkün değil
      - gerek yok

İplik Modeli

- ipliklerin paylaştıkları:
  - adres uzayı
  - global değişkenler
  - açık dosyalar
  - çocuk prosesler
  - bekleyen sinyaller
  - sinyal işleyiciler
  - muhasebe bilgileri
- her bir ipliğe özel:
  - program sayacı
  - saklayıcılar
  - yığın
  - durum

İplik Modeli

- işler birbirinden büyük oranda bağımsız ise ⇒ proses modeli
- işler birbirine çok bağlı ve birlikte yürütülüyorsa ⇒ iplik modeli
- iplik durumları = proses durumları
  - koşuyor
  - bloke
    - bir olay bekliyor: dış olay veya bir başka ipliği bekler
  - hazır

İplik Modeli

- her ipliğin kendi yığını var
  - yığında çağrılmış ama dönülmemiş yordamlarla ilgili kayıtlar ve yerel değişkenler
  - her iplik farklı yordam çağrıları yapabilir
    - geri dönecekleri yerler farklı ⇒ ayrı yığın gerekli

İplik Modeli

- prosesin başta bir ipliği var
- iplik kütüphane yordamları ile yeni iplikler yaratır
  - örn: `thread_create`
    - parametresi: koşturacağı yordamın adı
- yaratılan iplik aynı adres uzayında koşar
- bazı sistemlerde iplikler arası anne – çocuk hiyerarşik yapısı var
  - çoğu sistemde tüm iplikler eşit

İplik Modeli

- işi biten iplik kütüpane yordamı çağrısı ile sonlanır
  - örn: `thread_exit`
- zaman paylaşımı için zamanlayıcı yok
  - iplikler işlemciyi kendileri bırakır
    - örn: `thread_exit`
- iplikler arası
  - senkronizasyon ve
  - haberleşme olabilir

İplik Modeli

- ipliklerin gerçeklenmesinde bazı sorunlar:
  - örn. UNIX'te `fork` sistem çağrısı
    - anne çok iplikli ise çocuk proseste de aynı iplikler olacak mı?
    - olmazsa doğru çalışmayabilir
    - olursa
      - örneğin annedeki iplik giriş bekliyorsa çocuktaki de mi beklesin?
      - giriş olunca her ikisine de mi yollansın?
    - benzer problem açı ağ bağlantıları için de var

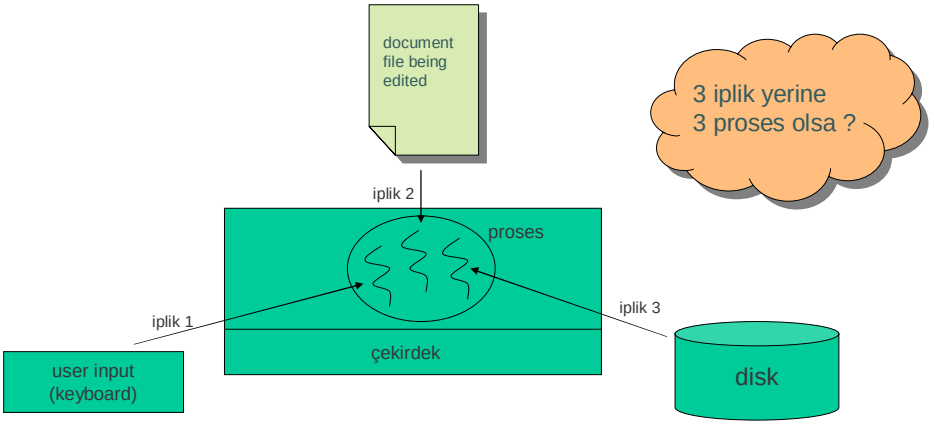
İplik Modeli

- *('sorunlar' devam)*
  - bir iplik bir dosyayı kapadı ama başka iplik o dosyayı kullanıyordu
  - bir iplik az bellek olduğunu farkedip bellek almaya başladı
    - işlem tamamlanmadan başka iplik çalıştı
    - yeni iplik de az bellek var diye bellek istedi
    - $\Rightarrow$ iki kere bellek alınabilir
- çözümler için iyi tasarım ve planlama gerekli

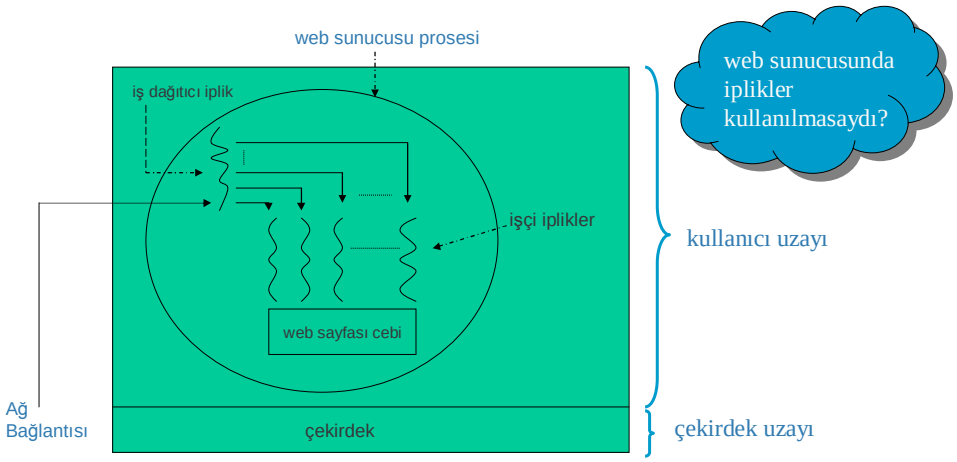
İpliklerin Kullanımı

- neden iplikler?
  - bir proses içinde birden fazla işlem olabilir
    - bazı işlemler bazen bloke olabilir; ipliklere bölmek performansı arttırır
  - ipliklerin kendi kaynakları yok
    - yaratılmaları / yok edilmeleri proseslere göre kolay
  - ipliklerin bazıları işlemciye yönelik bazıları giriş-çıkış işlemleri yapıyorsa performans artar
    - hepsi işlemciye yönelikse olmaz
  - çok işlemcili sistemlerde faydalı

İplik Kullanımına Örnek – 3 İplikli Kelime İşlemci Modeli



İplik Kullanımına Örnek – Web Sitesi Sunucusu



İplik Kullanımına Örnek – Web Sitesi Sunucusu

İş dağıtıcı iplik kodu

```
while TRUE {
    sıradaki_isteği_al(&tmp);
    işi_aktar(&tmp);
}
```

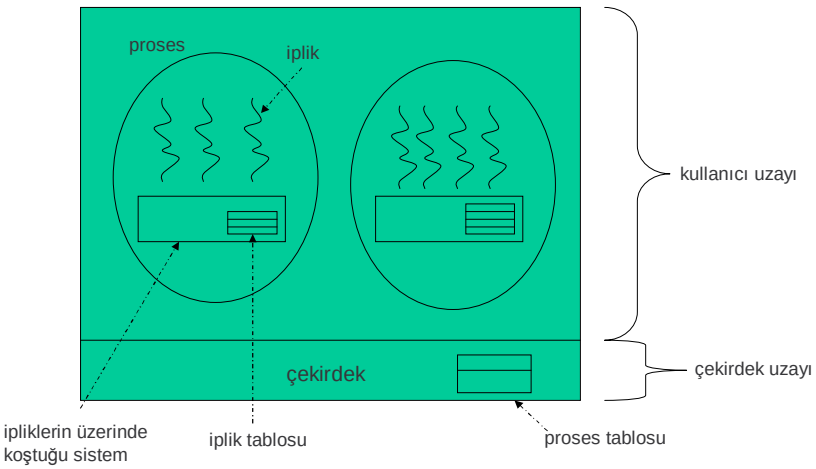
İşçi ipliklerin kodu

```
while TRUE {
    iş_bekle(&tmp);
    sayfayı_cepte_ara(&tmp,&sayfa);
    if (sayfa_cepte_yok(&sayfa)
        sayfayı_diskten_oku(&tmp,&sayfa);
    sayfayı_döndür(&sayfa);
}
```

İpliklerin Gerçeklenmesi

- iki türlü gerçekleme mümkün
  - kullanıcı uzayında
  - çekirdek uzayında
- hibrid bir gerçekleme de olabilir

İpliklerin Kullanıcı Uzayında Gerçeklenmesi



İpliklerin Kullanıcı Uzayında Gerçeklenmesi

- çekirdeğin ipliklerden haberi yok
- çoklu iplik yapısını desteklemeyen işletim sistemlerinde de gerçeklenebilir
- ipliklerin üzerinde koştuğu sistem
  - iplik yönetim yordamları
    - örn. `thread_create`, `thread_exit`, `thread_yield`, `thread_wait`, ...
    - iplik tablosu
      - program sayacı, saklayıcılar, yığın işaretçisi, durumu, ...

İpliklerin Kullanıcı Uzayında Gerçeklenmesi

- iplik bloke olacak bir işlem yürüttüyse
  - örneğin bir başka ipliğin bir işi bitirmesini beklemek
  - bir rutin çağırır
  - rutin ipliği bloke durum sokar
  - ipliğin program sayacı ve saklayıcı içeriklerini iplik tablosuna saklar
  - sıradaki ipliğin bilgilerini tablodan alıp saklayıcılara yükler
  - sıradaki ipliği çalıştırır
  - hepsi yerel yordamlar $\Rightarrow$ sistem çağrısı yapmaktan daha hızlı

İpliklerin Kullanıcı Uzayında Gerçeklenmesi

- avantajları:
  - ipliklerin ayrı bir iş sıralama algoritması olabilir
  - çekirdekte iplik tablosu yeri gerekmiyor
  - tüm çağrılar yerel rutinler ⇒ çekirdeğe çağrı yapmaktan daha hızlı

İpliklerin Kullanıcı Uzayında Gerçeklenmesi

- Problemler:
  - bloke olan sistem çağrılarının gerçeklenmesi
    - iplik doğrudan bloke olan bir sistem çağrısı yapamaz ⇒ tüm iplikler bloke olur
    - sistem çağrıları değiştirilebilir
      - işletim sisteminin değiştirilmesi istenmez
      - kullanıcı programlarının da değişmesi gerekir
    - bazı sistemlerde yapılan çağrının bloke olup olmayacağını döndüren sistem çağrıları var
      - sistem çağrılarına ara-birim (wrapper) yazılır
      - önce kontrol edilir, bloke olunacaksa sistem çağrısı yapılmaz, iplik bekletilir

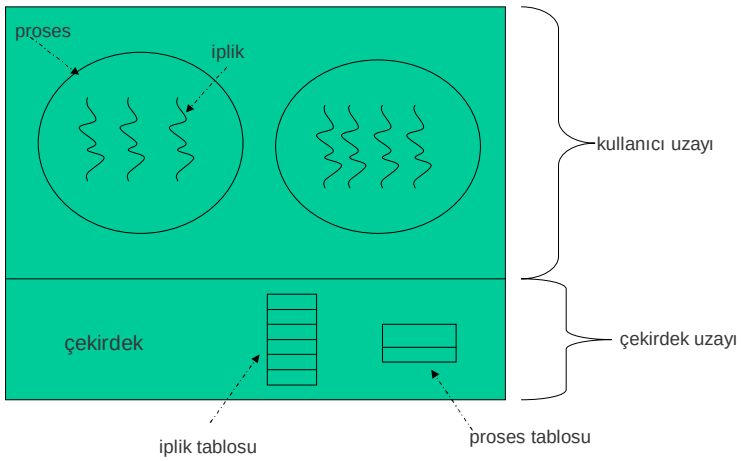
İpliklerin Kullanıcı Uzayında Gerçeklenmesi

- *(problemler devam)*
  - sayfa hataları
    - programın çalışması gereken kod parçasına ilişkin kısım ana bellekte değilse
      - sayfa hatası olur
      - proses bloke olur
      - gereken sayfa ana belleğe alınır
      - proses çalışabilir
    - sayfa hatasına iplik sebep olduysa
      - çekirdek ipliklerden habersiz
      - tüm proses bloke edilir

İpliklerin Kullanıcı Uzayında Gerçeklenmesi

- *(problemler devam)*
  - iş sıralama
    - iplik kendisi çalışmayı bırakmazsa diğer iplikler çalışamaz
      - altta çalışan sistem belirli sıklıkta saat kesmesi isteyebilir
        - » ipliklerin de saat kesmesi ile işi varsa karışır
  - çok iplikli çalışma istendiği durumlarda sıkça bloke olan ve sistem çağrısı yapan iplikler olur
    - çekirdek düzeyinde işlemek çok yük getirmez çekirdeğe

İpliklerin Çekirdek Uzayında Gerçeklenmesi



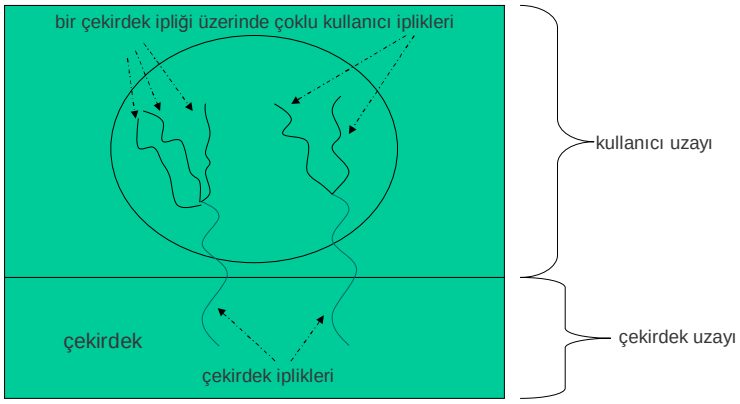
İpliklerin Çekirdek Uzayında Gerçeklenmesi

- çekirdek ipliklerden haberdar
- iplik tablosu çekirdekte
- yeni iplik yaratmak için çekirdeğe sistem çağrısı
- ipliği bloke edebilecek tüm çağrılar çekirdeğe sistem çağrısı
- işletim sistemi hangi ipliğin koşacağına karar verir
  - aynı prosesin ipliği olmayabilir

İpliklerin Çekirdek Uzayında Gerçeklenmesi

- bloke olan sistem çağrılarının yeniden yazılması gerekmez
- sayfa hatası durumu da sorun yaratmaz
  - sayfa hatası olunca çekirdek aynı prosesin koşabilir baka ipliği varsa çalıştırır
- sistem çağrısı gerçekleme ve yürütme maliyetli
  - çok sık iplik yaratma, yoketme, ... işlemleri varsa vakit kaybı çok

İpliklerin Hibrit Yapıda Gerçeklenmesi



İpliklerin Hibrit Yapıda Gerçeklenmesi

- çekirdek sadece çekirdek düzeyi ipliklerden haberdar
- bir çekirdek düzeyi iplik üzerinde birden fazla kullanıcı düzeyi iplik sıra ile çalışır
- kullanıcı düzeyi iplik işlemleri aynı şekilde

4 Prosesler Arası Haberleşme ve Senkronizasyon

Eş Zamanlılık

- Eş zamanlı prosesler olması durumunda bazı tasarım konuları önem kazanır:
  - Prosesler arası haberleşme
  - Kaynak paylaşımı
  - Birden fazla prosesin senkronizasyonu
  - İşlemci zamanı ataması

Sorunlar

Çoklu programlı ve tek işlemcili bir sistemde bir prosesin çalışma hızı öngörülemez:

- Diğer proseslerin yaptıklarına bağlıdır.
- İşletim sisteminin kesmeleri nasıl ele aldığına bağlıdır.
- İşletim sisteminin iş sıralama yaklaşımına bağlıdır.

Sorunlar

- Eş zamanlı çalışan prosesler olması durumunda dikkat edilmesi gereken noktalar:
  - Kaynakların paylaşımı (ortak kullanım)
  - Senkronizasyon

Örnek

- Çoklu programlama, tek işlemci
- **pd** paylaşılan değişken

```
isle()
  begin
     pd = oku();
     pd = pd + 1;
     yazdir(pd);
  end
```

Çözüm

Paylaşılan kaynaklara kontrollü erişim.

Proseslerin Etkileşimi

- Prosesler birbirinden habersizdir.
  - rekabet
- Proseslerin dolaylı olarak birbirlerinden haberleri vardır.
  - Paylaşma yoluyla işbirliği
- Proseslerin doğrudan birbirlerinden haberi vardır.
  - Haberleşme yoluyla işbirliği

Prosesler Arası Rekabet

- Birbirinden habersiz proseslerin aynı kaynağı (örneğin bellek, işlemci zamanı) kullanma istekleri
  - işletim sistemi kullanımı düzenlemeli
- Bir prosesin sonuçları diğerlerinden bağımsız olmalı
- Prosesin çalışma süresi etkilenebilir.

Prosesler Arası Rekabet

- Karşılıklı dışlama
  - Kritik bölge
    - Program kodunun, paylaşılan kaynaklar üzerinde işlem yapılan kısmı.
    - Belirli bir anda sadece tek bir proses kritik bölgesindeki kodu yürütebilir.
- Ölümcül kilitlenme (deadlock)
- Yarış (race)
- Açlık (starvation)

Karşılıklı Dışlama

```
P1()                          P2()
begin                         begin
  <KB olmayan kod>              <KB olmayan kod>
   gir_KB;                       gir_KB;
      <KB işlemleri>                <KB işlemleri>
   cik_KB;                       cik_KB;
  <KB olmayan kod>              <KB olmayan kod>
end                           end
```

- KB: **K**ritik **B**ölge
- İkiden fazla proses de aynı kaynaklar üzerinde çalışıyor olabilir.

Ölümcül Kilitlenme

- Aynı kaynakları kullanan prosesler
- Birinin istediği kaynağı bir diğeri tutuyor ve bırakmıyor
- Proseslerin hiç biri ilerleyemez

  ⇒ ölümcül kilitlenme

PA	PB
al(k1);	al(k2);
al(k2); ⇐ k2'yi bekler	al(k1); ⇐ k1'i bekler
….	….

Yarış

- Aynı ortak verilere erişen prosesler
- Sonuç, proseslerin çalışma hızına ve sıralarına bağlı
- Farklı çalışmalarda farklı sonuçlar üretilebilir

  ⇒ yarış durumu

Yarış

Örnek:

k=k+1 **makina dilinde:**

```
yükle Acc,k
artir Acc
yaz Acc,k
```

P1

```
…
while(TRUE)
  k=k+1;
…
```

P2

```
...
while(TRUE)
  k=k+1;
…
```

Not: k'nın başlangıç değeri 0 olsun. Ne tür farklı çalışmalar olabilir? Neden?

Açlık

- Aynı kaynakları kullanan prosesler
- Bazı proseslerin bekledikleri kaynaklara hiç erişememe durumu
- Bekleyen prosesler sonsuz beklemeye girebilir

  $\Rightarrow$ açlık

Prosesler Arasında Paylaşma Yoluyla İşbirliği

- Paylaşılan değişken / dosya / veri tabanı
  - prosesler birbirlerinin ürettiği verileri kullanabilir
- Karşılıklı dışlama gerekli
- Senkronizasyon gerekebilir
- Sorunlar:
  - ölümcül kilitlenme,
  - yarış
  - açlık

Prosesler Arasında Paylaşma Yoluyla İşbirliği

- İki tür erişim:
  - yazma
  - okuma
- Yazmada karşılıklı dışlama olmalı
- Okuma için karşılıklı dışlama gereksiz
- Veri tutarlılığı sağlanması amacıyla,
  - kritik bölgeler var
  - senkronizasyon

Senkronizasyon

- Proseslerin yürütülme sıraları önceden kestirilemez
- Proseslerin üretecekleri sonuçlar çalışma sıralarına bağlı olmamalıdır
- **Örnek:** Bir P1 prosesi bir P2 prosesinin ürettiği bir sonucu kullanıp işlem yapacaksa, P2'nin işini bitirip sonucunu üretmesini beklemeli

Prosesler Arasında Paylaşma Yoluyla İşbirliği

Örnek: a=b korunacak, başta a=1, b=1

P1: a=a+1;
b=b+1;

P2: b=2*b;
a=2*a;

- Sıralı çalışırsa sonuçta a=4 ve b=4 √

a=a+1;
b=2*b;
b=b+1;
a=2*a;

- Bu sırayla çalışırsa sonuçta a=4 ve b=3 X

Prosesler Arasında Haberleşme Yoluyla İşbirliği

- Mesaj aktarımı yoluyla haberleşme
  - Karşılıklı dışlama gerekli değil
- Ölümcül kilitlenme olabilir
  - Birbirinden mesaj bekleyen prosesler
- Açlık olabilir
  - İki proses arasında mesajlaşır, üçüncü bir proses bu iki prosesten birinden mesaj bekler

Karşılıklı Dışlama İçin Gerekler

- Bir kaynağa ilişkin kritik bölgede sadece bir proses bulunabilir
- Kritik olmayan bölgesinde birdenbire sonlanan bir proses diğer prosesleri etkilememeli
- Ölümcül kilitlenme ve açlık olmamalı

Karşılıklı Dışlama İçin Gerekler

- Kullanan başka bir proses yoksa kritik bölgesine girmek isteyen proses bekletilmemelidir.
- Proses sayısı ve proseslerin bağıl hızları ile ilgili kabuller yapılmamalıdır.
- Bir proses kritik bölgesi içinde sonsuza kadar kalamamalıdır.

Çözümler

- Yazılım çözümleri
- Donanıma dayalı çözümler
- Yazılım ve donanıma dayalı çözümler

Meşgul Bekleme

- Bellek gözü erişiminde bir anda sadece tek bir prosese izin var
- Meşgul bekleme
  - Proses sürekli olarak kritik bölgesine girip giremeyeceğini kontrol eder
  - Proses kritik bölgesine girene kadar başka bir iş yapamaz, bekler.
  - Yazılım çözümleri
  - Donanım çözümleri

Meşgul Bekleme İçin Yazılım Çözümleri - 1

- Meşgul beklemede bayrak/paylaşılan değişken kullanımı
- Karşılıklı dışlamayı garanti etmez
- Her proses bayrakları kontrol edip, boş bulup, kritik bölgesine aynı anda girebilir.

Meşgul Bekleme İçin Yazılım Çözümleri - 2

- Ölümcül kilitlenme (deadlock)
- Ölümcül olmayan kilitlenme
  - proseslerin çalışma hızına göre problem sonsuza kadar devam edebilir
- Açlık (starvation)

Meşgul Bekleme İçin Donanım Çözümleri—1

- Özel makina komutları ile
- Tek bir komut çevriminde gerçekleşen komutlar
- Kesilemezler

`test_and_set` komutu

`exchange` komutu

Donanım Desteği

- Test and Set Instruction

```
boolean testset (int i) {
    if (i == 0) {
        i = 1;
        return true;
    }
    else {
        return false;
    }
}
```

Donanım Desteği

- Exchange Instruction

```
void exchange(int register,
    int memory) {
    int temp;
    temp = memory;
    memory = register;
    register = temp;
}
```


Donanım Desteği

```
/* program mutualexclusion */
const int n = /* number of processes */;
int bolt;
void P(int i)
{
   while (true)
   {
      while (!testset (bolt))
          /* do nothing */;
      /* critical section */;
      bolt = 0;
      /* remainder */
   }
}
void main()
{
   bolt = 0;
   parbegin (P(1), P(2), . . . ,P(n));

}
```

(a) Test and set instruction

```
/* program mutualexclusion */
int const n = /* number of processes*/;
int bolt;
void P(int i)
{
   int keyi;
   while (true)
   {
      keyi = 1;
      while (keyi != 0)
            exchange (keyi, bolt);
      /* critical section */;
      exchange (keyi, bolt);
      /* remainder */
   }
}
void main()
{
   bolt = 0;
   parbegin (P(1), P(2), . . ., P(n));
}
```

(b) Exchange instruction

Figure 5.2 Hardware Support for Mutual Exclusion

Meşgul Bekleme İçin Donanım Çözümleri—2

- Sakıncaları
  - Meşgul bekleme olduğundan bekleyen proses de işlemci zamanı harcar
  - Bir proses kritik bölgesinden çıktığında bekleyen birden fazla proses varsa açlık durumu oluşabilir.
  - Ölümcül kilitlenme riski var

Donanım Desteği ile Karşılıklı Dışlama

- Kesmeleri kapatmak
  - Normal durumda proses kesilene veya sistem çağrısı yapana kadar çalışmaya devam eder.
  - Kesmeleri kapatmak karşılıklı dışlama sağlamış olur
  - Ancak bu yöntem işlemcinin birden fazla prosesi zaman paylaşımlı olarak çalıştırma özelliğine karışmış olur

Makina Komutları ile Karşılıklı Dışlama

- Yararları
  - İkiden fazla sayıda proses için de kolaylıkla kullanılabilir.
  - Basit.
  - Birden fazla kritik bölge olması durumunda da kullanılabilir.

Donanım + Yazılım Desteği ile Karşılıklı Dışlama: Semaforlar

- Prosesler arasında işaretleşme için semafor adı verilen özel bir değişken kullanılır.
- Semafor değişkeninin artmasını bekleyen prosesler askıya alınır.
  - signal(sem)
  - wait(sem)
- *wait* ve *signal* işlemleri kesilemez
- Bir semafor üzerinde bekleyen prosesler bir kuyrukta tutulur

Semaforlar

- Semafor tamsayı değer alabilen bir değişkendir
  - Başlangıç değeri ≥0 olabilir
  - *wait* işlemi semaforun değerini bir eksiltir.
  - *signal* işlemi semaforun değerini bir arttırır.
- Bu iki yol dışında semaforun değerini değiştirmek mümkün değildir.

Semaforlar

- Sadece 0 veya 1 değerini alabilen semaforlara **ikili semafor** adı verilir.
- Herhangi bir tamsayı değeri alabilen semaforlara **sayma semaforu** adı verilir.

Semafor

```
struct semaphore {
     int count;
     queueType queue;
}

void semWait(semaphore s)
{
     s.count--;
     if (s.count < 0)
     {
          place this process in s.queue;
          block this process
     }
}
void semSignal(semaphore s)
{
     s.count++;
     if (s.count <= 0)
     {
          remove a process P from s.queue;
          place process P on ready list;
     }
}
```

Figure 5.3 A Definition of Semaphore Primitives

Binary Semaphore Primitives

```
struct binary_semaphore {
     enum {zero, one} value;
     queueType queue;
};

void semWaitB(binary_semaphore s)
{
     if (s.value ==  1)
          s.value = 0;
     else
          {
               place this process in s.queue;
               block this process;
          }
}
void semSignalB(semaphore s)
{
     if (s.queue.is_empty())
          s.value = 1;
     else
     {
          remove a process P from s.queue;
          place process P on ready list;
     }
}
```

Figure 5.4 A Definition of Binary Semaphore Primitives

Semaforlar ile Karşılıklı Dışlama

```
semafor s = 1;
P1()
begin
  <KB olmayan kod>
  wait(s);
    <KB işlemleri>
  signal(s);
  <KB olmayan kod>
end
```

```
semafor s = 1;
P2()
begin
  <KB olmayan kod>
  wait(s);
    <KB işlemleri>
  signal(s);
  <KB olmayan kod>
end
```

- KB: **K**ritik **B**ölge
- İkiden fazla proses de aynı kaynaklar üzerinde çalışıyor olabilir.

Semaforlar ile Senkronizasyon

```
semafor s = 0;
P1()
begin
  <senkronizasyon noktası
   öncesi işlemleri>
  signal(s);
  <senkronizasyon noktası sonrası
   işlemleri>
end
```

```
semafor s = 0;
P2()
begin
 <senkronizasyon noktası öncesi
  işlemleri>
 wait(s);
 <senkronizasyon noktası sonrası
  işlemleri>
end
```

- İkiden fazla prosesin senkronizasyonu da olabilir.

Örnek: Üretici/Tüketici Problemi

- Bir veya daha fazla sayıda *üretici* ürettikleri veriyi bir tampon alana koyar
- Tek bir *tüketici* verileri birer birer bu tampon alandan alır ve kullanır
- Tampon boyu sınırsız

Örnek: Üretici/Tüketici Problemi

- Belirli bir anda sadece bir üretici veya tüketici tampon alana erişebilir

  $\Rightarrow$ karşılıklı dışlama
- Hazır veri yoksa, tüketici bekler

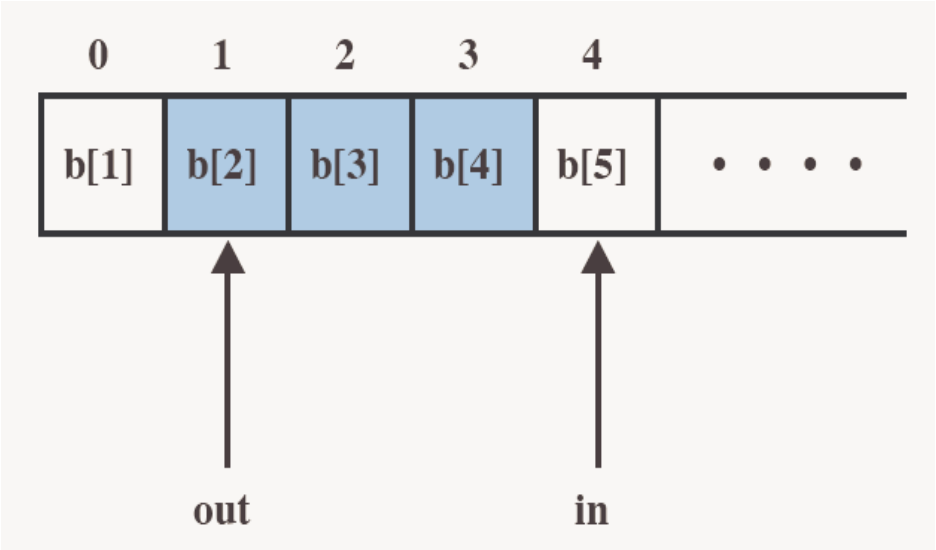
  $\Rightarrow$ senkronizasyon

Üretici / Tüketici Problemi

```
producer:
while (true) {
   /* produce item v */
   b[in] = v;
   in++;
}
```

```
consumer:
while (true) {
   while (in <= out)
        /*do  nothing */;
   w = b[out];
   out++;
   /* consume item w */
}
```


Producer/Consumer Problem

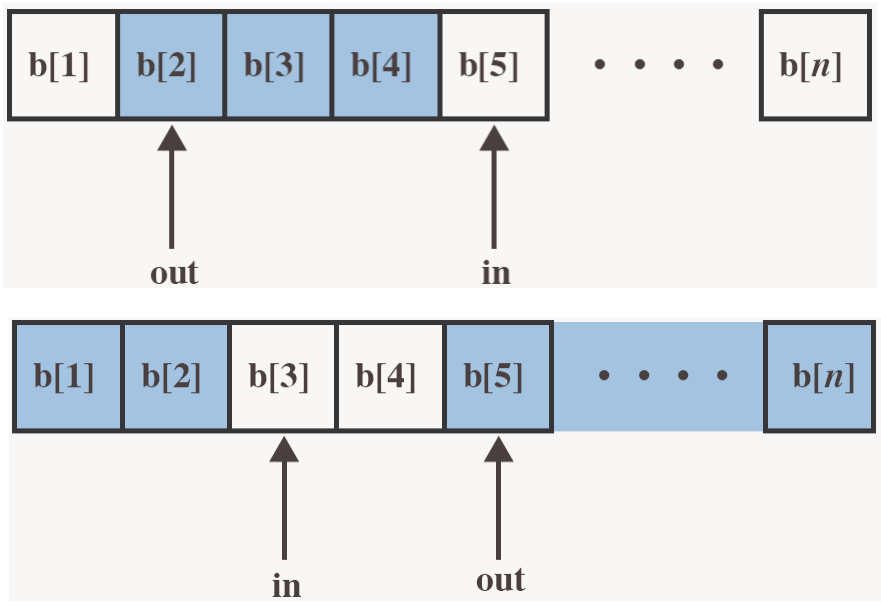


Producer with Circular Buffer

```
producer:
while (true) {
  /* produce item v */
  while ((in + 1) % n == out)  /* do
  nothing */;
  b[in] = v;
  in = (in + 1) % n
}
```


Consumer with Circular Buffer

```
consumer:
while (true) {
  while (in == out)
     /* do nothing */;
  w = b[out];
  out = (out + 1) % n;
  /* consume item w */
}
```



Üretici / Tüketici Problemi (Sonsuz Büyüklükte Tampon)

```
semafor s=1;
semafor n=0;
uretici()
begin
  while(true)
  begin
    uret();
    wait(s);
    tampona_ekle();
    signal(s);
    signal(n);
  end
end
```

```
semafor s=1;
semafor n=0;
tuketici()
begin
  while(true)
  begin
    wait(n);
    wait(s);
    tampondan_al();
    signal(s);
    tuket();
  end
end
```

Not: Birden fazla üretici prosesi çalışabilir. Tüketici prosesi tek.

Örnek: Okuyucu / Yazıcı Problemi

- Birden fazla okuyucu dosyadan okuma yapabilir.
- Bir anda sadece bir yazıcı dosyaya yazma yapabilir ⇒ karşılıklı dışlama
- Bir yazıcı dosyaya yazıyorsa okuyucu aynı anda okuyamaz ⇒ karşılıklı dışlama

LINUX'da Semafor İşlemleri

- Semafor ile ilgili tutulan bilgiler:
  - semaforun değeri
  - semaforun =0 olmasını bekleyen proses sayısı
  - semaforun değerinin artmasını bekleyen proses sayısı
  - semafor üzerinde işlem yapan son prosesin kimliği (pid)

LINUX'da Semafor İşlemleri

- Başlık dosyaları:
  - sys/ipc.h
  - sys/sem.h
  - sys/types.h
- Yaratma

```
int semget(key_t key, int nsems,int semflg);
semflag : IPC_CREAT|0700
```


LINUX'da Semafor İşlemleri

- İşlemler

```
int semop(int semid, struct sembuf *sops,
        unsigned nsops);
struct sembuf{
   ...
   unsigned short sem_num; /*numaralama 0 ile baslar*/
   short sem_op;
   short sem_flg;
};
sem_flg:        SEM_UNDO (proses sonlanınca işlemi geri al)
                IPC_NOWAIT (eksiltemeyince hata ver ve dön)
sem_op : =0     sıfır olmasını bekle (okuma hakkı olmalı)
         ≠0     değer semafor değerine eklenir (değiştirme
        hakkı olmalı)
```

LINUX'da Semafor İşlemleri

- Değer kontrolü

int semctl(int semid, int semnum,int cmd, arg);

cmd: IPC_RMID
GETVAL
SETVAL

LINUX'da Semafor İşlemleri

- Eksiltme işlemi gerçeklemesi

```
void sem_wait(int semid, int val)
{
  struct sembuf semafor;

  semafor.sem_num=0;
  semafor.sem_op=(-1*val);
  semafor.sem_flg=0;
  semop(semid, &semafor,1);
}
```

LINUX'da Semafor İşlemleri

- Artırma işlemi gerçeklemesi

```
void sem_signal(int semid, int val)
{
  struct sembuf semafor;

  semafor.sem_num=0;
  semafor.sem_op=val;
  semafor.sem_flg=0;
  semop(semid, &semafor,1);
}
```


Linux'da Semafor İşlemleri Örneği-1

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <signal.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#define SEMKEY 1234

int sonsem;

void signal12(void)
{
}
```

Linux'da Semafor İşlemleri Örneği - 2

```
void sem_signal(int semid, int val){
  struct sembuf semafor;
  semafor.sem_num=0;
  semafor.sem_op=val;
  semafor.sem_flg=0;
  semop(semid, &semafor,1);
}

void sem_wait(int semid, int val)
{
  struct sembuf semafor;
  semafor.sem_num=0;
  semafor.sem_op=(-1*val);
  semafor.sem_flg=0;
  semop(semid, &semafor,1);
}
```


Linux'da Semafor İşlemleri Örneği - 3

```
int main(void)
{
  int f;

  sem=semget(SEMKEY, 1, 0700|IPC_CREAT);
  semctl(sem, 0, SETVAL,0);
  f=fork();

  if (f==-1)
  {
    printf("fork error\n");
    exit(1);
  }
```

Linux'da Semafor İşlemleri Örneği - 4

```
if (f>0) /*anne */
{
  printf("Anne calışmaya başladı…\n");
  sem_wait(sem,10);

  printf("cocuk semaforu artırdı\n");
  semctl(sem,0,IPC_RMID,0);
}
```


Linux'da Semafor İşlemleri Örneği - 5

```
else /*cocuk */
  {
    printf("  Cocuk çalışmaya başladı…\n");
    sem=semget(SEMKEY, 1,0);

    sem_signal(sem,1);
    printf("  Cocuk: semafor değeri = %d\n",
      semctl(sonsem,0,GETVAL,0));
  }
  return(0);
}
```

Linux'da Sinyal Mekanizması

- sinyaller asenkron işaretlerdir
  - işletim sistemi prosese yollayabilir
  - bir proses bir başka prosese yollayabilir
- sinyal alınınca yapılacak işler tanımlanır
- öntanımlı işleri olan sinyaller var
  - öntanımlı işler değiştirilebilir
  - SIGKILL (9 no.lu) sinyali yakalanamaz

Linux'da Sinyal Mekanizması

- Başlık dosyası:
  - `sys/types.h`
  - `signal.h`
- bir sinyal alınınca yapılacak işin bildirilmesi:

```
#typedef void (sighandler_t  *)(int);
sighandler_type signal(int signum,
                       sighandler_t sighandler);
sighandler:   SIG_IGN
              SIG_DFL
              kullanıcı tarafından tanımlanır
```

Linux'da Sinyal Mekanizması

- bir prosesten diğerine sinyal yollama:

```
int kill(pid_t pid,  int sig);
```

- bir sinyal bekleme:
  - başlık dosyası: `unistd.h`

```
int pause(void);
```


Linux'da Sinyal Mekanizması Örnek-1


```
#include <stdio.h>
#include <sys/types.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>

void signal12(void)
{
    printf("12 numarali sinyali aldim. \n");
    exit(0);
}
```

Linux'da Sinyal Mekanizması Örnek - 2

```
int main (void)
{
  int f;

  signal(12, (void *)signal12);

  f=fork();
  if (f==-1)
  {
    printf("fork hatasi\n");
    exit(1);
  }
```


Linux'da Sinyal Mekanizması Örnek - 3

```
 else if (f==0)  /*cocuk*/
  {
    printf("  COCUK: basladi\n");
    pause();
  }
  else  /*anne*/
  {
    printf("ANNE: basliyorum...\n");
    sleep(3);
    printf("ANNE: cocuga sinyal yolluyorum\n");
    kill(f,12);
    exit(0);
  }
  return 0;
}
```

Linux'da Paylaşılan Bellek Mekanizması

- Birden fazla proses tarafından ortak kullanılan bellek bölgeleri
- Prosesin adres uzayına eklenir
- Başlık dosyaları:
  - sys/ipc.h
  - sys/shm.h
  - sys/types.h

Linux'da Paylaşılan Bellek Mekanizması

- Paylaşılan bellek bölgesi yaratma

```
int shmget(key_t key, int size, int shmflag);
shmflag: IPC_CREAT|0700
```

- Adres uzayına ekleme

```
void *shmat(int shmid,const *shmaddr,int shmflg);
```

- Adres uzayından çıkarma

```
int shmdt(const  void *shmaddr);
```

- Sisteme geri verme

```
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
cmd: IPC_RMID
```

Linux'da Paylaşılan Bellek Mekanizması

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define SHMKEY 5678
```

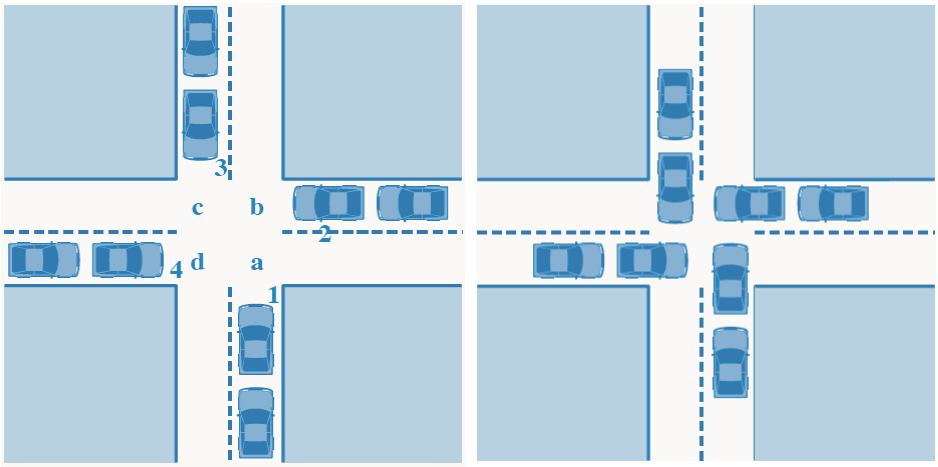

Linux'da Paylaşılan Bellek Mekanizması

```
int main (void)
{
  int *pb, pbid, i;
  pbid=shmget(SHMKEY, sizeof(int),0700|IPC_CREAT);
  pb=(int *)shmat(pbid,0,0);
  *pb=0;
  for (i=0;i<10;i++)
  {
   (*pb)++;
    printf("yeni deger %d\n", (*pb));
  }
  shmdt((void *)pb);
  return 0;
}
```

5 ÖLÜMCÜL KİLİTLENME

Ölümcül Kilitlenme

- Sistem kaynaklarını ortak olarak kullanan veya birbiri ile haberleşen bir grup prosesin kalıcı olarak bloke olması durumu : *ölümcül kilitlenme*
- Birden fazla proses olması durumunda proseslerin bir kaynağı ellerinde tutmaları ve bir başka kaynağı istemeleri durumunda ölümcül kilitlenme olası.
- Etkin bir çözüm yok

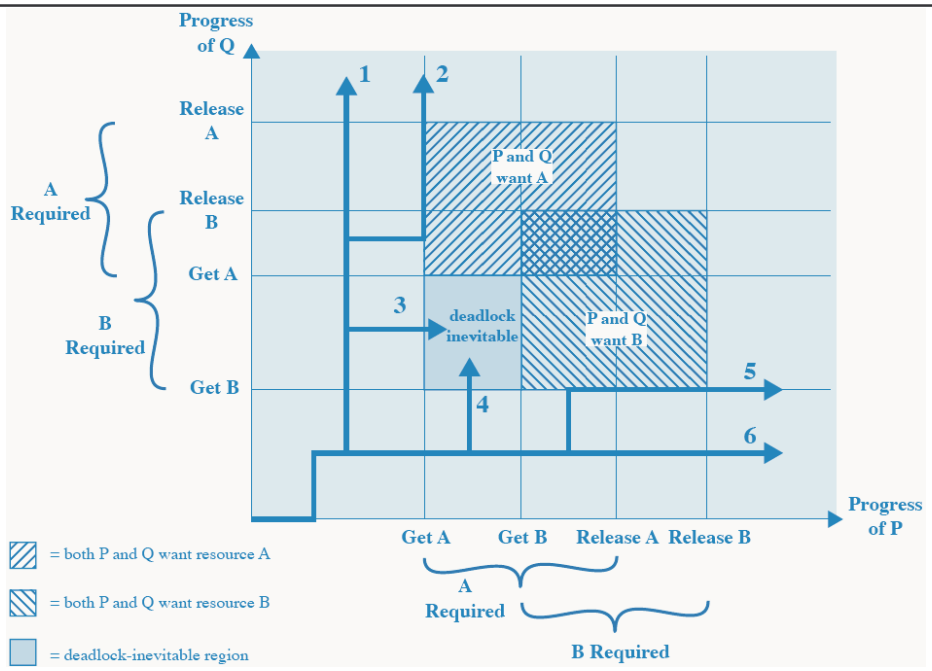


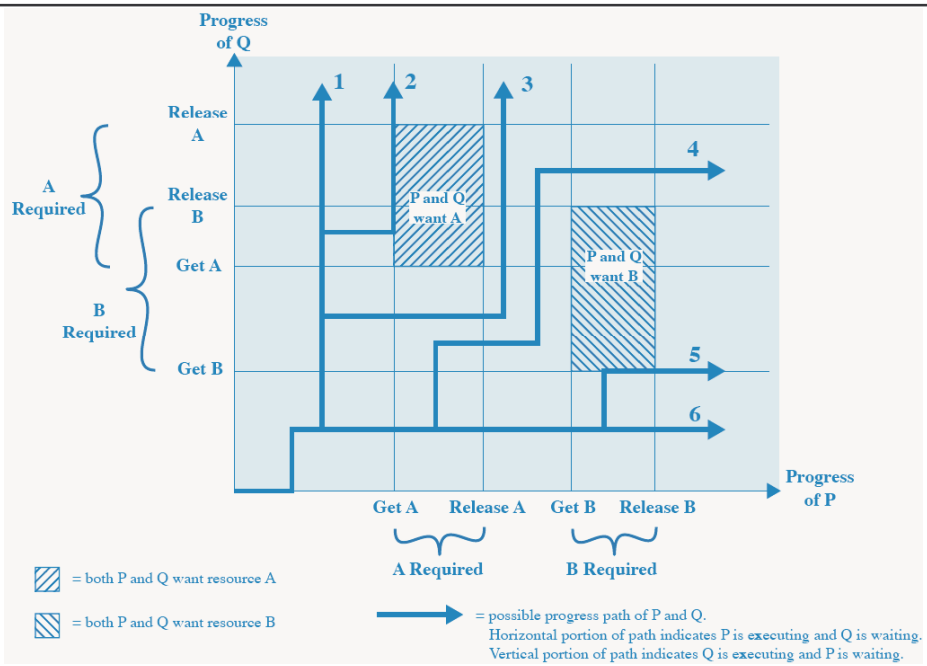
a) Olası ölümcül kilitlenme b) Ölümcül kilitlenme

Ölümcül Kilitlenme Örneği - 1

Process P	
Step	**Action**
p_0	Request (D)
p_1	Lock (D)
p_2	Request (T)
p_3	Lock (T)
p_4	Perform function
p_5	Unlock (D)
p_6	Unlock (T)

Process Q	
Step	**Action**
q_0	Request (T)
q_1	Lock (T)
q_2	Request (D)
q_3	Lock (D)
q_4	Perform function
q_5	Unlock (T)
q_6	Unlock (D)


Progress of Q
Release A
Release B
Get A
Get B
A Required
B Required
1
2
3
4
5
6
P and Q want A
deadlock inevitable
P and Q want B
Progress of P
Get A
Get B
Release A
Release B
A Required
B Required
= both P and Q want resource A
= both P and Q want resource B
= deadlock-inevitable region


Progress of Q
Release A
Release B
Get A
Get B
A Required
B Required
1
2
3
4
5
6
P and Q want A
P and Q want B
Progress of P
Get A
Release A
Get B
Release B
A Required
B Required
= both P and Q want resource A
= both P and Q want resource B
= possible progress path of P and Q.
Horizontal portion of path indicates P is executing and Q is waiting.
Vertical portion of path indicates Q is executing and P is waiting.

Ölümcül Kilitlenme Örneği - 2

- 200K sekizlilik bir bellek bölgesi proseslere atanabilir durumda ve aşağıdaki istekler oluşuyor:
- Her iki proses de ilk isteklerini aldıktan sonra ikinci isteklerine devam ederlerse kilitlenme oluşur.

P1 Prosesi	**P2 Prosesi**
80K sekizli iste	70K sekizli iste
...	...
60K sekizli iste	80K sekizli iste

Ölümcül Kilitlenme Örneği - 3

- Mesaj alma komutu bloke olan türden ise ölümcül kilitlenme olabilir.

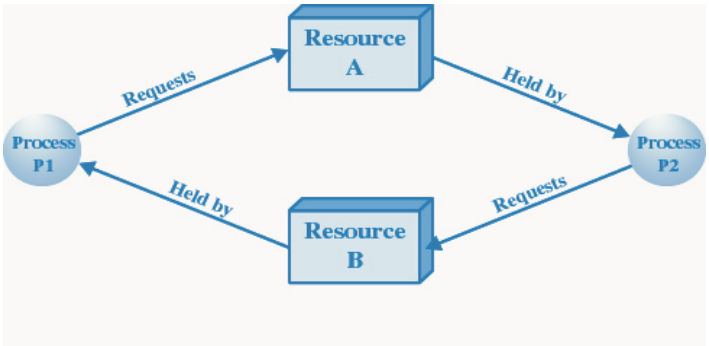
P1 Prosesi	**P2 Prosesi**
Al (P2);	Al (P1);
...	...
Gönder (P2,M1);	Gönder (P1,M2);

Ölümcül Kilitlenme Olması İçin Gereken Koşullar

- Karşılıklı dışlama
- Proseslerin ellerine geçirdikleri kaynakları diğer istedikleri kaynakları da ele geçirene kadar bırakmamaları
- Proseslerin ellerinde tuttukları kaynaklar işletim sistemi tarafından zorla geri alınamıyorsa (“pre-emption” yok)

Ölümcül Kilitlenme Olması İçin Gereken Koşullar

- Çevrel bekleme durumu oluşuyorsa



Kaynak Atama Grafı

- Yönlü bir graf
- Kaynakların ve proseslerin durumunu gösterir
- Prosesler P_i ve kaynaklar K_j olsun.
  - $R_3 \rightarrow P_5$: Bir adet R_3 kaynağının P_5 prosesine atanmış olduğunu gösterir: ***atama kenarı***
  - $P_2 \rightarrow R_1$: P_2 prosesi bir adet R_1 kaynağı için istekte bulunmuştur: ***istek kenarı***

Kaynak Atama Grafı

Örnek:

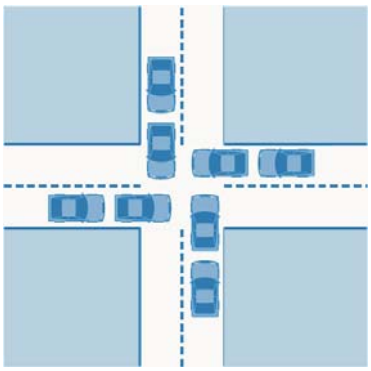
P, R ve E kümeleri olsun.

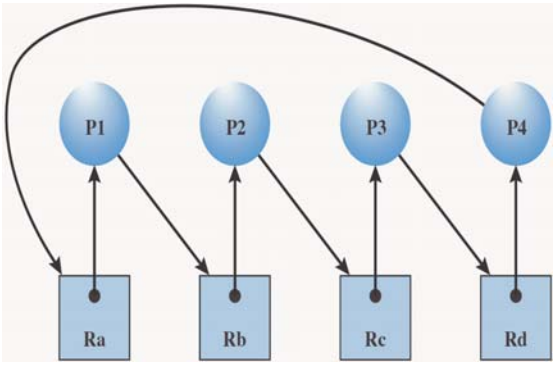
- $P=\{P_1,P_2,P_3\}$
- $R=\{R_1,R_2,R_3,R_4\}$
- $E=\{P_1\rightarrow R_1,\ P_2\rightarrow R_3,\ R_1\rightarrow P_2,\ R_2\rightarrow P_2,\ R_2\rightarrow P_1,\ R_3\rightarrow P_3\}$

Kaynaklar:
- 1 adet R_1
- 2 adet R_2
- 1 adet R_3
- 3 adet R_4

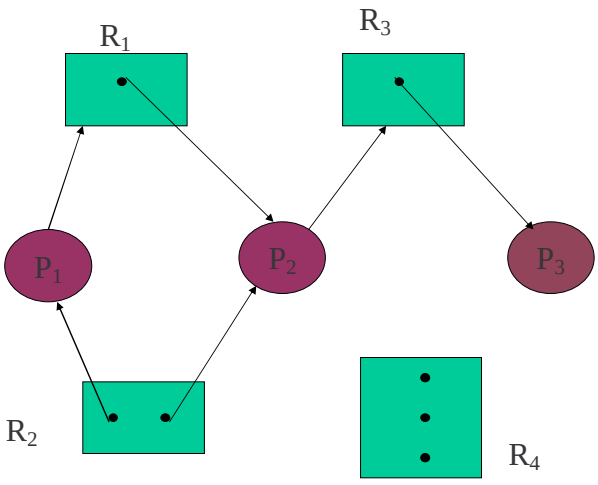
Proses Durumları:
- P_1: bir adet R_2 elinde tutuyor, bir adet R_1 istiyor
- P_2: birer adet R_1 ve R_2 elinde tutuyor, bir adet R_3 istiyor
- P_3: bir adet R_3 elinde tutuyor

Örnek

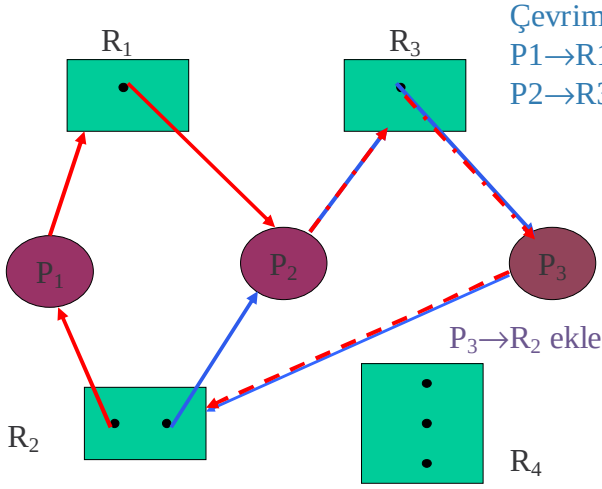




Kaynak Atama Grafı



Kaynak Atama Grafı

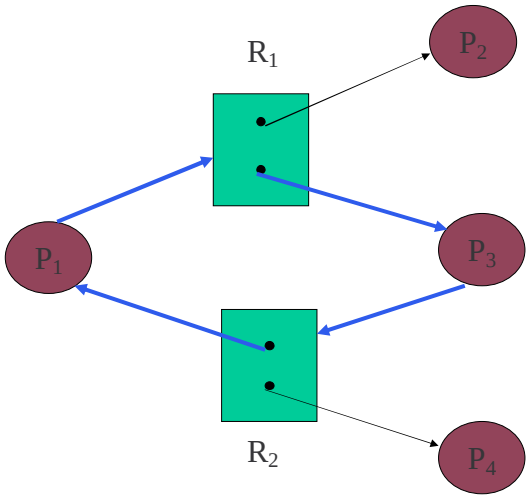


Çevrimler:
P1→R1→P2→R3→P3→R2→P1
P2→R3→P3→R2→P2

$P_3 \rightarrow R_2$ eklensin.

P_1, P_2 ve P_3 Ölümcül kilitlenme durumundalar!

Kaynak Atama Grafı



Ölümcül kilitlenme durumu yok.
NEDEN?

Ölümcül Kilitlenme Durumunda Kullanılan Yaklaşımlar

- Sistemin hiçbir zaman ölümcül kilitlenme durumuna girmemesini sağlamak
- Sistemin, ölümcül kilitlenme durumuna girdikten sonra bu durumdan kurtulmasını sağlamak
- Problemi gözardı edip sistemde ölümcül kilitlenme olmayacağını varsaymak

Ölümcül Kilitlenme Durumunda Kullanılan Yaklaşımlar

- Sistemin hiçbir zaman ölümcül kilitlenme durumuna girmemesini sağlamak
  - ölümcül kilitlenmeyi önlemek
  - ölümcül kilitlenmeden kaçınmak

Ölümcül Kilitlenmeyi Önleme

- Ölümcül kilitlenmenin oluşmasının mümkün olmadığı bir sistem tasarlanması
  - Dört gerekli koşuldan en az birinin geçerli olmaması

Ölümcül Kilitlenmeyi Önleme

- **Karşılıklı Dışlama:**
  - Paylaşılan kaynaklar olması durumunda bu koşulun engellenmesi mümkün değil.
- **Tut ve Bekle:**
  - Kaynak isteyen prosesin elinde başka k aynak tutmuyor olmasının sağlanması
  - Proseslerin tüm kaynak isteklerini baştan belirtmeleri ve tüm istekleri birden karşılanana kadar proseslerin bekletilmesi yaklaşımı ile bu koşul engellenebilir.

Ölümcül Kilitlenmeyi Önleme

- Etkin değil:
  - Proses tüm istekleri karşılanana kadar çok bekleyebilir, halbuki bir kısım kaynağını ele geçirerek işinin bir bölümünü bitirebilir.
  - Bir prosese atanan kaynakların bir kısmı bir süre kullanılmadan boş bekleyebilir.
- Proses tüm kaynak ihtiyaçlarını baştan bilemeyebilir.

Ölümcül Kilitlenmeyi Önleme

- Diğer yöntem: Bir prosesin yeni bir kaynak isteğinde bulunduğunda, kaynak ataması yapılmadan önce elindeki tüm kaynakları bırakmasının beklenmesi:
  - kaynak kullanımı etkin değil
  - tutarlılık sorunu olabilir
- Her iki yöntemde de açlık (starvation) olası.

Ölümcül Kilitlenmeyi Önleme

Örnek:

Teyp sürücüsündeki verileri okuyup diskte bir dosyaya yazan, sonra diskteki dosyadaki bilgileri sıralayıp, yazıcıya bastıran proses.

- Her iki yöntemin farkı ne?
- Her iki yöntemin sorunları ne?

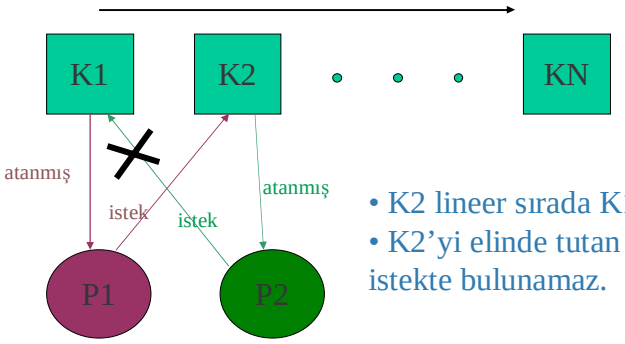
Ölümcül Kilitlenmeyi Önleme

- **Pre-Emption Olmaması Koşulu :**
  - Elinde bazı kaynakları tutan bir prosesin yeni bir isteği karşılanamıyorsa elindekileri de bırakması ve gerekiyorsa yeni kaynakla birlikte tekrar istemesi ile bu koşul engellenebilir.
  - Bir proses bir başka proses tarafından elinde tutulan bir kaynak isterse, ikinci prosesin kaynaklarını bırakması işletim sistemi tarafından sağlanabilir. (*Bu yöntem proseslerin eşit öncelikli olmadıkları durumda uygulanabilir.*)
  - Bu yaklaşım durumları saklanabilen ve yeniden yüklenebilen kaynaklar için uygun.

Ölümcül Kilitlenmeyi Önleme

- **Çevrel Bekleme Koşulu:**
  - Kaynak tipleri arasında lineer bir sıralama belirleyerek önlenebilir.
    - Örneğin elinde R tipi kaynaklar tutan bir proses sadece lineer sıralamada R'yi izleyen kaynakları alabilir.
  - Bu yöntem de prosesleri yavaşlatmasının ve kaynakları gereksiz yere boş bekletmesinin olası olması nedeniyle etkin değil

Ölümcül Kilitlenmeyi Önleme



- K2 lineer sırada K1'den sonra.
- K2'yi elinde tutan P2, K1 için istekte bulunamaz.

Ölümcül Kilitlenmeden Kaçınma

- Yeni bir kaynak isteği geldiğinde, bu isteğin karşılanmasının bir kilitlenmeye neden olup olamayacağı dinamik olarak belirlenir.
- Proseslerin gelecekteki kaynak istekleri (ne zaman ve hangi sırayla) ve kaynakları geri verme anları hakkında önceden bilgi gerektirir.

Ölümcül Kilitlenmeden Kaçınma

- Temel olarak iki yöntem var:
  - Eğer istekleri bir ölümcül kilitlenmeye yol açabilecekse prosesi başlatma
  - Eğer istek bir ölümcül kilitlenmeye yol açabilecekse prosesin ek kaynak isteklerini karşılama.

Prosesin Başlamasının Önlenmesi

- n adet proses ve m adet farklı tip kaynak olsun.
- Sistemdeki tüm kaynaklar:

  Kaynak=(R1, R2, ..., Rm)
- Sistemdeki boş kaynaklar:

  Boş=(V1, V2, ..., Vm)

Prosesin Başlamasının Önlenmesi

- Proseslerin her kaynak için maksimum istek matrisi:

$$\text{İstek}=\begin{bmatrix} C_{11} & C_{12} & \ldots\ldots & C_{1m} \\ C_{21} & C_{22} & \ldots\ldots & C_{2m} \\ & \ldots\ldots\ldots\ldots & & \\ C_{n1} & C_{n2} & \ldots\ldots & C_{nm} \end{bmatrix}$$

Prosesin Başlamasının Önlenmesi

- Proseslere şu anda her kaynaktan atanmış olanların gösterildiği matris:

$$\text{Atanmış} = \begin{bmatrix} A11 & A12 & \dots & A1m \\ A21 & A22 & \dots & A2m \\ & \dots\dots\dots & & \\ An1 & An2 & \dots & Anm \end{bmatrix}$$

Prosesin Başlamasının Önlenmesi

- Tüm kaynaklar ya boştur ya da kullanılmaktadır.
- Prosesler sistemde bulunandan daha fazla kaynak isteğinde bulunamaz.
- Prosese, ilk başta belirttiğinden fazla kaynak ataması yapılmaz.

Prosesin Başlamasının Önlenmesi

- P_{n+1} prosesini ancak ve ancak aşağıdaki koşul gerçeklendiğinde başlat:

$$R_i \geq C_{(n+1)i} + \sum_{k=1}^{n} C_{ki} \quad \text{, tüm i'ler için}$$

- Optimal değil. Tüm proseslerin maksimum kaynak gereksinimlerini birden isteyeceğini varsayar.

Kaynak Atamasının Engellenmesi

- **Banker algoritması**
- Sistemde sabit sayıda proses ve kaynak var.
- ***Sistemin durumu*:** Kaynakların proseslere o anki ataması $\Rightarrow$ durum ***Kaynak*** ve ***Boş*** vektörlerinden ve ***Atanmış*** ve ***İstek*** matrislerinden oluşur.
- ***Emin durum*:** Ölümcül kilitlenmeye yol açmayacak en az bir atama sekansı olduğu durum (yani tüm proseslerin sonlanabileceği durum)
- ***Emin olmayan durum***

Emin Durumun Belirlenmesi
Başlangıç Durumu

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	2	2	2
P2	0	0	1
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
0	1	1

Available vector **V**

Bu emin bir durum mu?

Emin Durumun Belirlenmesi
P2 Sonlanır

	R1	R2	R3
P1	3	2	2
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	1	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	2	2	2
P2	0	0	0
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
6	2	3

Available vector **V**

Emin Durumun Belirlenmesi
P1 Sonlanır

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
7	2	3

Available vector **V**

Emin Durumun Belirlenmesi
P3 Sonlanır

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
9	3	4

Available vector **V**

En son olarak da P4 sonlanır $\Rightarrow$ Emin Durum $\surd$

Ölümcül Kilitlenmeden Kaçınma

- Bir proses bir grup kaynak için istekte bulunduğunda işletim sistemi,
  - Kaynakların atandığını varsayarak sistem durumunu günceller.
  - Yeni durum emin bir durum mu diye kontrol eder.
  - Emin ise atamayı gerçekleştirir.
  - Emin değilse atamayı yapmaz ve istekte bulunan prosesi isteklerin karşılanması uygun olana kadar bloke eder.

Emin Olmayan Durumun Belirlenmesi: Başlangıç Durumu

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	1	0	0
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	2	2	2
P2	1	0	2
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
1	1	2

Available vector **V**

P2 bir adet R1 ve bir adet de R3 kaynağı için daha istekte bulunursa ne olur?

Emin Olmayan Durumun Belirlenmesi

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	2	0	1
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	1	2	1
P2	1	0	2
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
0	1	1

Available vector **V**

P1 bir adet R1 ve bir adet de R3 kaynağı için daha istekte bulunursa ne olur?

Emin Olmayan Durumun Belirlenmesi

- Emin bir durum değil. Bu nedenle P1'in yeni istekleri karşılanmaz ve P1 bloke edilir.
- Oluşan durum ölümcül kilitlenme durumu değildir sadece ölümcül kilitlenme oluşması potansiyelini taşımaktadır.

Ölümcül Kilitlenmeden Kaçınma

- Kullanımındaki kısıtlamalar:
  - Her prosesin maksimum kaynak ihtiyacı önceden belirtilmeli.
  - Prosesler birbirlerinden bağımsız olmalı. Koşma sıralarının önemi olmamalı.
  - Kaynak sayısı sabit olmalı.
  - Elinde kaynak tutan proses kaynakları bırakmadan sonlanmamalı.

Ölümcül Kilitlenmeyi Sezme

- Ölümcül kilitlenmeyi önleme yöntemleri kadar kısıtlayıcı değil.
- Tüm kaynak istekleri karşılanır. İşletim sistemi periyodik olarak sistemde çevrel bekleme durumu oluşup oluşmadığını kontrol eder.
- Kontrol sıklığı ölümcül kilitlenme oluşması sıklığına bağlıdır: Her yeni kaynak isteğinde bile yapılabilir.

Ölümcül Kilitlenmeyi Sezme

- Ölümcül kilitlenmenin sezilmesi için kullanılan algoritmada *Atanmış* matrisi ve *Boş* vektörü kullanılıyor. Q istek matrisi tanımlanıyor. (Burada q_{ij}: i. prosesinin j tipi kaynaktan kaç tane istediği.)
- Algoritma kilitlenmemiş prosesleri belirleyip işaretler.
- Başlangıçta tüm prosesler işaretsizdir. Aşağıdaki adımlar gerçekleştirilir:

Ölümcül Kilitlenmeyi Sezme

- **Adım 1:** Atanmış matrisinde bir satırının tamamı sıfır olan prosesleri işaretle.
- **Adım 2:** *Boş* vektörüne karşılık düşecek bir W geçici vektörü oluştur.
- **Adım 3:** Q matrisinin i. satırındaki tüm değerlerin W vektöründeki değerlerden küçük ya da eşit olduğu bir i belirle (P_i henüz işaretlenmemiş olmalı).

$$Q_{ik} \leq W_k \ , \ 1 \leq k \leq m$$

Ölümcül Kilitlenmeyi Sezme

- **Adım 4:** Böyle bir satır bulunamıyorsa algoritmayı sonlandır.
- **Adım 5:** Böyle bir satır bulunduysa i prosesini işaretle ve *Atanmış* matrisinde karşılık düşen satırı W vektörüne ekle.

  $W_k = W_k + A_{ik}$, $1 \le k \le m$
- **Adım 6:** Adım 3'e dön.
- Algoritma sonlandığında işaretsiz proses varsa ⇒ Ölümcül kilitlenme var.

Ölümcül Kilitlenmeyi Sezme

- Algoritma sonlandığında işaretsiz proses varsa ⇒ Ölümcül kilitlenme var.
- İşaretsiz prosesler kilitlenmiş.
- Temel yaklaşım, mevcut kaynaklarla istekleri karşılanabilecek bir proses bulmak, kaynakları atamak ve o proses koşup sonlandıktan sonra bir başka prosesin aranmasıdır.
- Algoritma sadece o an ölümcül kilitlenme olup olmadığını sezer.

Ölümcül Kilitlenmeyi Sezme

	R1	R2	R3	R4	R5
P1	0	1	0	0	1
P2	0	0	1	0	1
P3	0	0	0	0	1
P4	1	0	1	0	1

Request matrix **Q**

	R1	R2	R3	R4	R5
P1	1	0	1	1	0
P2	1	1	0	0	0
P3	0	0	0	1	0
P4	0	0	0	0	0

Allocation matrix **A**

R1	R2	R3	R4	R5
2	1	1	2	1

Resource vector

R1	R2	R3	R4	R5
0	0	0	0	1

Available vector

1) P4'ü işaretle.
2) W = (0 0 0 0 1)
3) P3'ün isteği W'dan küçük veya eşit ⇒ P3 işaretle.
   W = W + (0 0 0 1 0) = (0 0 0 1 1)
4) İşaretsiz hiçbir proses için Q'daki ilgili satır W'dan küçük ya da eşit değil ⇒ Algoritmayı sonlandır.

Sonuçta P1 ve P2 işaretsiz ⇒ kilitlenmişler

Ölümcül Kilitlenme Sezildikten Sonra Yapılabilecekler

- Tüm kilitlenmiş prosesleri sonlandır.
- Tüm kilitlenmiş proseslerin eski bir kontrol noktasına kadar kopyasını al ve tüm prosesleri bu noktadan yeniden başlat.
  - aynı ölümcül kilitlenme yeniden oluşabilir
- Kilitlenme ortadan kalkana kadar sırayla kilitlenmiş prosesleri sonlandır.
- Kilitlenme ortadan kalkana kadar, sırayla atanmış kaynakları geri al.

Kilitlenmiş Prosesler İçin Seçim Kriterleri

- O ana kadar en az işlemci zamanı kullanmış olan
- O ana kadar en az sayıda çıktı satırı oluşturmuş olan
- Beklenen çalışma süresi en uzun olan
- O ana kadar en az kaynak atanmış olan
- En düşük öncelikli olan

Makarnacı Düşünürler Problemi