İşletim Sistemleri

Binnur Kurt
binnur.kurt@ieee.org

İstanbul Teknik Üniversitesi
Bilgisayar Mühendisliği Bölümü

Version 0.0.8

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

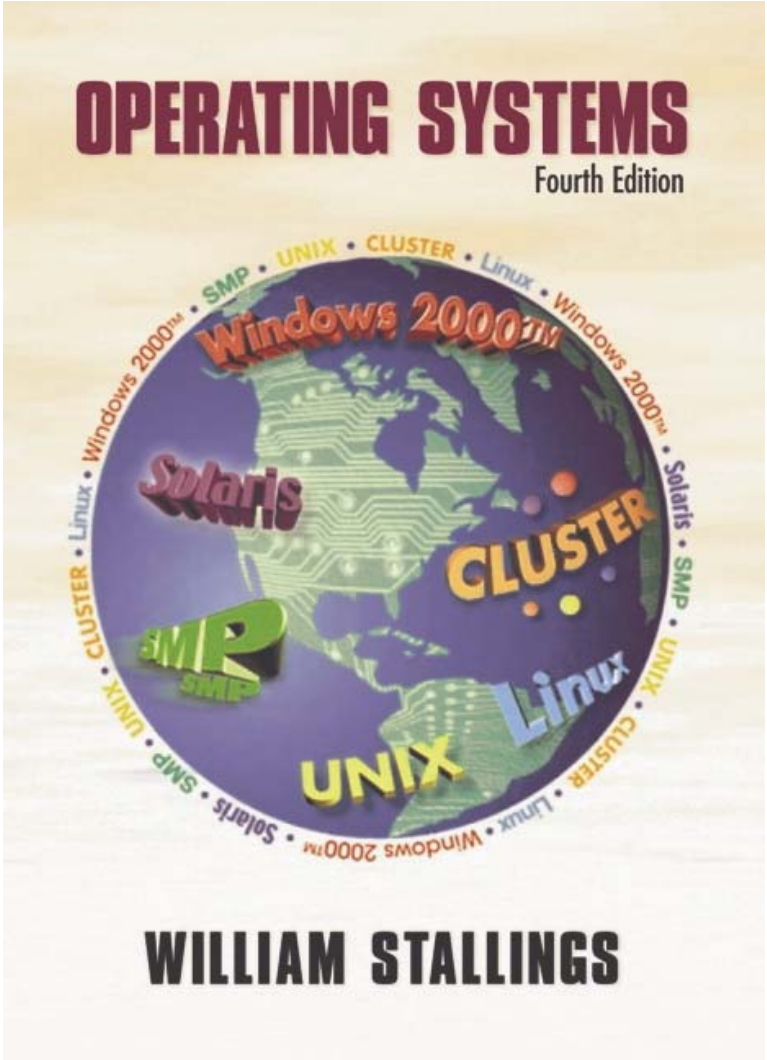
Önemli Bilgiler

- **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


OPERATING SYSTEMS
Fourth Edition
Windows 2000™
Solaris
CLUSTER
SMP
Linux
UNIX
WILLIAM STALLINGS

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

—Chinese Proverb

İçerik

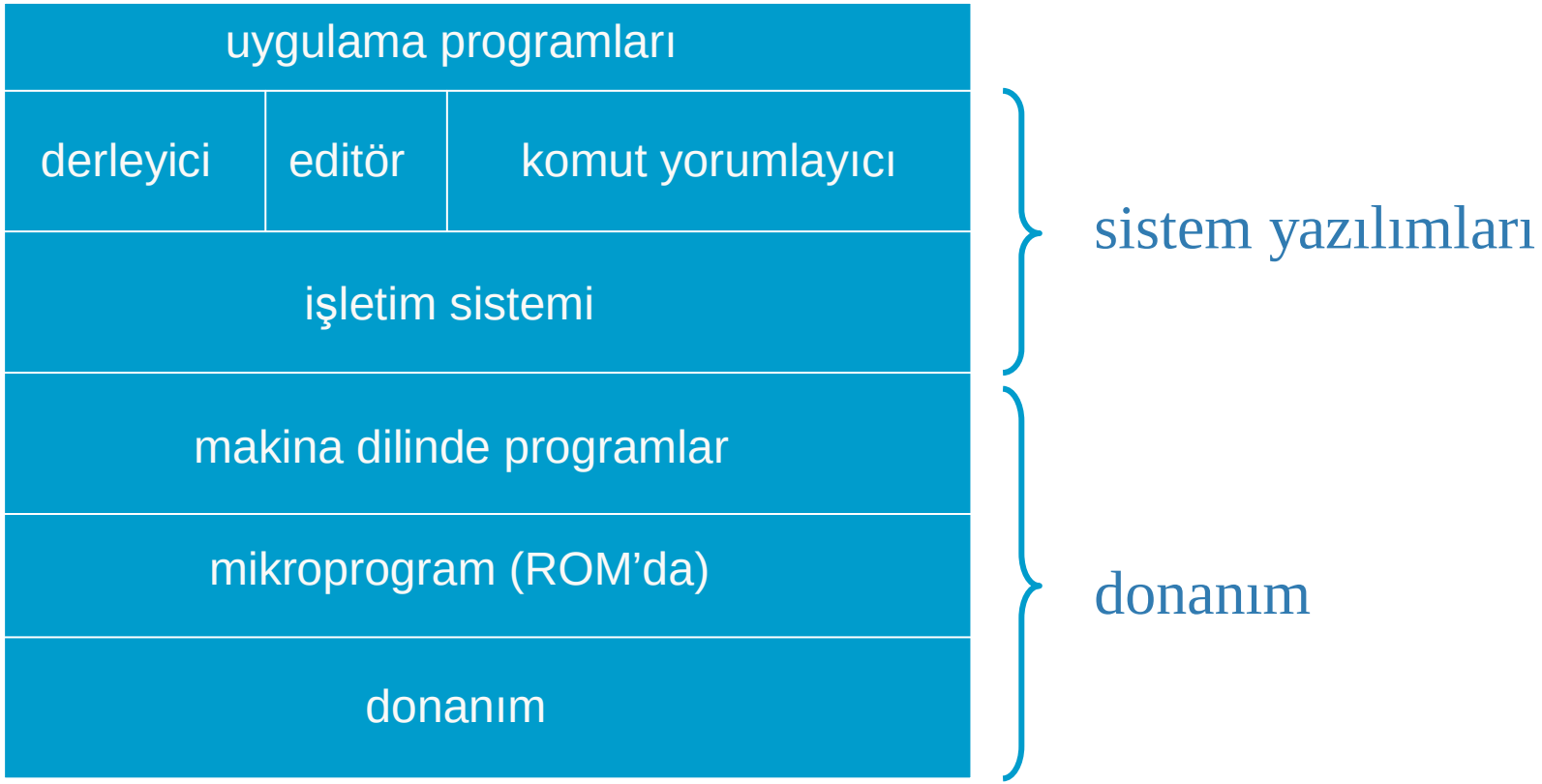
1. Giriş.
2. Prosesler ve Proses Kontrolü.
3. İplikler
4. Prosesler Arası İletişim
5. Ölümcül Kilitlenme
6. İş Sıralama
7. Bellek Yönetimi
8. Dosya Sistemi

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

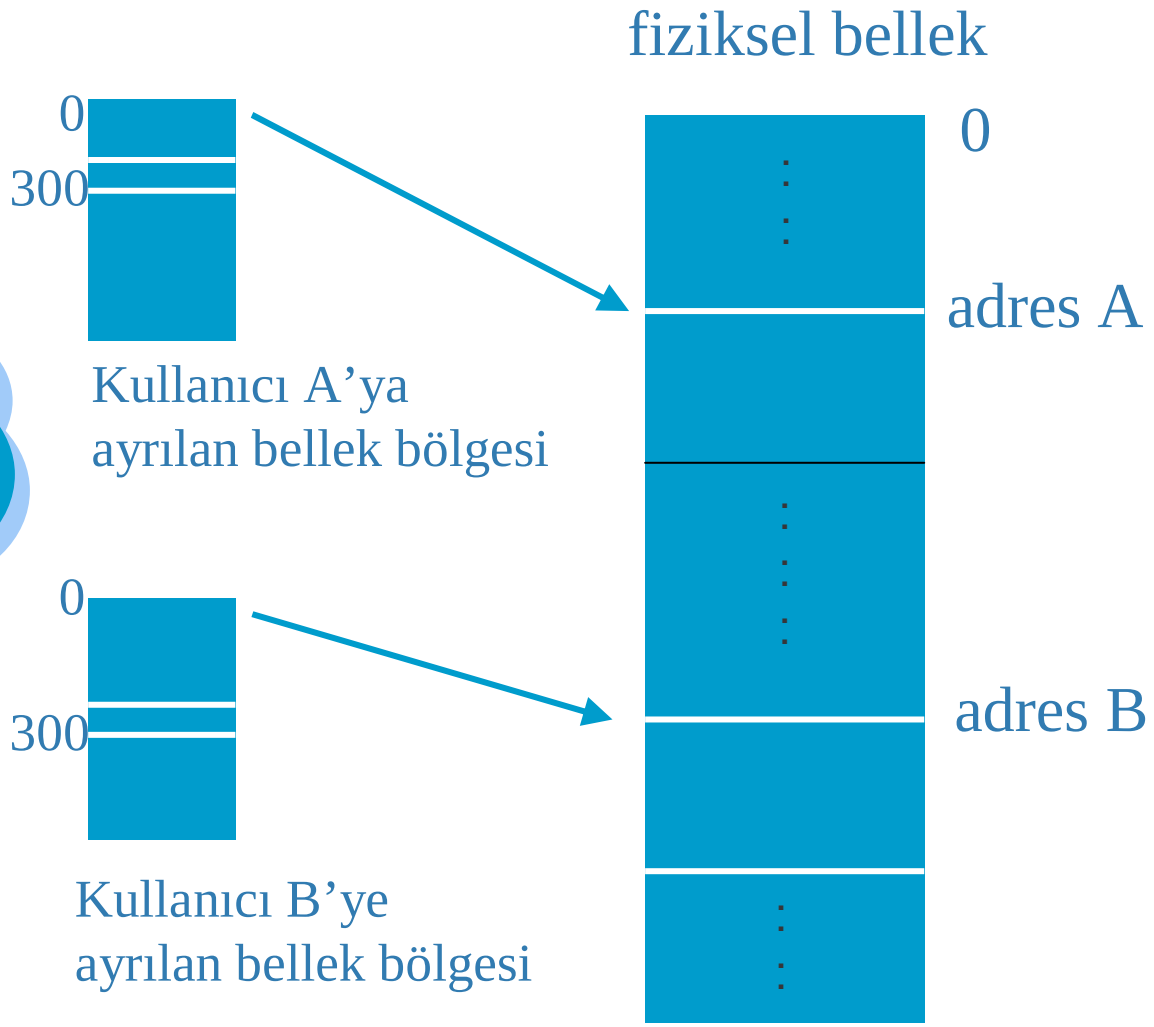
- G/Ç
  - donanıma yakın programlama gerekir
  - işletim sistemi kullanımı kolaylaştırır
    - aygıt sürücüler
  - örnek: diskten / disketten okuma

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

300'ler aynı adres değil!
her kullanıcı için kendi başlangıç (0) adresinden kayıklığı (offset) gösterir



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: IBM System z9™

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, Windows Mobile

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 tarafından 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

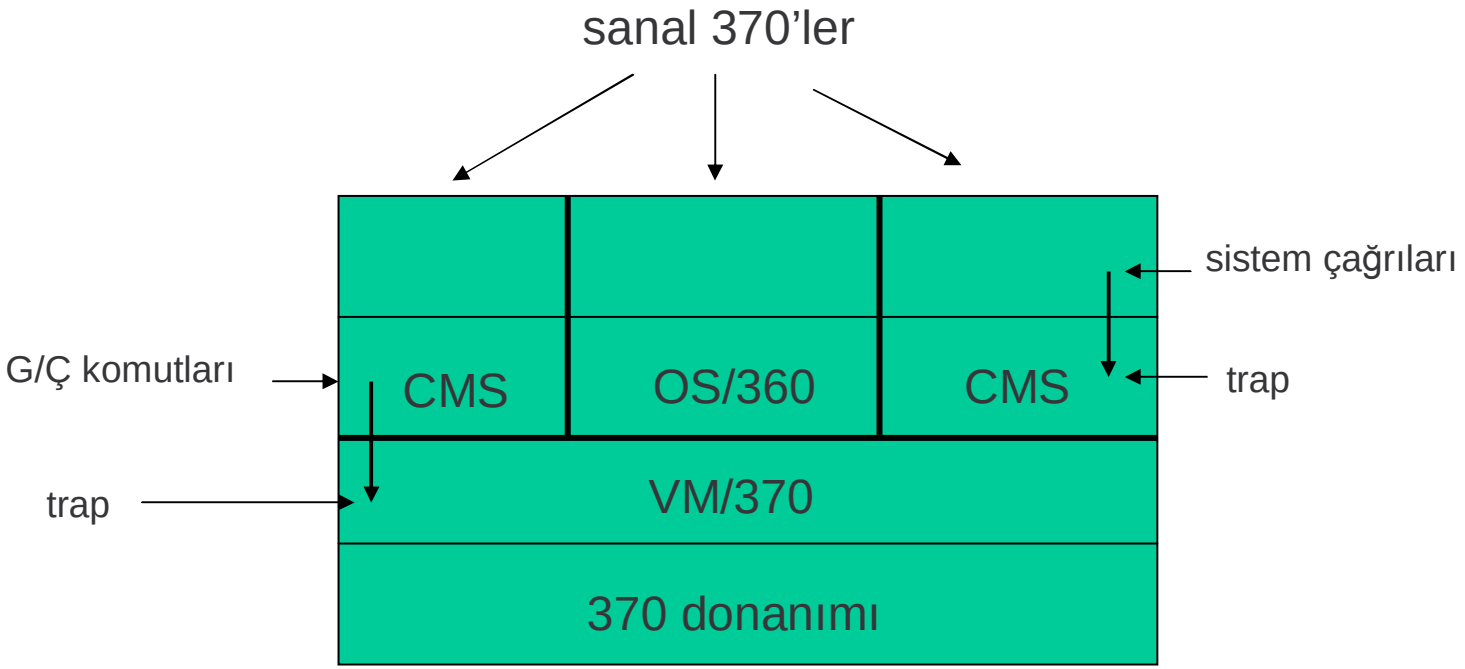
5	operatör
4	kullanıcı programları
3	G/Ç yönetimi
2	operatör-proses iletişimi
1	bellek ve tambur yönetimi
0	işlemci paylaştırma ve çoklu-programlama

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

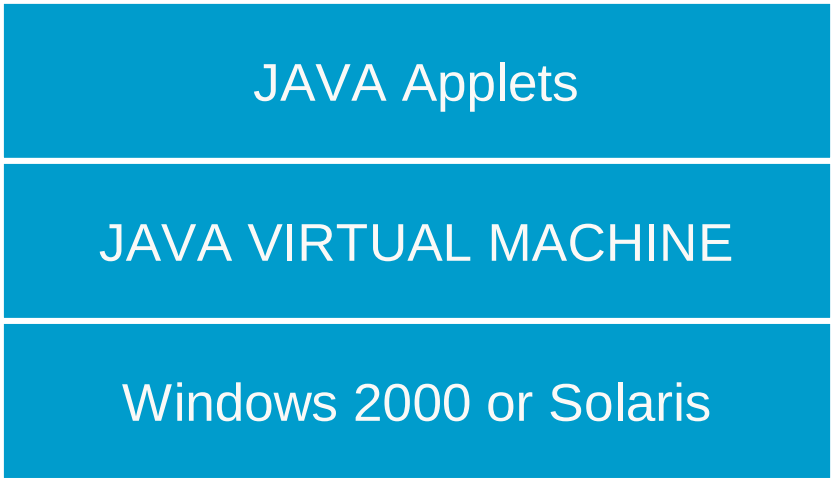
Her katman altındakinin yaptıklarıyla ilgilenmez. Örnek: 2. katmandaki işlemler için prosesin bellek veya tamburda olması önemli değil.

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



Sanal Makina



The Java Virtual Machine allows Java code to be portable between various hardware and OS platforms.

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

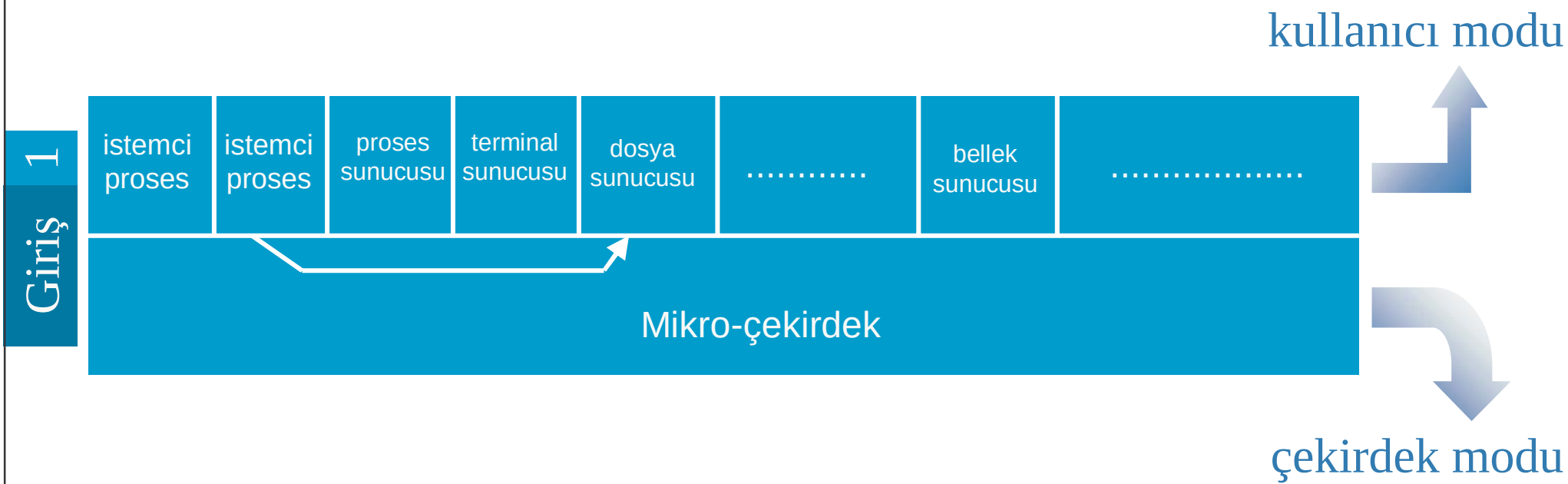
Sunucu-İstemci Modeli

- çekirdek minimal (mikro-çekirdek)
- işletim sisteminin çoğu kullanıcı modunda
- sunucu 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 sistemi 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



UYGULAMA

Bu ders kapsamında tüm uygulamalarımızı
RedHat Enterprise Linux (RHEL) 4.0 üzerinde yapacağız.

Brief History of Linux

- began with this post to the Usenet newsgroup `comp.os.minix`, in August, 1991

```
Hello everybody out there using minix-
I'm doing a (free) operating system (just a
hobby, won't be big and professional like
gnu) for 386(486) AT clones.
```

- written by a Finnish college student: Linus Torvalds
- Version 1.0 of the kernel was released on March 14, 1994.
- Version 2.x, the current stable kernel release, was officially released on ????, 2006.

GNU/Linux and Unix

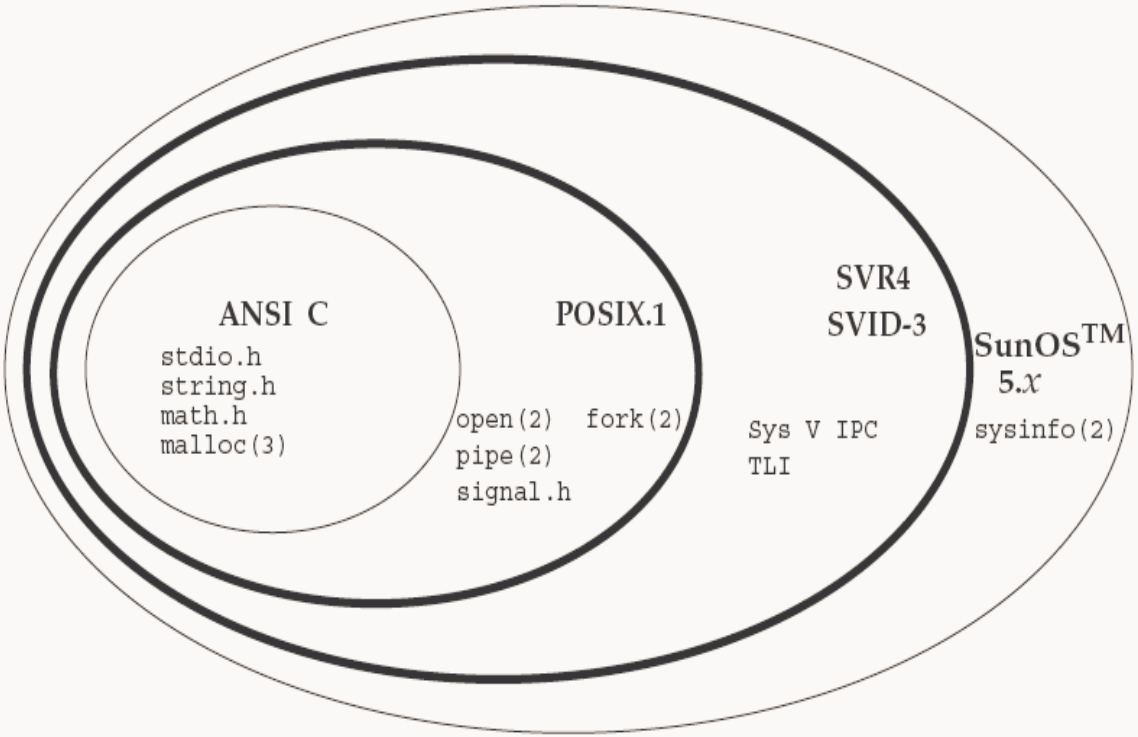
- GNU/Linux is not UNIX.
- UNIX is a registered trademark
- Linux is a UNIX clone
- All of the kernel code was written from scratch by Linus Torvalds and et al, $\Rightarrow$ Linux
- Many programs that run under Linux were also written from scratch, or were simply ports of software mostly from UNIX $\Rightarrow$ GNU Project

  [GNU is a recursive acronym: “GNU’s Not UNIX”]
- Powerful combination: the Linux kernel and software from the GNU Project; often called “Linux” for short

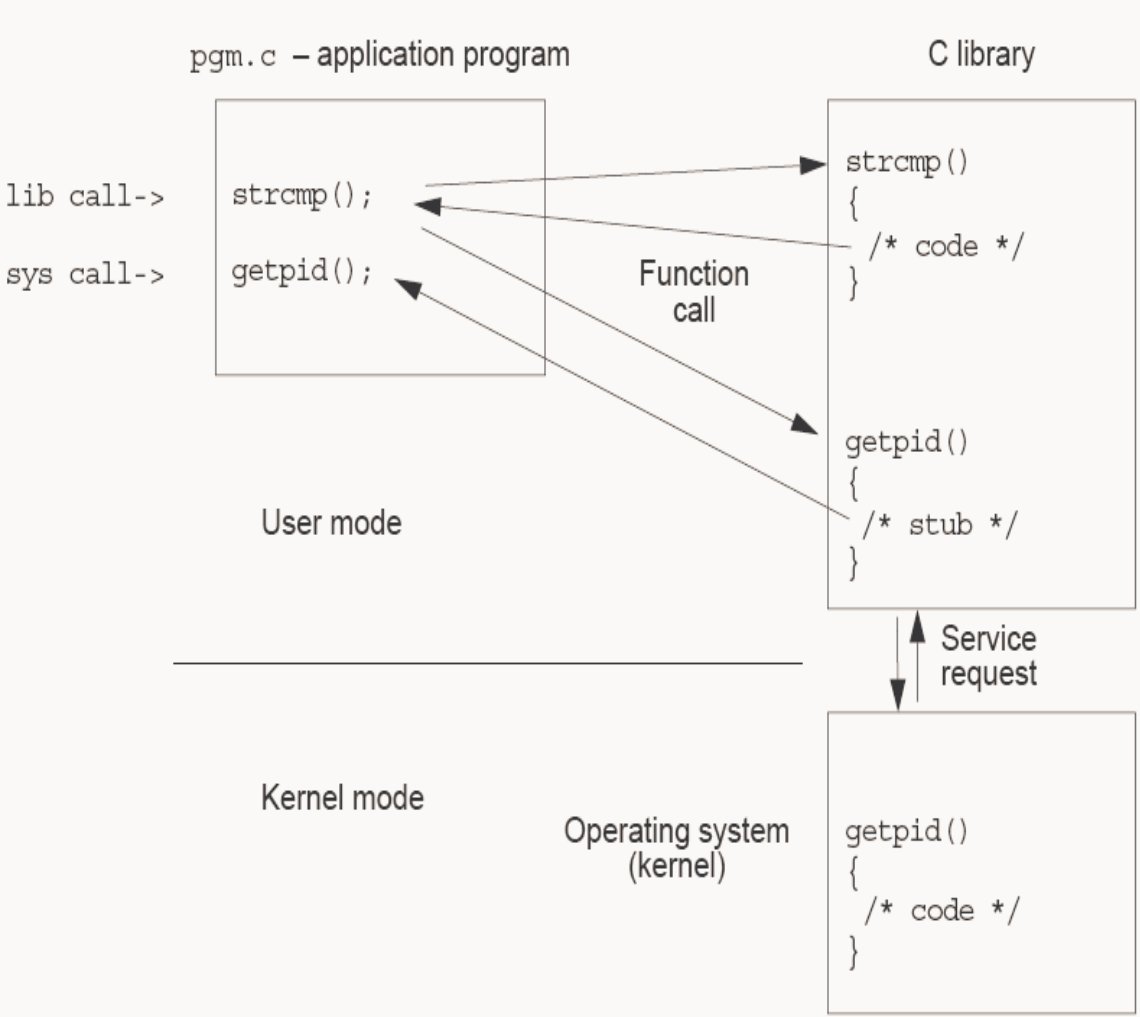
Programming Linux

- Linux is a POSIX operating system.
- POSIX is a family of standards developed by IEEE
- POSIX defines a portable operating system interface.
- What makes Linux such a high quality UNIX clone

Supported Programming Standards



System Calls and Library Calls



Making a System Call

mytime.c

```
1. #include <sys/types.h>
2. #include <time.h>
3. #include <stdio.h>
4. main() {
5.  /* Declare an object and pass its address */
6.  time_t t;
7.  time(&t);
8.  printf("Machine time in seconds = %d\n", t);
9. }
```

Making a System Call

badtime.c

```
1. #include <sys/types.h>
2. #include <time.h>
3. #include <stdio.h>
4. main() {
5.   /* Declare apointer variable only*/
6.   time_t *tptr;
7.   time(tptr);
8.   printf("Machine time in seconds = %d\n", *tptr);
9. }
```

Making a Library Call

mylibcall.c

```
1.  #include <sys/types.h>
2.  #include <time.h>
3.  #include <stdio.h>
4.  #include <string.h>
5.  #include <unistd.h>
6.  main() {
7.  time_t t;
8.  char then[30];
9.  char *now;
10. time(&t);
11. /* ctime() put the current time into static space */
12. now = ctime(&t);
13. /* Copy the data into a new space */
```

Making a Library Call

```
14. strcpy(then, now);
15. /* Let time pass for one minute */
16. sleep(60);
17. time(&t);
18. /* ctime() puts new time into old static space */
19. now = ctime(&t);
20. printf("%s%s", then, now);
21. }
```

Making a Library Call

```
1.  #include <sys/types.h>
2.  #include <time.h>
3.  #include <stdio.h>
4.  main() {
5.  time_t t;
6.  char *then;
7.  char *now;
8.  time(&t);
9.  /* ctime() put the current time into static space */
10. then = ctime(&t);
11. /* Let time pass for one minute */
12. sleep(60);
```

Making a Library Call

```
13. time(&t);

14. /* ctime() puts new time into same space */
15. now = ctime(&t);

16. /* then and now point to the same space */
17. printf("%s%s", then, now);
18. }
```

Error Handling


```
1. #include <sys/types.h>
2. #include <unistd.h>
3. #include <stdio.h>
4. void perror();

5. main() {

6. if (setuid(23) == -1) {
7.     perror(“Setuid failure”);
8.  }
9. }
```

Error Handling

```
1. #include <errno.h>
2. #include <string.h>
3. #include <stdio.h>
4. #include <fcntl.h>
5. #define NAMESIZZE 20
6.  main() {
7.      char filename[NAMESIZE];
8.      int fd;
9.      gets(filename)
10.    if ((fd = open(filename, O_RDWR)) == -1) {
11.        fprintf (stderr, “Error opening %s: %s\n”,
12.                    filename, strerror(errno));
13.        exit (errno);
14.     }
15.  }
```

System Calls Compared With Library Calls

System Call	Library Call
Described in Section 2 of the man pages.	Described in sections 3 of the man pages.
Never allocates space for parameters.	Can allocate space for parameters (see man pages). If allocates space, it can be static or dynamic.
Executes in system mode (kernel mode).	Executes in user mode.
When a failure occurs:	When a failure occurs:
Returns -1.	Often returns NULL (see man pages).
Sets `errno` (so you can use `perror(3C)`).	Can set `errno` (see man pages).

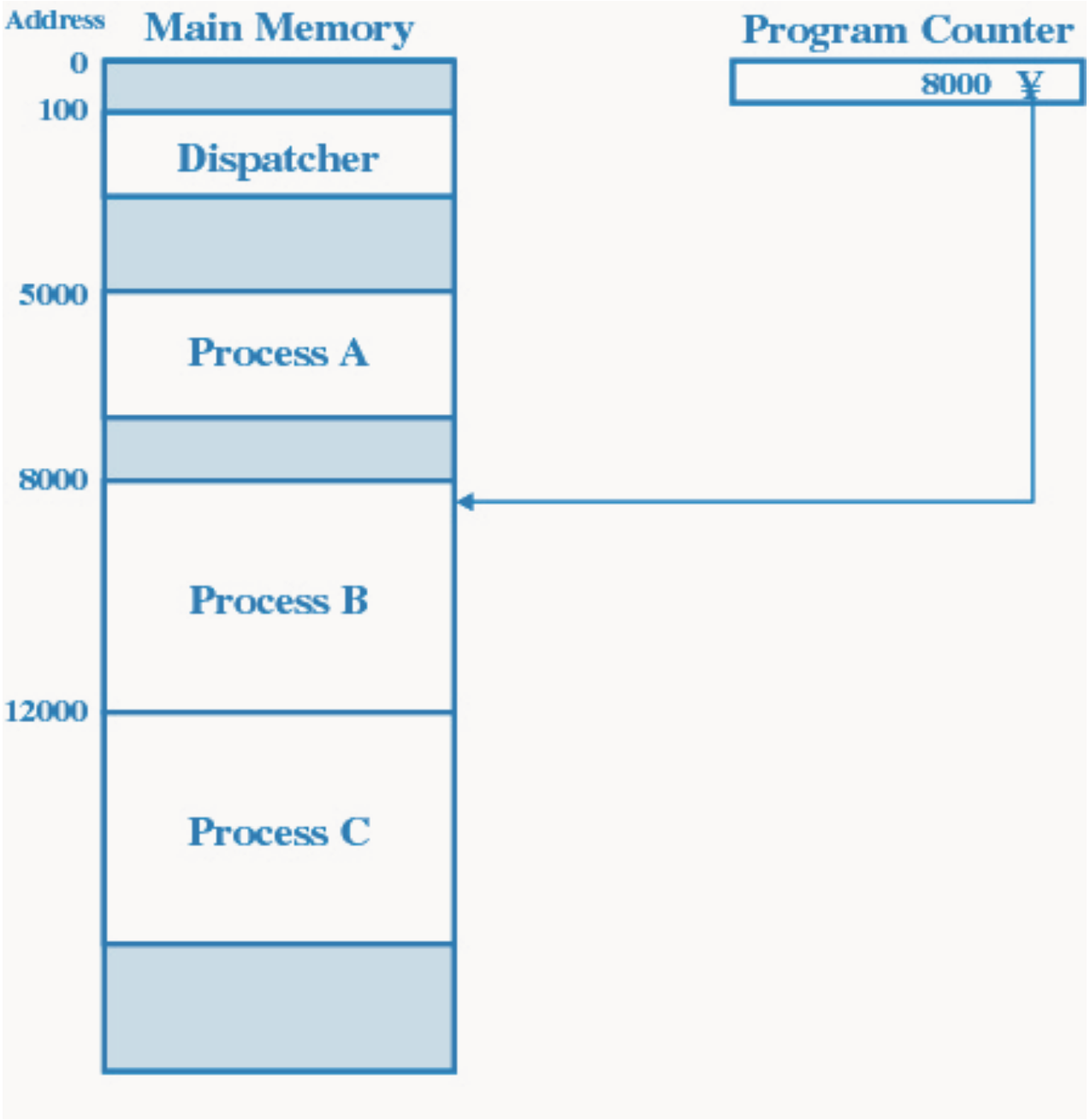
2 PROSESLER

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


Address
Main Memory
Program Counter
0
100
Dispatcher
8000
5000
Process A
8000
Process B
12000
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

(a) Trace of Process A **(b) Trace of Process B** **(c) Trace of Process C**

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

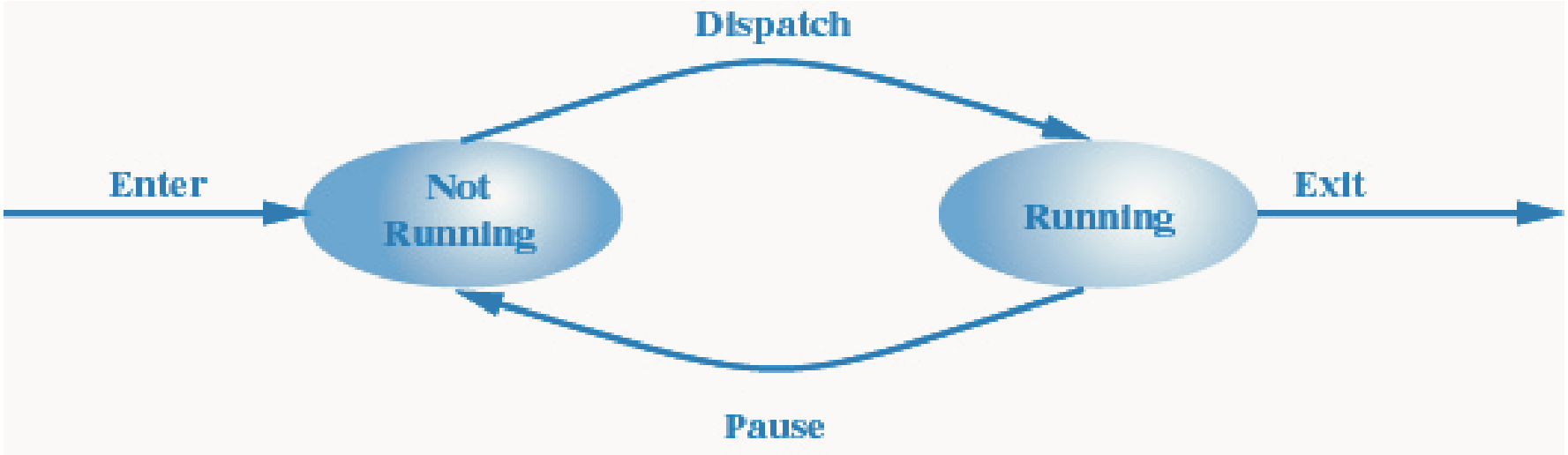
100 = Starting address of dispatcher program

Proses

- Proseslerin işlemciye sahip olma sıraları kestirilemez $\Rightarrow$ program kodunda zamanlamaya dayalı işlem olmamalı

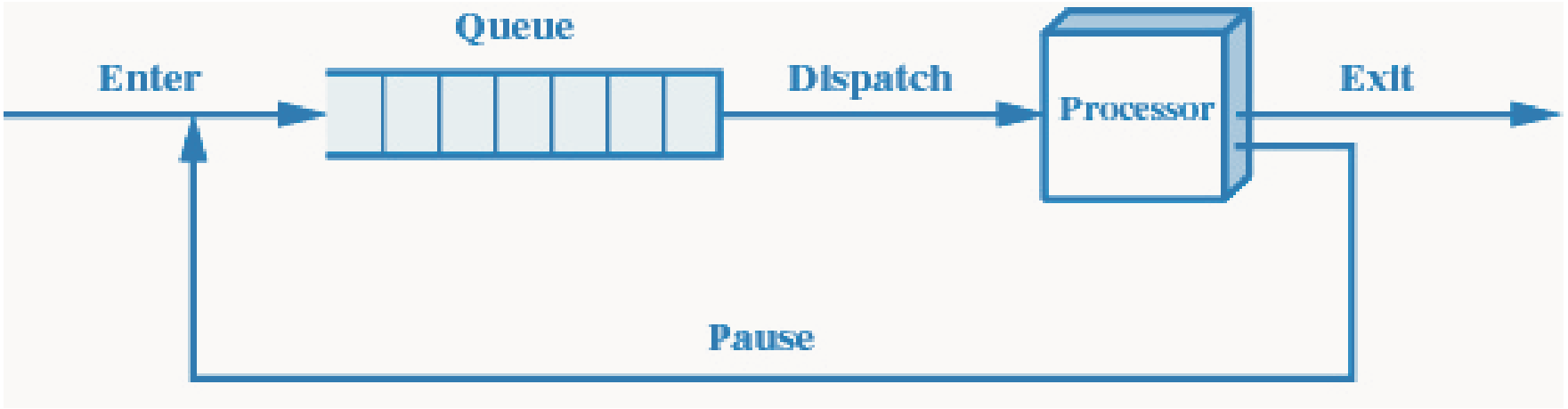
İki Durumlu Proses Modeli

- Proses iki durumdan birinde olabilir:
  - Koşuyor
  - Koşmuyor



Proses Kuyruğu

O anda çalışmayan proses sırasını bir kuyrukta bekler:

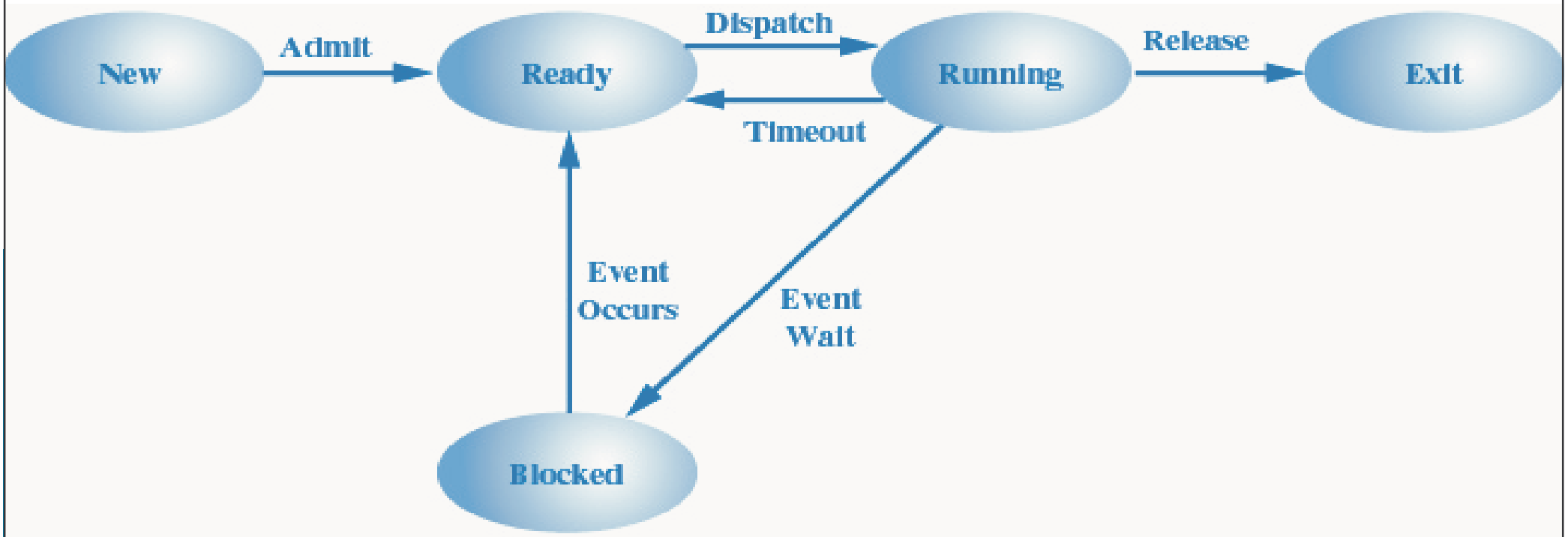


Proses

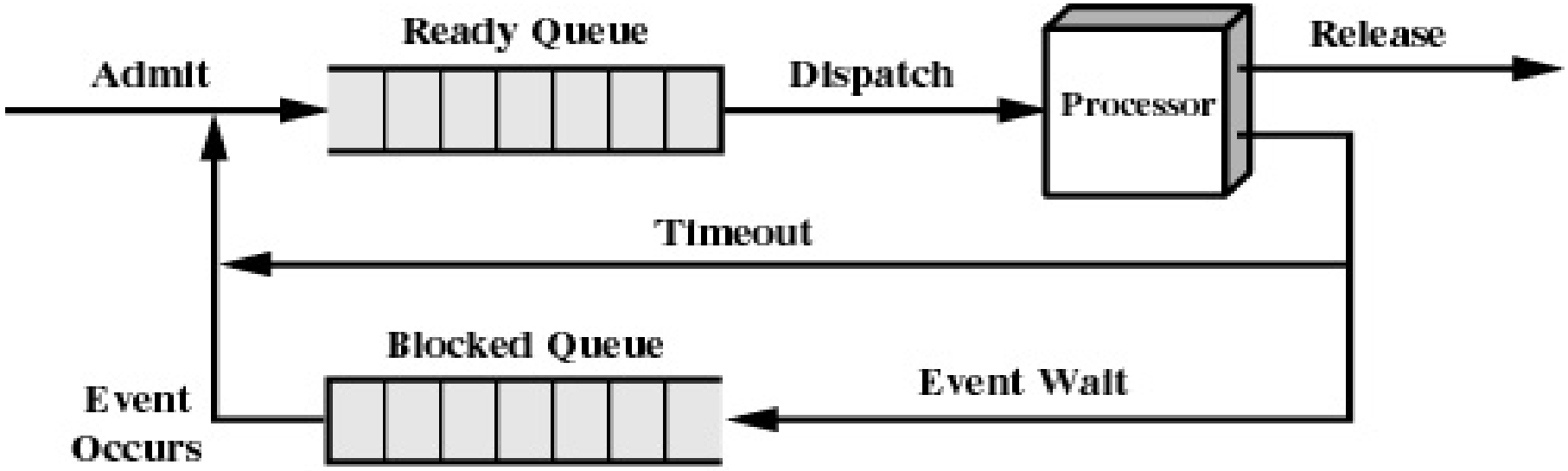
- Koşmuyor
  - çalışmaya hazır
- Bloke
  - G/Ç bekliyor
- Kuyrukta en uzun süre beklemiş prosesin çalıştırılmak üzere seçilmesi doğru olmaz
  - Bloke olabilir

Beş-Durumlu Model

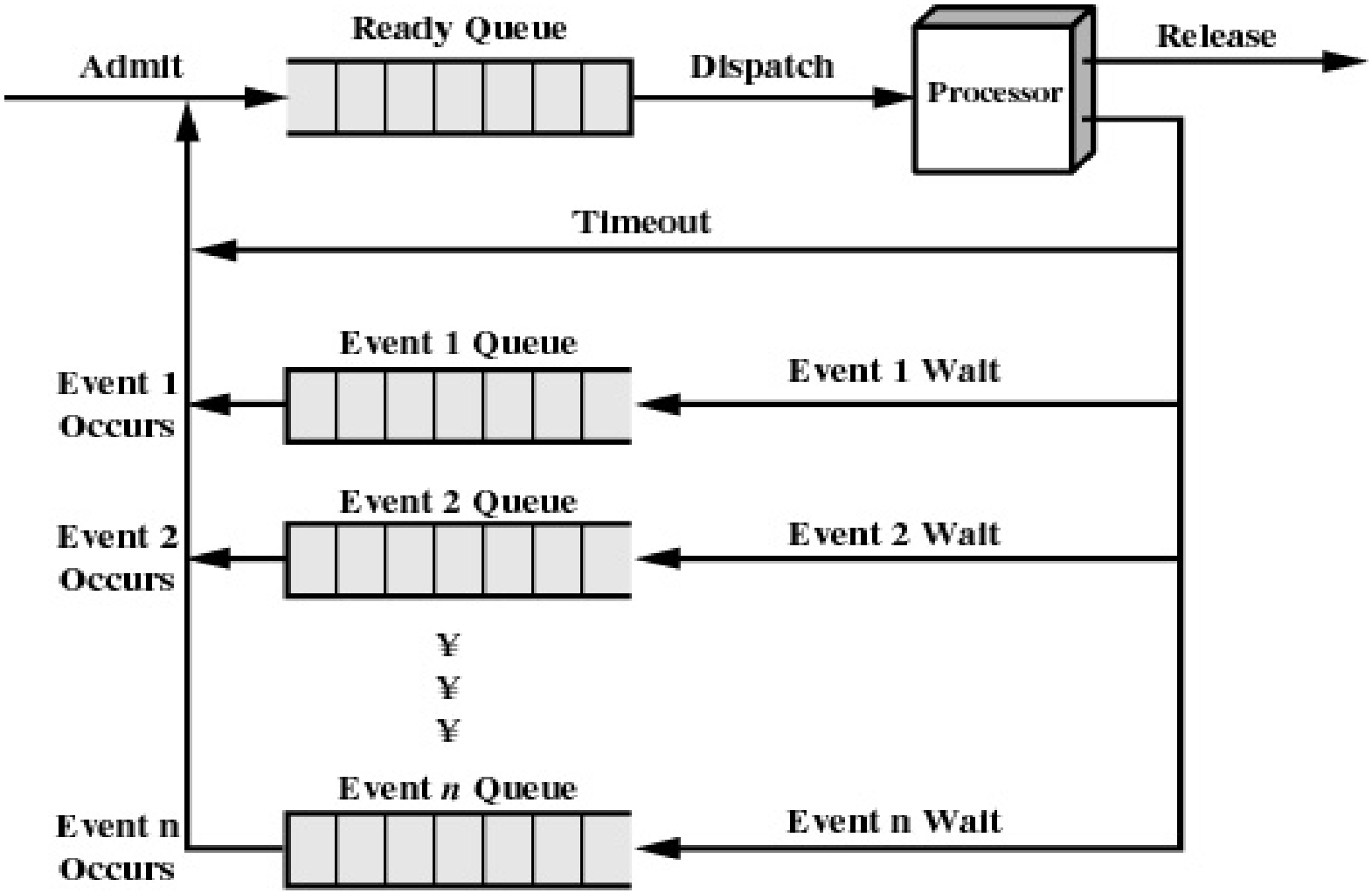
- Koşuyor
- Hazır
- Bloke
- Yeni
- Sonlanıyor


New
Admit
Ready
Dispatch
Running
Release
Exit
Timeout
Event
Occurs
Event
Wait
Blocked

İki Kuyruk


Admit
Ready Queue
Dispatch
Processor
Release
Timeout
Blocked Queue
Event Wait
Event
Occurs

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

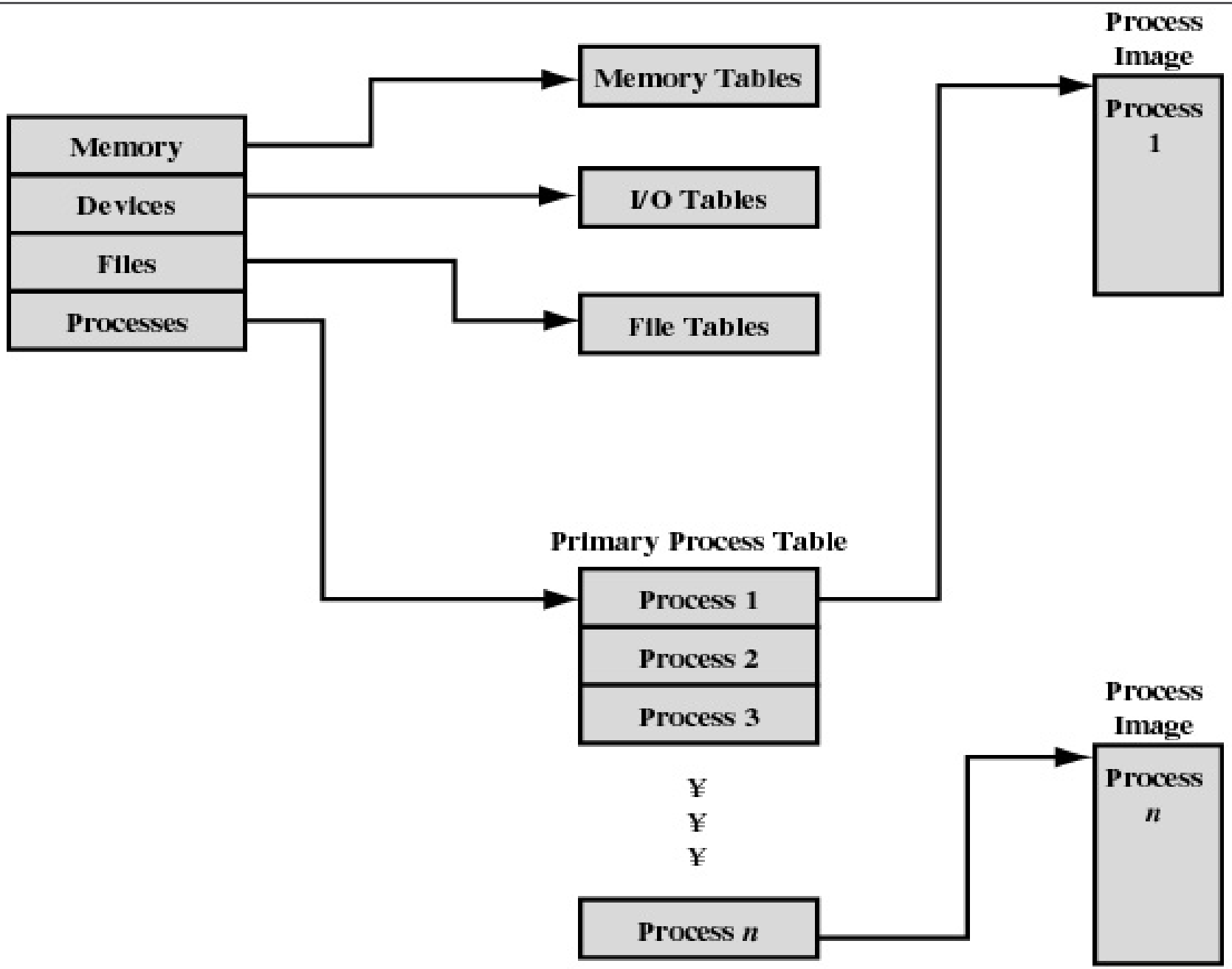
- Her proses ve kaynak ile ilgili durum bilgilerinin tutulması gerekir
  - İşletim sistemi tarafından yönetilen her varlık için tablolar tutulur
    - G/Ç Tabloları
    - Bellek Tabloları
    - Dosya Tabloları
    - Proses Tabloları

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


Memory
Devices
Files
Processes
Memory Tables
I/O Tables
File Tables
Primary Process Table
Process 1
Process 2
Process 3
Process n
Process Image
Process 1
Process Image
Process n

Proses Kontrol Bloğu

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

Proses Kontrol Bloğu

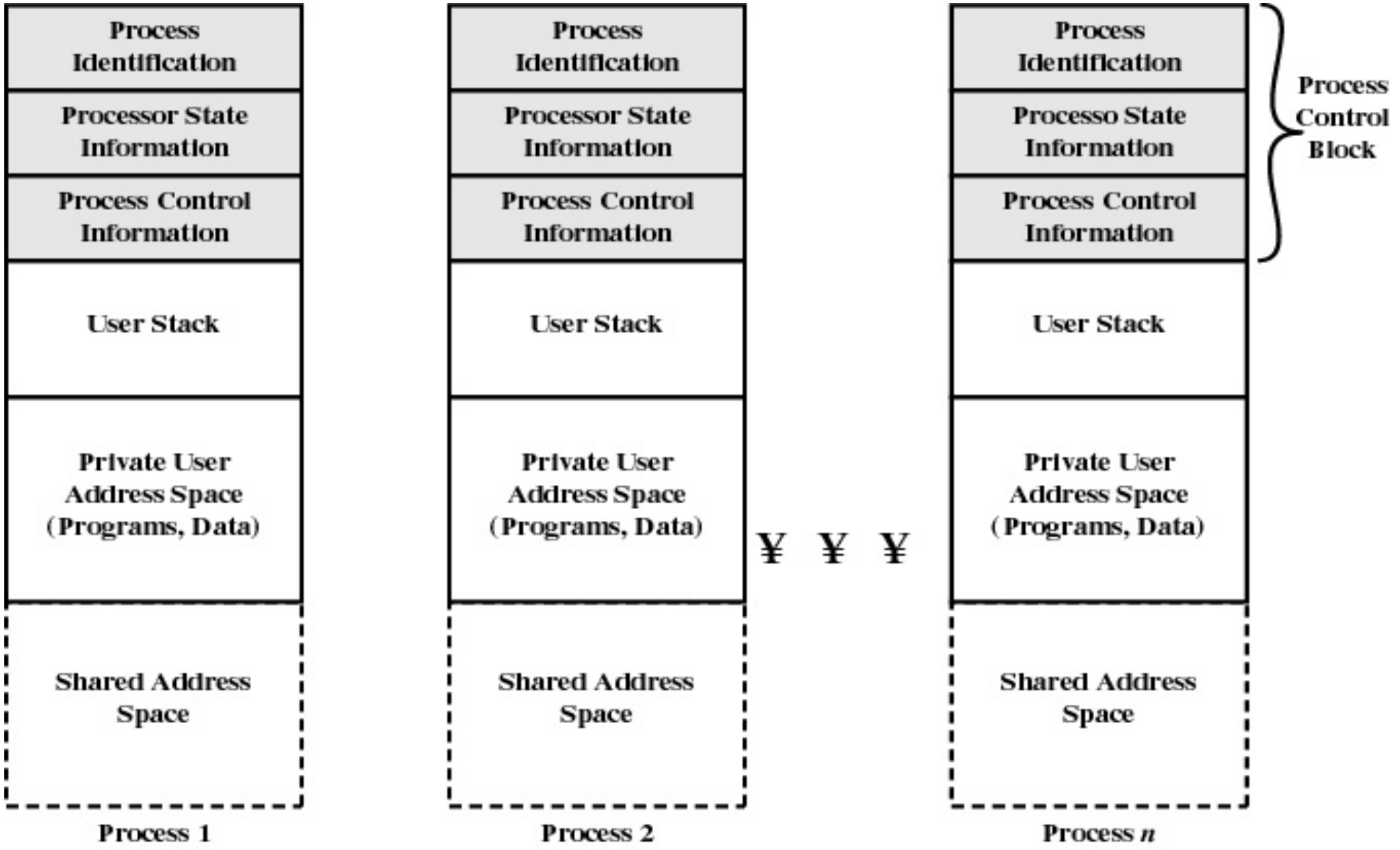
- İşlemci Durum Bilgisi
  - Kullanıcıya açık saklayıcılar
    - İşlemcinin makina dili kullanılarak erişilebilen saklayıcıları.
  - Kontrol ve Durum saklayıcıları
    - Program sayacı
    - Durum saklayıcısı
    - Yığın işaretçileri
    - Program durum sözcüğü (çalışma modu biti var)

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


Process Identification
Processor State Information
Process Control Information
User Stack
Private User Address Space (Programs, Data)
Shared Address Space
Process 1
Process Identification
Processor State Information
Process Control Information
User Stack
Private User Address Space (Programs, Data)
Shared Address Space
Process 2
¥ ¥ ¥
Process Identification
Processo State Information
Process Control Information
User Stack
Private User Address Space (Programs, Data)
Shared Address Space
Process n
Process Control Block

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

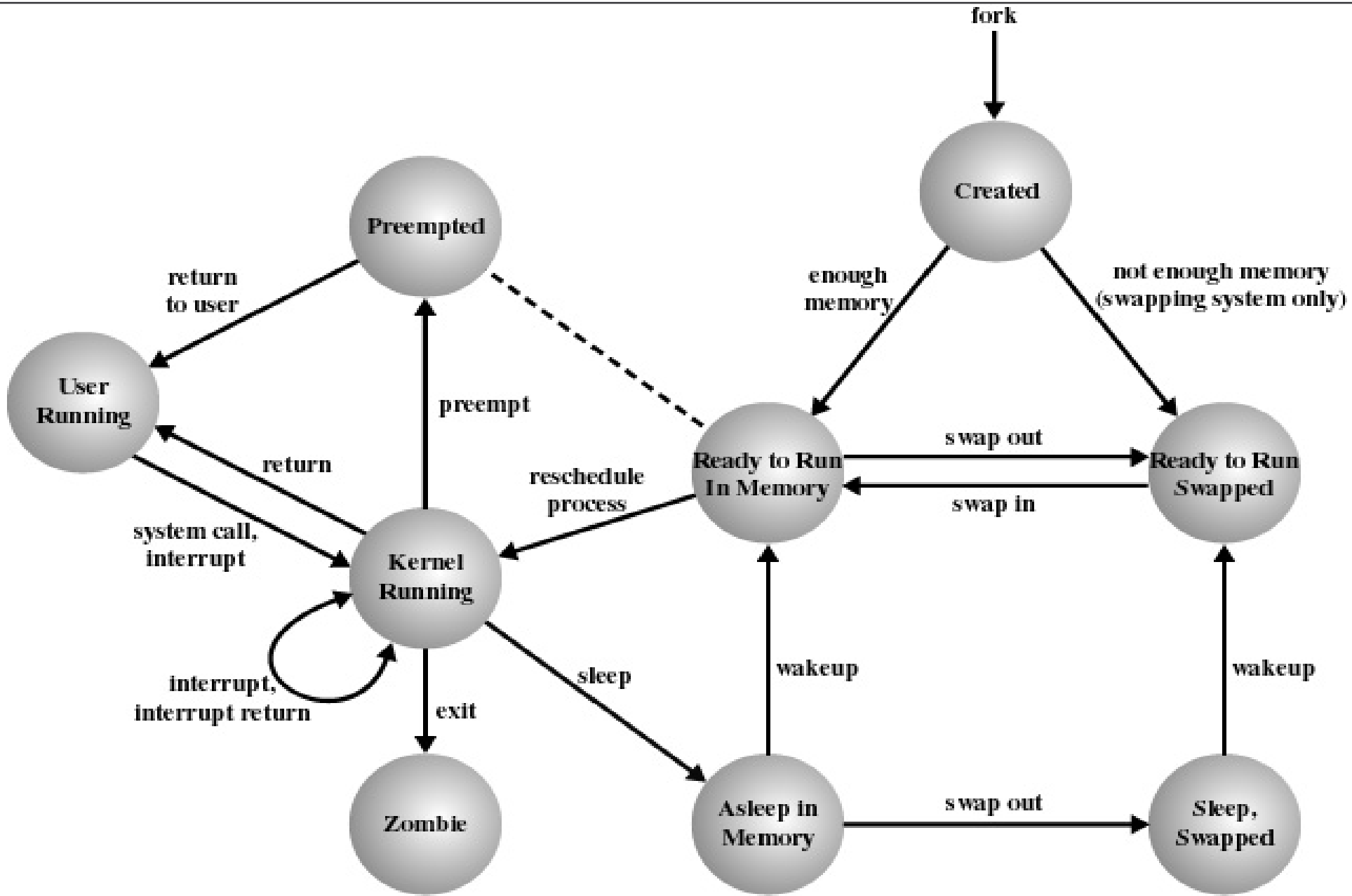
- Seçilen prosesin proses kontrol bloğunun güncellenmesi
- Bellek yönetimi ile ilgili bilgilerin güncellenmesi
- Seçilen prosesin bağlamının yüklenmesi

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)


fork
Created
Preempted
return to user
enough memory
not enough memory (swapping system only)
User Running
preempt
return
reschedule process
Ready to Run In Memory
swap out
Ready to Run Swapped
swap in
system call, interrupt
Kernel Running
interrupt, interrupt return
exit
sleep
wakeup
wakeup
Zombie
Asleep in Memory
swap out
Sleep, Swapped

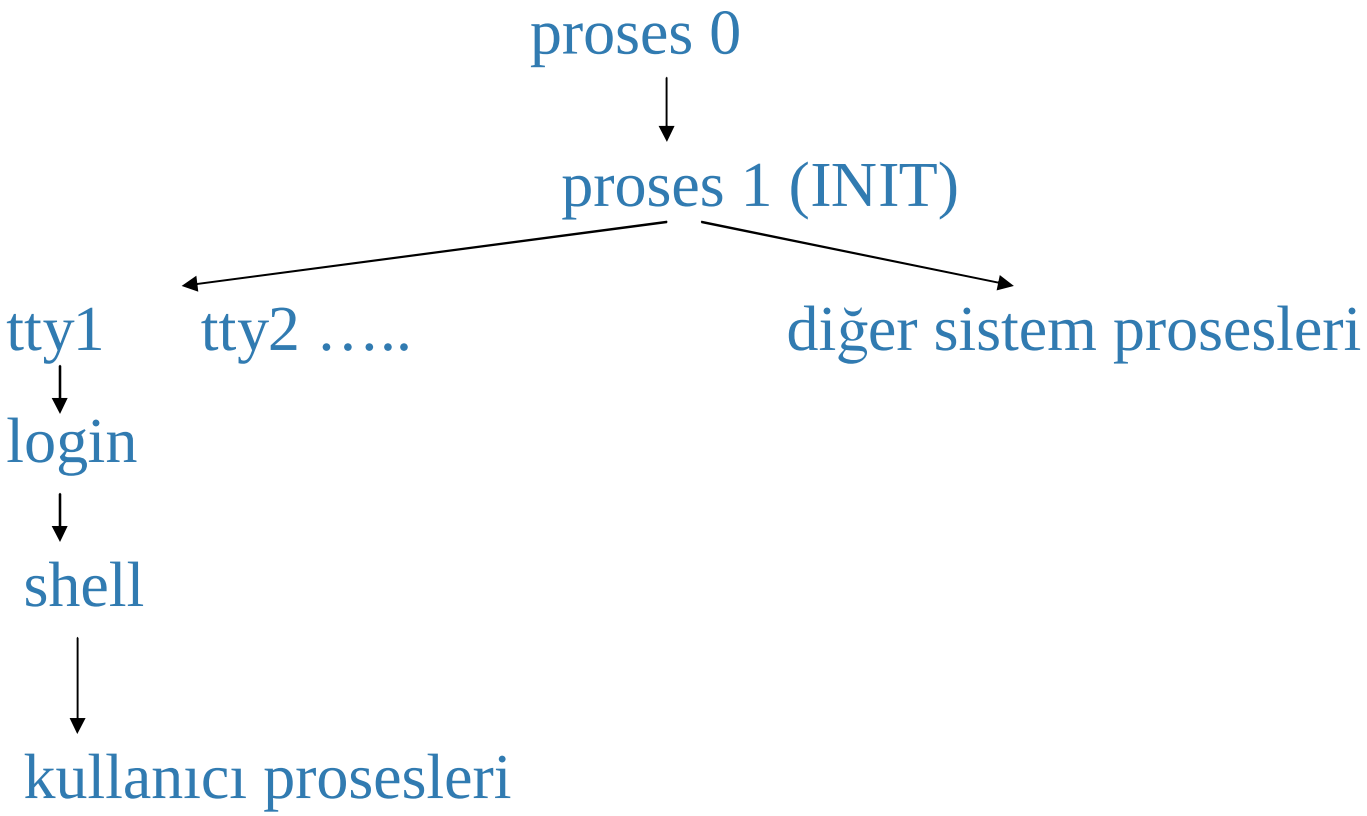
UNIX'de Proses Yaratma

- 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);
}
```

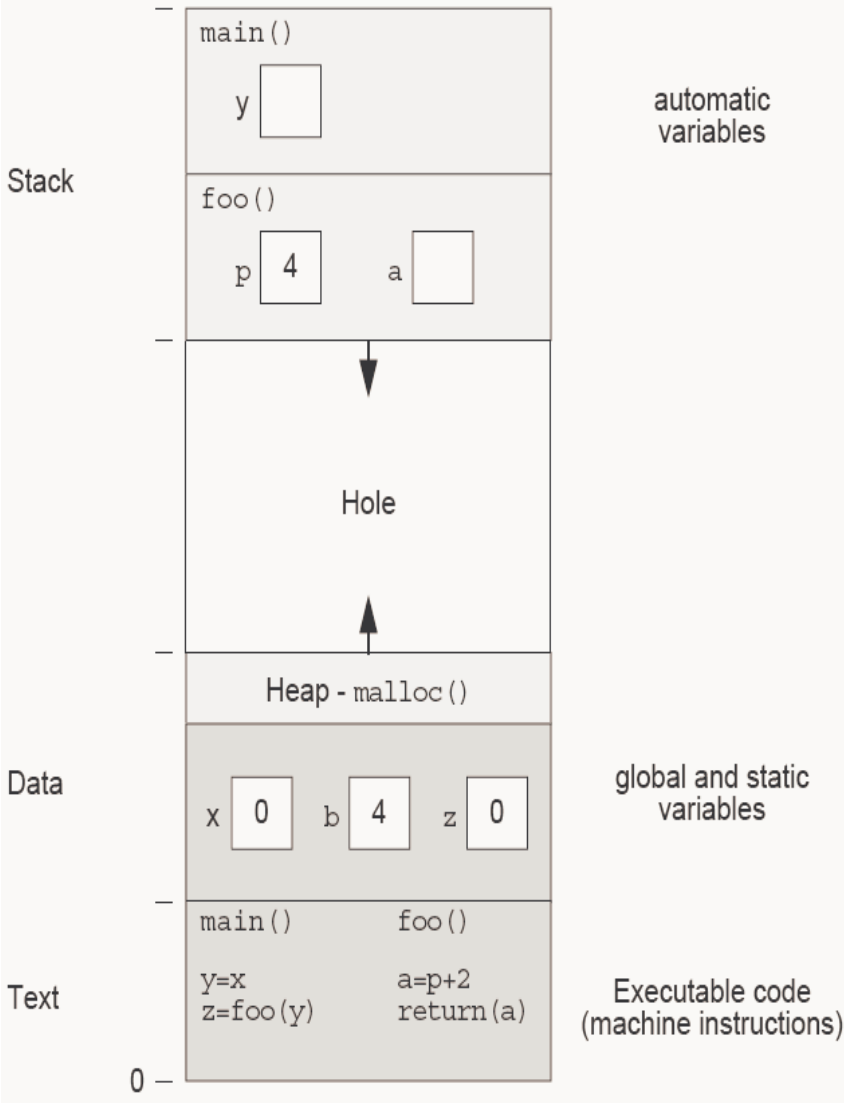
UYGULAMA

Defining a Program

pgm.c

```
1 int x;
2 static int b = 4;
3
4 main() {
5 int y;
6 static int z;
7
8 y = b;
9 z = foo(y);
10 }
11
12 foo( int p ) {
13 int a;
14 a = p + 2;
15 return(a);
16 }
```

Processes as Compared to Procedure Calls



Creating a Process

mysystem.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 main() {
5
6 int rv;
7
8 /* Sometimes useful for troubleshooting, */
9 /* E.g., do "ps" command inside program and save
  output */
10 rv = system("ps -le | grep mysystem > /tmp/junk");
11
12 if ( rv != 0 ) {
13 fprintf(stderr, "Something went wrong!\n");
14 }
15 }
```

myfork.c

```
1 #include <sys/types.h>
2 #include <unistd.h>
3 #include <stdio.h>
4
5 main() {
6
7 pid_t pid;
8
9 pid = fork();
10
11 switch(pid) {
12
13 /* fork failed! */
14 case -1:
15 perror("fork");
16 exit(1);
17
18 /* in new child process */
19 case 0:
20 printf("In Child, my pid is: %d\n",getpid());
```

```
21 do_child_stuff();
22 exit(0);
23
24 /* in parent, pid is PID of child */
25 default:
26 break;
27 }
28
29 /* Rest of parent program code */
30 printf("In parent, my child is %d\n", pid);
31 }
32
33 int do_child_stuff() {
34 printf("\t Child activity here \n");
35 }
```

Running a New Program

	Give Absolute Path	Use PATH Variable	New Environment
argv as list	`execl()`	`execlp()`	`execle()`
argv as vector	`execv()`	`execvp()`	`execve()`

Running a New Program

```
1 #include <stdio.h>
2 #include <sys/types.h>
3 #include <unistd.h>
4 #include <sys/wait.h>
5
6 main() {
7
8 pid_t pid;
9
10 pid = fork();
11
12 switch(pid) {
13
14 /* fork failed! */
15 case -1:
16 perror("fork");
17 exit(1);
18 /* in new child process */
19 case 0:
```

```
20 execlp("ls", "ls", "-F", (char *)0);
21 /* why no test? */
22 perror("execlp");
23 exit(1);
24
25 /* in parent, pid is PID of child */
26 default:
27 break;
28 }
29
30 /* Rest of parent program code */
31 wait(NULL);
32 printf("In parent, my child is %d\n", pid);
33 }
```

Terminating a Process

- exit()— Flushes buffers and calls _exit() to terminate process
- _exit()— Closes files and terminates process
- atexit()— Stores function for future execution before terminating a process
- abort()— Closes files, terminates the process, and produces a core dump for debugging

Terminating a Process


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

void cleanup() {
char *message = "cleanup invoked\n";
      write(STDOUT_FILENO, message,
  strlen(message));
}

main() {
 /* Register cleanup() as atexit function */
 atexit(cleanup);
 /* Other things in program done here */
 exit(0);
 }
```

Cleaning Up Terminated Processes

- wait()– Blocks the process until one of its children is ready to have its status reaped.
- waitpid()– Allows you to specify which process to wait for and whether to block.

Cleaning Up Terminated Processes


```
1 #include <sys/types.h>
2 #include <sys/wait.h>
3 #include <stdio.h>
4 #include <unistd.h>
5 #include <stdlib.h>
6
7 main() {
8
9 pid_t pid;
10 int status;
11
12 /* fork() a child */
13 switch(pid = fork()) {
14
15 case -1:
16 perror("fork");
17 exit(1);
18
19 /* in child */
```

```
20 case 0:
21 execlp("ls","ls","-F",(char *)NULL);
22 perror("execlp");
23 exit(1);
24
25 /* parent */
26 default:
27 break;
28 }
29
30 if (waitpid(pid, &status, 0) == -1) {
31 perror("waitpid");
32 exit(1);
33 }
34
35 /* See wstat(5) for macros used with status
36 from wait(2) */
37 if (WIFSIGNALED(status)) {
38 printf("ls terminated by signal %d.\n",
39 WTERMSIG(status));
40 } else if (WIFEXITED(status)) {
41 printf("ls exited with status %d.\n",
```

```
42 WEXITSTATUS(status));
43 } else if (WIFSTOPPED(status)) {
44 printf("ls stopped by signal %d.\n",
45 WSTOPSIG(status));
46 }
47 return 0;
48 }
```

Retrieving Host Information

- uname()— Retrieves host name and other related information
- sysinfo()— Reports and sets information about the operating system

myuname.c

```
1 #include <sys/utsname.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 main() {
6
7 struct utsname uts;
8
9 if( uname(&uts) == -1 ) {
10 perror("myuname.c:main:uname");
11 exit(1);
12 }
13
14 printf("operating system: %s\n", uts.sysname);
15 printf("hostname: %s\n", uts.nodename);
16 printf("release: %s\n", uts.release);
17 printf("version: %s\n", uts.version);
18 printf("machine: %s\n", uts.machine);
19 }
```

mysysinfo.c

```
1 #include <sys/systeminfo.h>
2 #include <stdio.h>
3
4 #define BUFSIZE 1024
5
6 main() {
7 char buf[BUFSIZE];
8 int num;
9
10 num = sysinfo( SI_HW_SERIAL, buf, BUFSIZE);
11 if (num == -1) {
12 perror("sysinfo");
13 exit(1);
14 }
15
16 printf("hostid: %s\n", buf);
17 printf("hostid: %x\n", atoi(buf));
18 }
```

Retrieving System Variables


```
1 #include <sys/unistd.h>
2 #include <stdio.h>
3
4 main() {
5
6 printf("Number of processors: %d\n",
7 sysconf(_SC_NPROCESSORS_CONF));
8 printf("Memory page size: %d\n",
9 sysconf(_SC_PAGESIZE));
10 printf("Clock ticks/second: %d\n",
11 sysconf(_SC_CLK_TCK));
12 printf("Number of files that can be
13 opened: %d\n", sysconf(_SC_OPEN_MAX));
14 }
```

Determining File and Directory Limits


```
1 #include <unistd.h>
2 #include <stdio.h>
3
4 main() {
5
6 printf("Maximum filename length: %d\n",
7 pathconf(".", _PC_NAME_MAX));
8 printf("Maximum path length: %d\n",
9 pathconf("/", _PC_PATH_MAX));
10 printf("Pipe buffer size: %d\n",
11 pathconf("/var/spool/cron/FIFO",
12 _PC_PIPE_BUF));
13 }
```

Retrieving User Information

- getuid()— Retrieves user identification numbers
- getpwuid()— Retrieves the user identification number from the password database
- getpwnam()— Retrieves the user name from the password database

Retrieving User Information


```
1 #include <sys/types.h>
2 #include <unistd.h>
3 #include <pwd.h>
4 #include <stdio.h>
5
6 main() {
7
8 struct passwd *pw;
9
10 pw = getpwuid( getuid() );
11 printf("Logged in as %s\n", pw->pw_name);
12 }
```

Retrieving Machine Time

- time() – Returns number of seconds since 0:00:00, January 1, 1970
- gettimeofday() – Returns time in seconds and microseconds
- ctime() – Returns time in a human readable format
- gmtime()– Breaks time from time() into fields from seconds to years, Greenwich mean time
- localtime()– Same as gmtime() except local time
- strftime() – Returns time in customized string format

Using Environment Variables

- getenv()— Retrieves the value of an environment variable
- putenv()— Adds variables to the environment
- unsetenv()— Removes an environment variable

Using Environment Variables

mygetenv.c

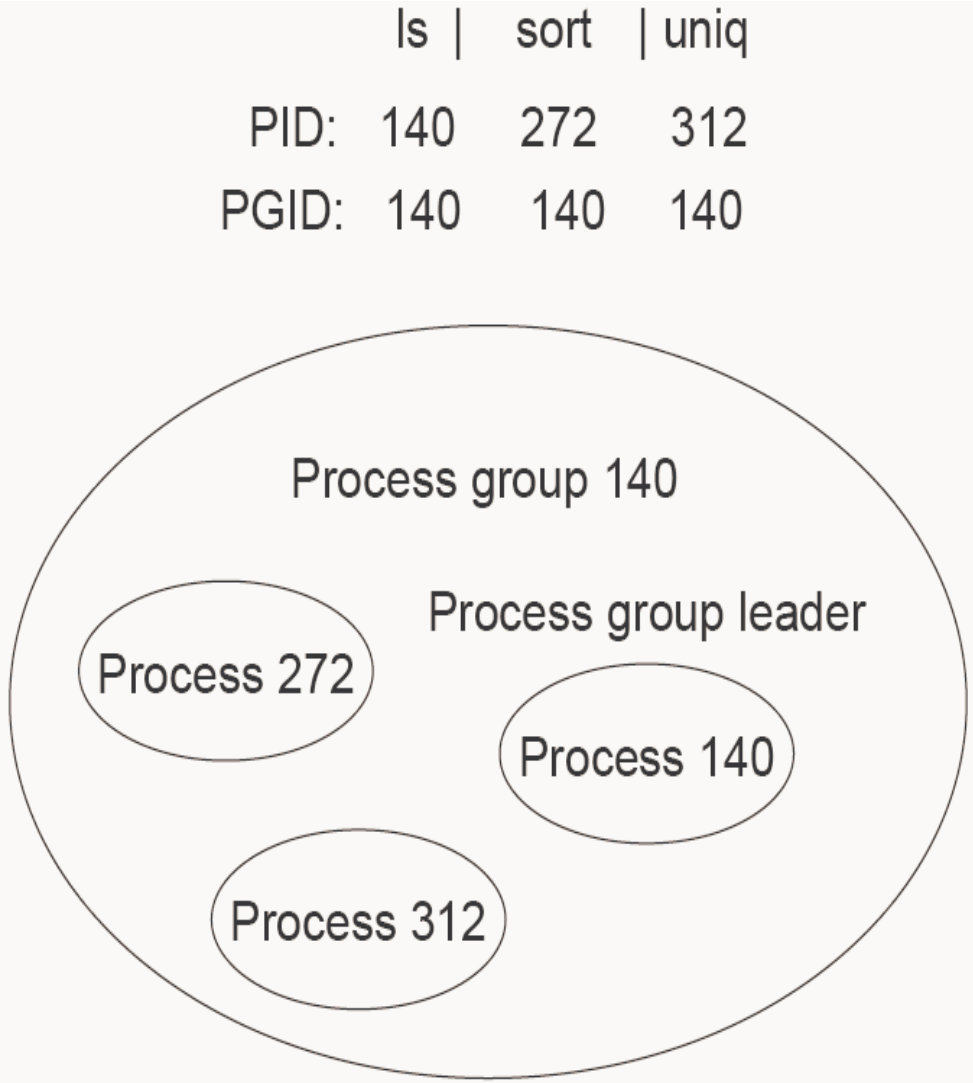
```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 main() {
5
6 char *value;
7
8 value = getenv("HOME");
9 if( value == NULL ) {
10 printf("HOME is not defined\n");
11 } else if( *value == '\0') {
12 printf("HOME defined but has no value\n");
13 } else {
14 printf("HOME = %s\n", value);
15 }
16 }
```

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 main() {
5
6 char *value;
7
8 putenv("HOME=/tmp");
9
10 value = getenv("HOME");
11 if( value == NULL ) {
12 printf("HOME is not defined\n");
13 } else if( *value == '\0') {
14 printf("HOME defined but has no value\n");
15 } else {
16 printf("HOME = %s\n", value);
17 }
18 }
```

Using Process IDs and Process Groups IDs

- Function Calls
  - getpid()— Retrieves process ID number
  - getppid()— Retrieves parent PID
  - getpgrp()— Retrieves process group ID number
  - getpgid()— Retrieves process group GID
- Identification Numbers
  - Process ID
  - Process group ID
  - User group ID and group ID
  - Effective user ID and effective group ID

Using Process IDs and Process Groups IDs



Using Real and Effective Group IDs

- getuid()— Retrieves real user ID
- getgid()— Retrieves real group user ID
- geteuid()— Retrieves effective user ID
- geteguid()— Retrieves effective group user ID

Resource Limits

- limit()— Displays and sets resources limits
- getrlimit()— Displays resource limits
- setrlimit()— Sets resources limits

Resource Limits

Resource Macro	Meaning	Signal	errno
RLIMIT_CORE	The maximum size of a core file in bytes that a process can create.		
RLIMIT_CPU	The maximum amount of CPU time in seconds used by a process. Soft only.	SIGXCPU	
RLIMIT_DATA	The maximum size of a process's heap in bytes.		ENOMEM
RLIMIT_FSIZE	The maximum size of a file in bytes that a process may create.	SIGXFSZ	EFBIG
RLIMIT_NOFILE	The maximum number of file descriptors that a process may create.		EMFILE
RLIMIT_STACK	The maximum size of a process's stack in bytes.	SIGSEGV	
RLIMIT_VMEM	The maximum size of a process's mapped address space in bytes.		ENOMEM

Resource Limits


```
1 #include <sys/resource.h>
2 #include <unistd.h>
3 #include <stdio.h>
4 main() {
5
6 struct rlimit myrlim;
7
8 getrlimit(RLIMIT_NOFILE, &myrlim);
9 printf("I can only open %d files\n", myrlim.rlim_cur);
10
11 myrlim.rlim_cur = 512;
12
13 if (setrlimit(RLIMIT_NOFILE, &myrlim) == -1) {
14 perror("setrlimit");
15 }
16
17 getrlimit(RLIMIT_NOFILE, &myrlim);
```

Resource Limits

```
18 printf("I can now open %d files\n",
myrlim.rlim_cur);
19 printf("sysconf() says %d files.\n",
20 sysconf(_SC_OPEN_MAX));
21 }
```

3 İPLİKLER

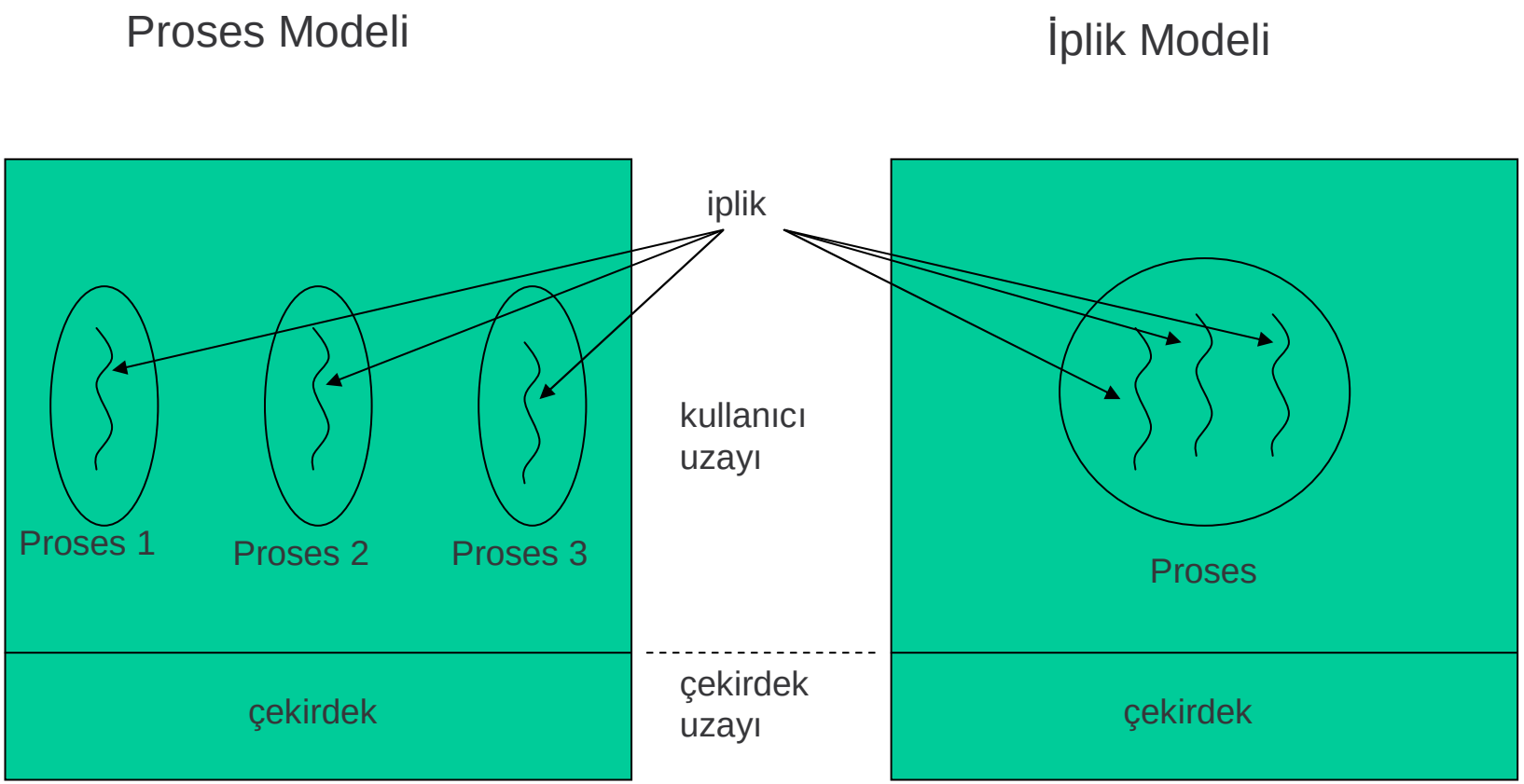
Giriş

- 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 $\Rightarrow$ 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

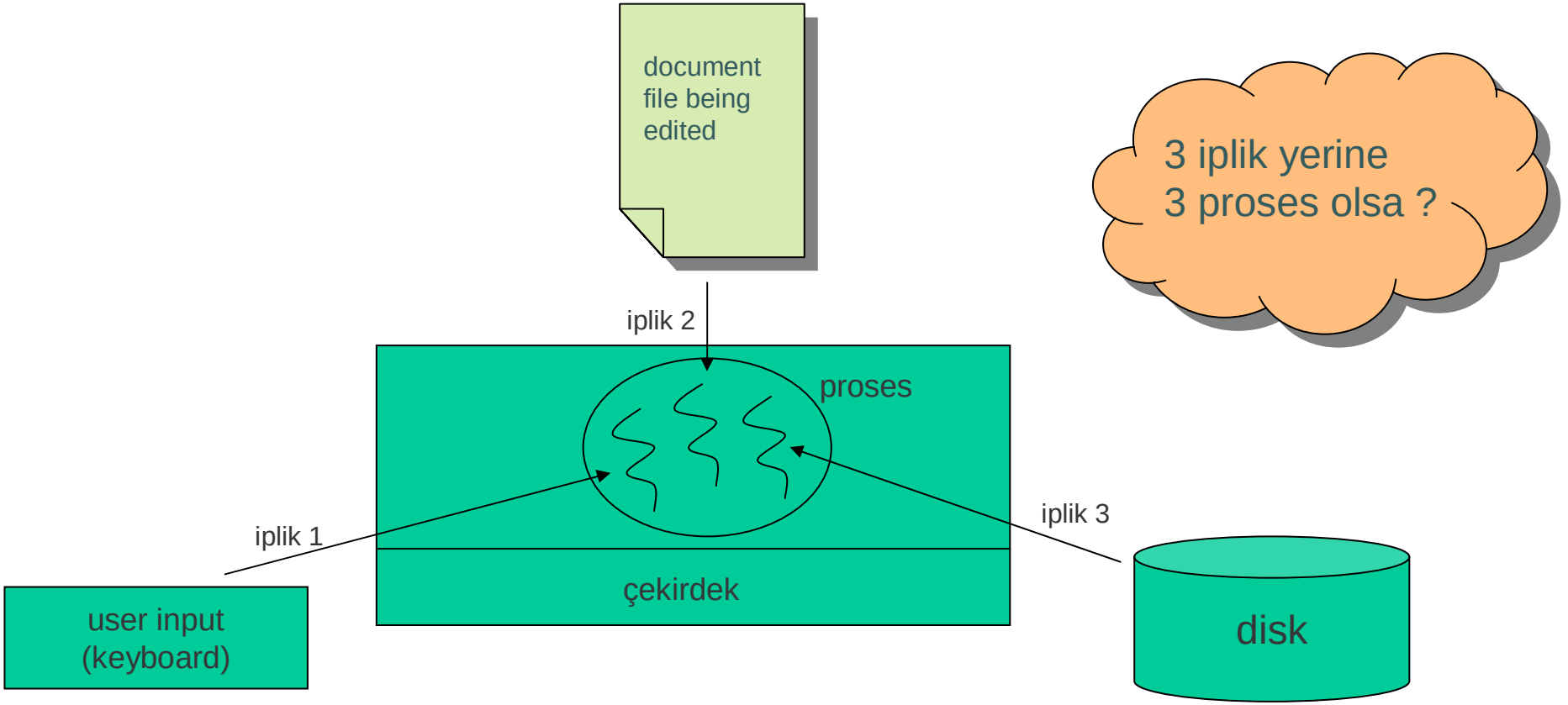
- *(‘sorunlar’ devam)*
  - bir iplik bir dosyayı kapadı ama başka iplik o dosyayı kullanıyordu
  - bir iplik az bellek olduğunu farkedip bellek almaya başladı
    - işlem tamamlanmadan başka iplik çalıştı
    - yeni iplik de az bellek var diye bellek istedi

    $\Rightarrow$ iki kere bellek alınabilir
- çözümler için iyi tasarım ve planlama gerekli

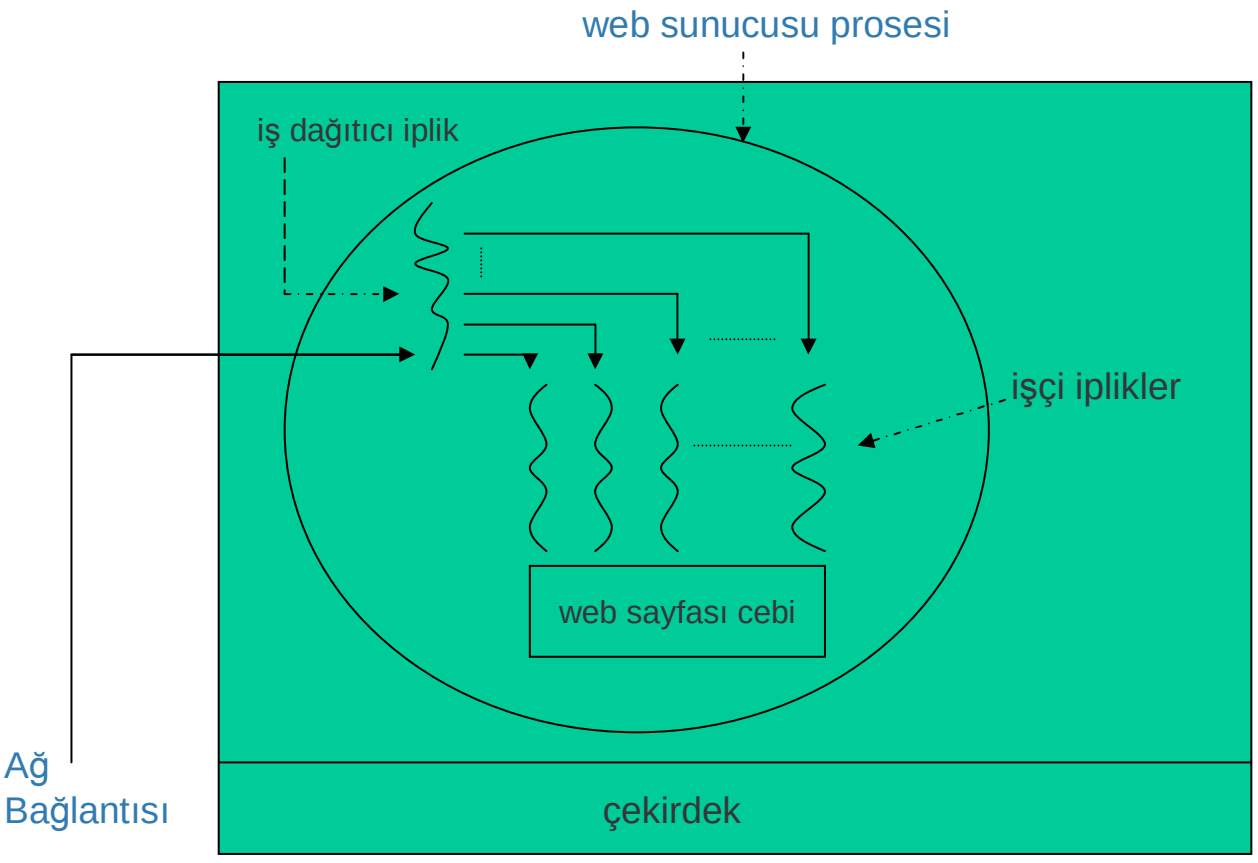
İpliklerin Kullanımı

- 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



web sunucusunda iplikler kullanılmasaydı?

kullanıcı uzayı

çekirdek uzayı

İplik Kullanımına Örnek – Web Sitesi Sunucusu

İş dağıtıcı iplik kodu

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

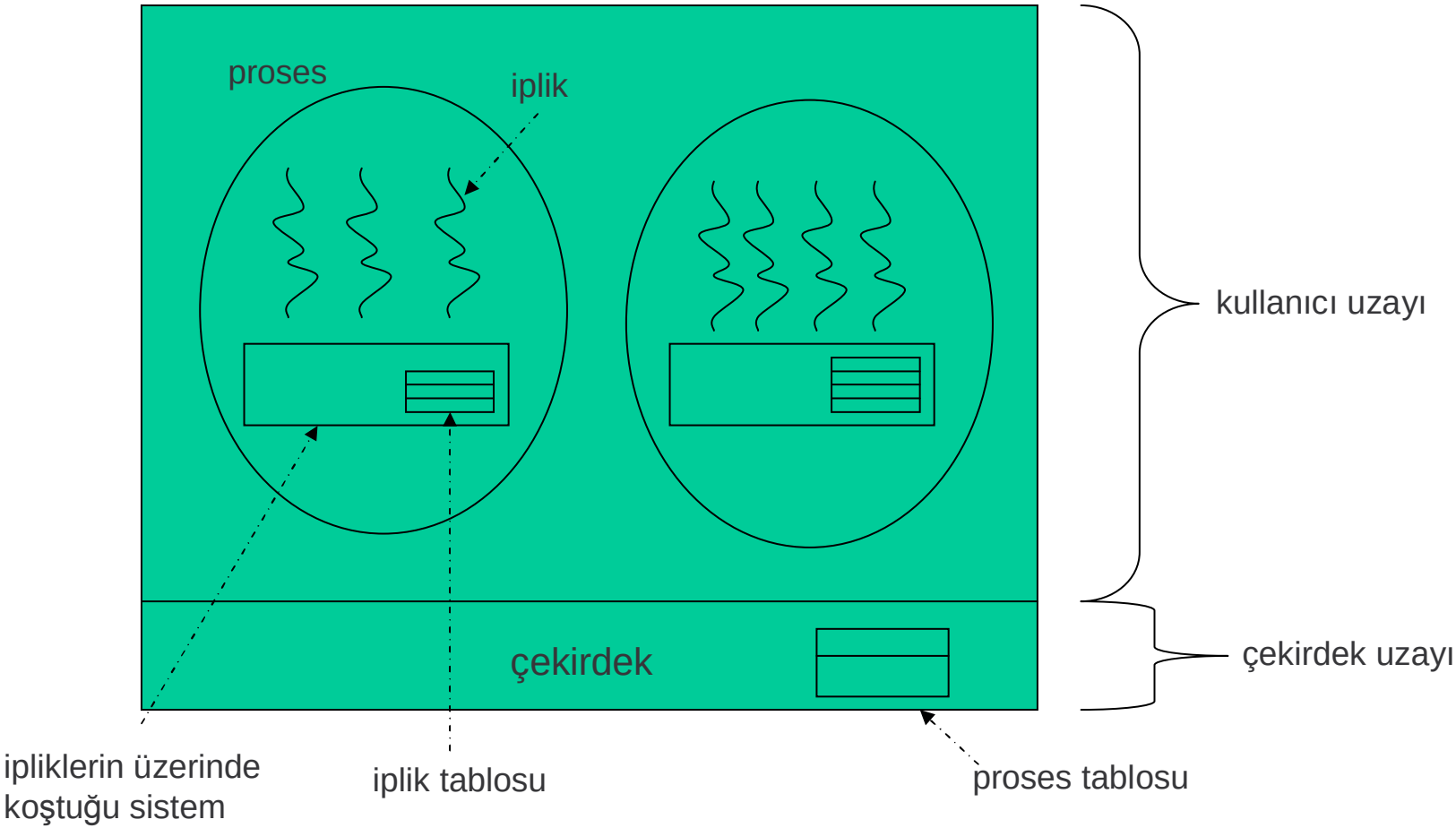
İşçi ipliklerin kodu

```
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 $\Rightarrow$ 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

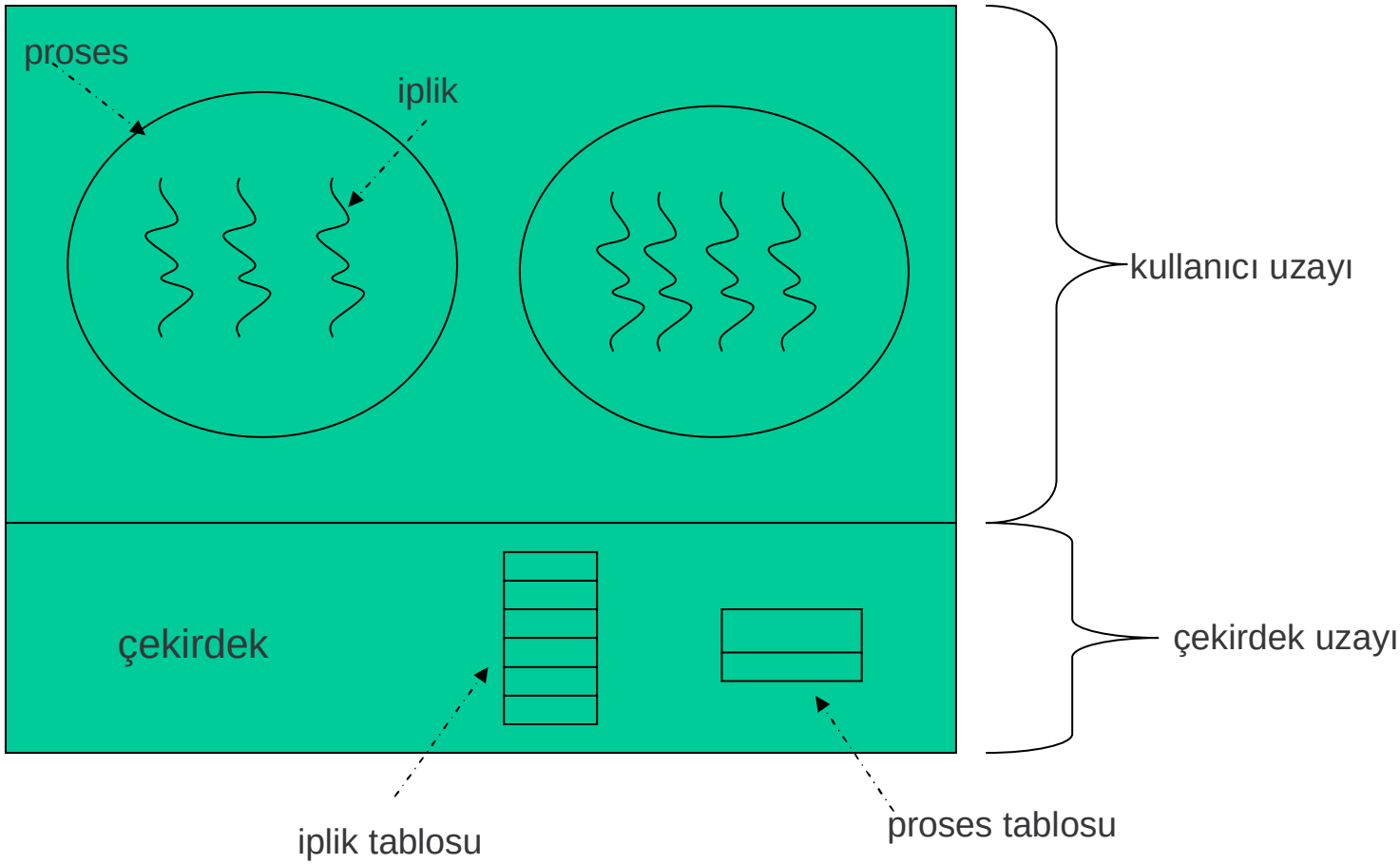
İpliklerin Kullanıcı Uzayında Gerçeklenmesi

- *(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



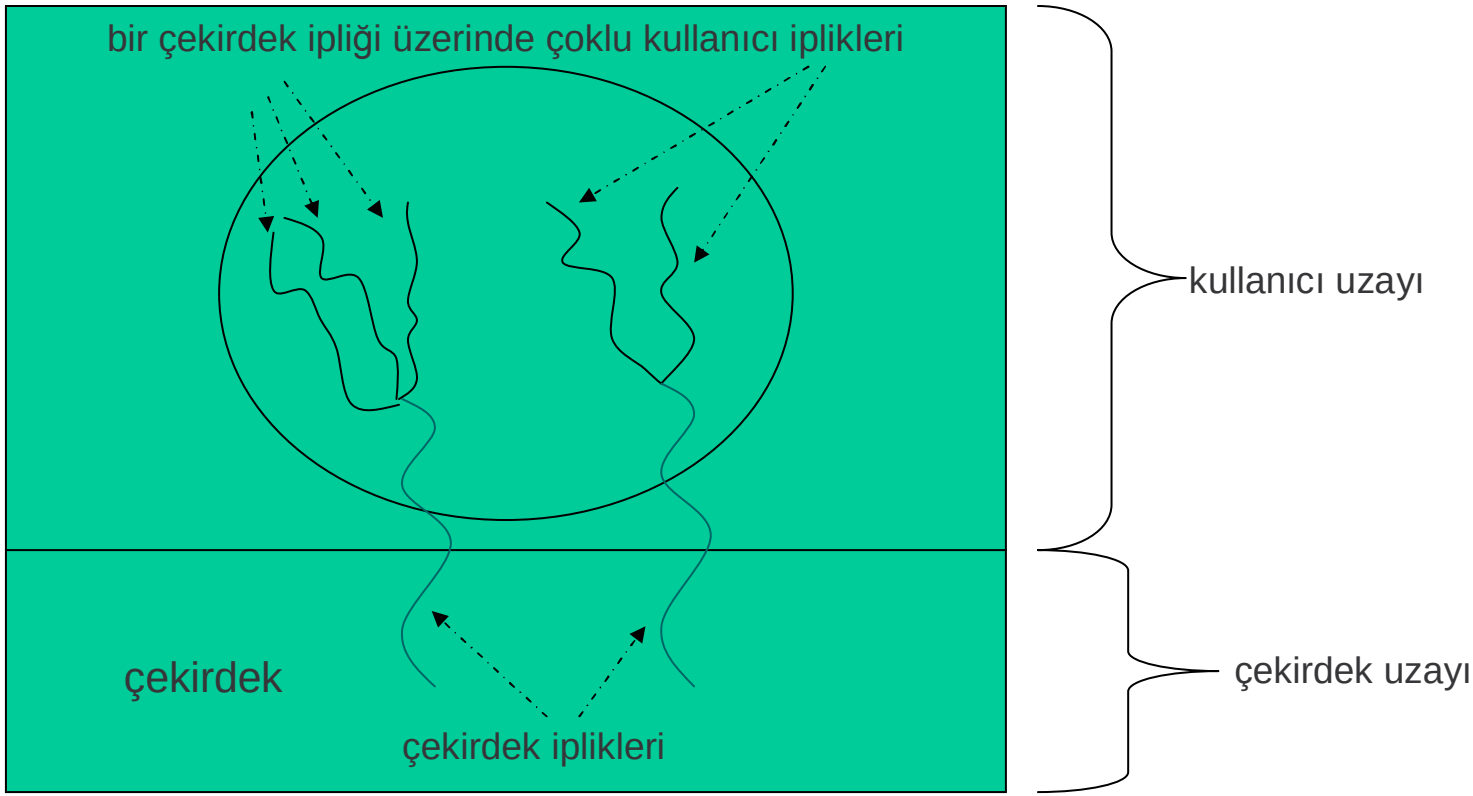
İpliklerin Çekirdek Uzayında Gerçeklenmesi

- 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

UYGULAMA

Creating a Thread


```
#include <pthread.h>
#define NUM_THREADS 5
#define SLEEP_TIME 10

pthread_t tid[NUM_THREADS]; /* array of thread IDs */

void *sleeping(void *); /* thread routine */

void start() {
 int i;
    for ( i = 0; i < NUM_THREADS; i++)
        pthread_create(&tid[i], NULL, sleeping, (void *)SLEEP_TIME);
 }
```

Terminating a Thread

- pthread_join()— Examines exit status
- pthread_exit()— Terminates itself
- pthread_cancel()— Terminates another thread

Summary

Activity	UNIX Mechanism	POSIX Threads Mechanism
Creation	`fork()` and `exec()`	`pthread_create()`
Self-termination	`_exit()`	`pthread_exit()`
Termination by another thread	`kill()`	`pthread_cancel()`
Get unique identifier	`pid = fork()`	`pthread_create(&pid...)`
Mutual exclusion	`fcntl()` and `semop()`	`pthread_mutex_lock()` and `pthread_mutex_unlock()`
Synchronization	`signal()` and `semop()`	`sem_wait()` and `sem_post()`
Change Attributes	`sigaction()` and `nice()`	`pthread_setschedparam()`

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

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

  $\Rightarrow$ ölümcül kilitlenme

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

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

Yarış

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

  $\Rightarrow$ 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

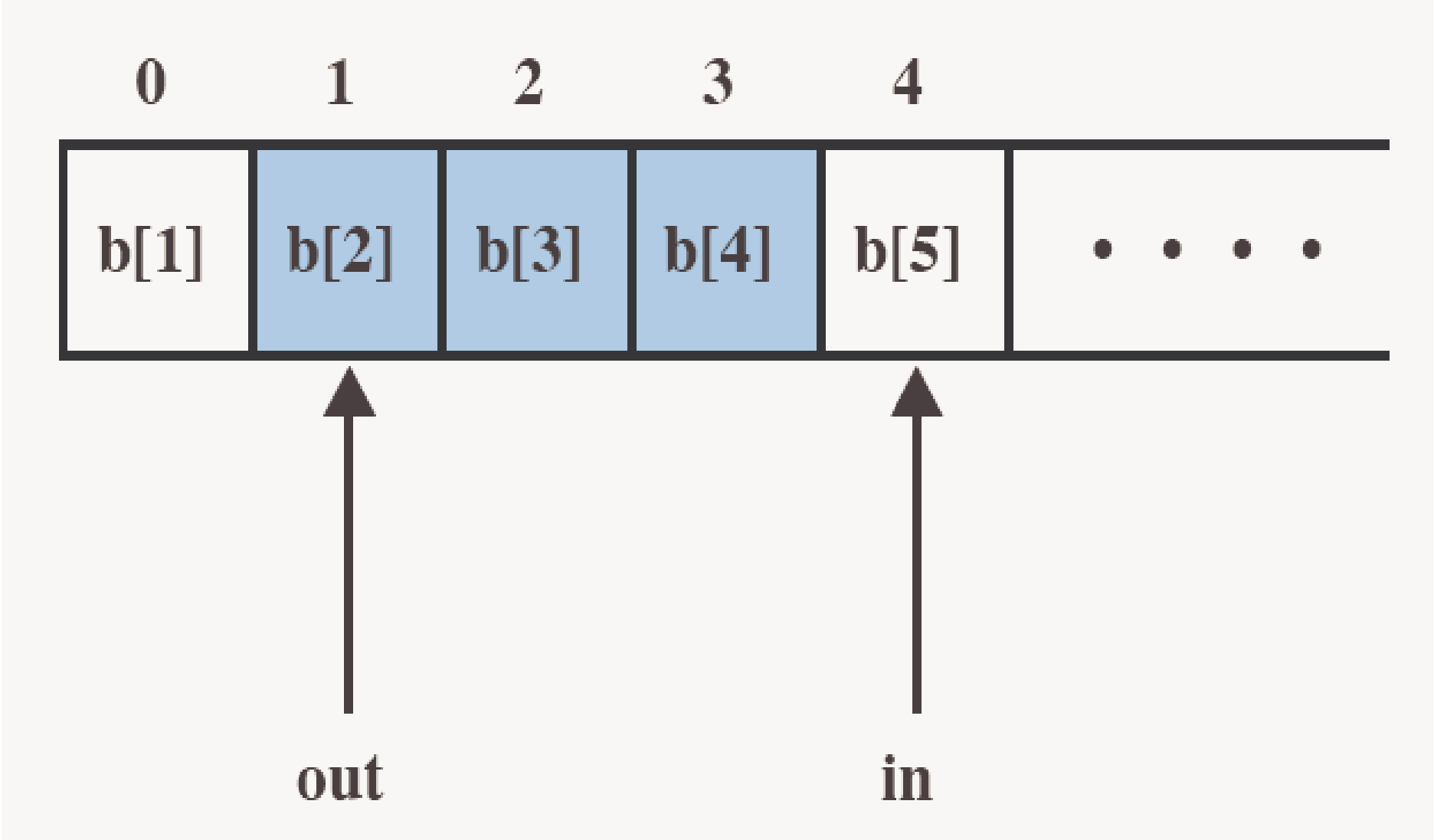
  $\Rightarrow$ senkronizasyon

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

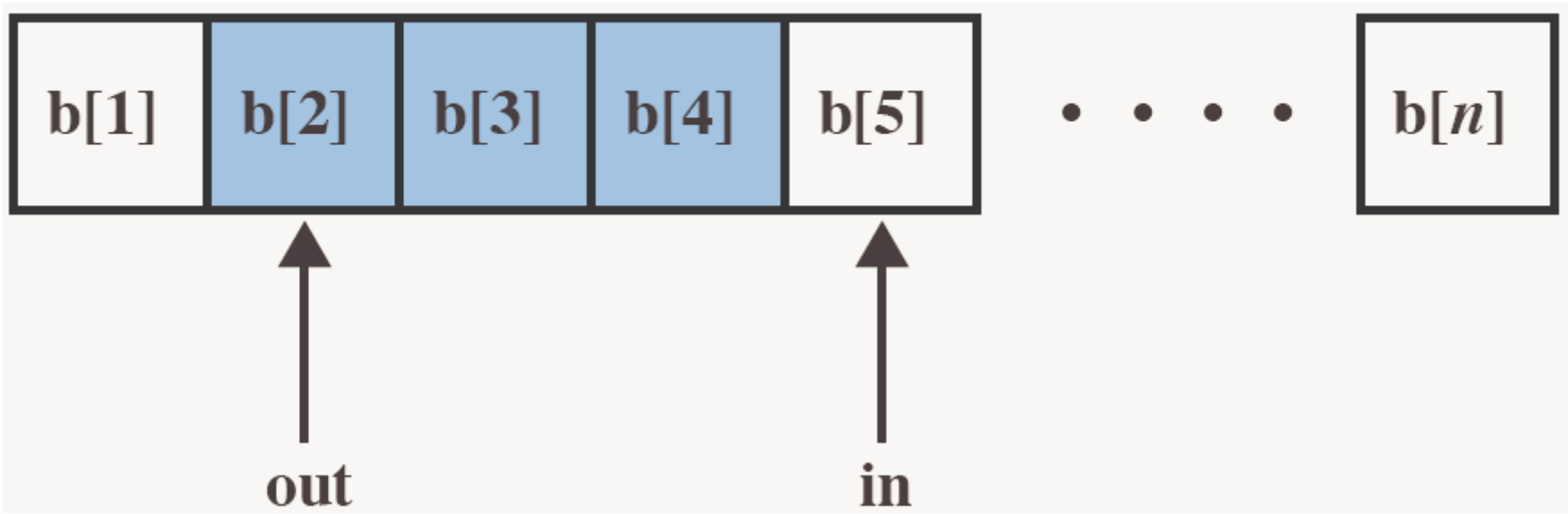


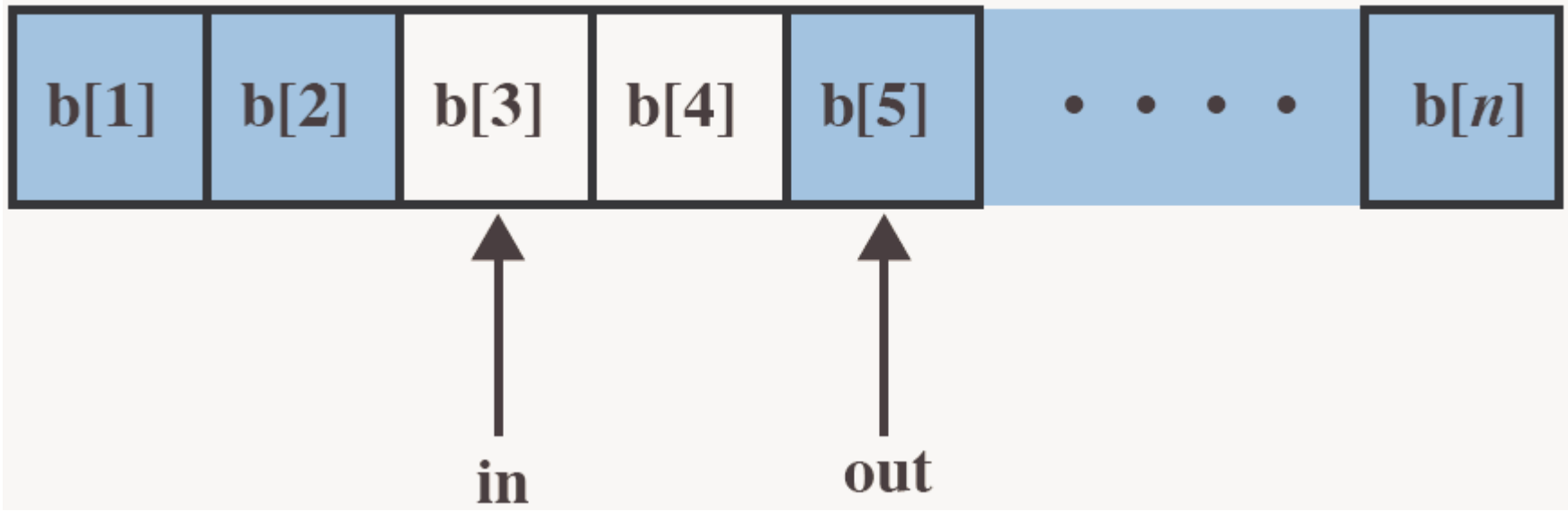
Producer with Circular Buffer

```
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 */
}
```


b[1]
b[2]
b[3]
b[4]
b[5]
b[n]
out
in


b[1]
b[2]
b[3]
b[4]
b[5]
b[n]
in
out

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 $\Rightarrow$ karşılıklı dışlama
- Bir yazıcı dosyaya yazıyorsa okuyucu aynı anda okuyamaz $\Rightarrow$ karşılıklı dışlama

Overview of IPC

- System V Interprocess Communications
- ipcs — Lists all current IPC objects
- ipcrm — Removes all IPC objects
- IPC Objects
  - Message
  - Shared Memory
  - Semaphore

System V Interprocess Communications


```
$ ipcs
IPC status from <running system> as of Fri Sep 3 08:50:16 BST 2006
T ID KEY MODE OWNER GROUP
Message Queues:
q 00xdead-Rrw-rw-rw-marcusginstruct
q 10xbeef-Rrw-rw-rw-marcusginstruct
Shared Memory:
m 00xfeed--rw-rw-rw-marcusginstruct
m 10xdeaf--rw-rw-rw-marcusginstruct
Semaphores:
s 20xcafe--ra-ra-ra-marcusginstruct
s 30xface--ra-ra-ra-marcusginstruct
```

System V Interprocess Communications

```
$ ipcrm -q0 -m1 -s2
$ ipcs
IPC status from <running system> as of Fri Sep 3 08:51:04 BST 2006
T ID KEY MODE OWNER GROUP
Message Queues:
q 10xbeef-Rrw-rw-rw-marcusginstruct
Shared Memory:
m 00xfeed--rw-rw-rw-marcusginstruct
Semaphores:
s 30xface--ra-ra-ra-marcusginstruct
$
```

API Features Common to System V IPC

- key_t key — Used to specify the IPC object at create and get time
- int id — Handle used to access the IPC object
- ftok(3C) — Generates a key from a filename and numeric value
- xxxget(2) — “Get” function that takes a key and returns an id
- xxxctl(2) — “Control” function for setup, delete, admin, and debugging

ftok()

- ftok(3C) function generates a key from a path name, plus some integer value.
- The contents of the file do not determine the key, but rather its ***inode*** number does
- Syntax

```
#include <sys/ipc.h>
key_t ftok (const char *path,int id);
```

The xxxget Function

- The “Get” functions take a key and return an identifier to an IPC resource.
- Each IPC resource has its own “Get” function

```
int msgget(key_t key, int flag);
int shmget(key_t key,size_t size,int flag);
int semget(key_t key,int numsems,int flag);
```

Example

```
key_t key;
int id;
key= ftok(“/home/kurt/keyfile”,1);
…
id= msgget(key,flag);
```

The xxxctl Function

- The “ctl” functions takes commands for setting up, deleting, doing administration, and debugging.
- Each IPC resource has its own “ctl” function

```
int msgctl(id, IPC_RMID,(struct msqid_ds *)NULL);
int shmctl(id, IPC_RMID, (struct shmid_ds *)NULL);
int semctl(id,ignored, IPC_RMID,(union semun *)NULL);
```

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:
    donanım/yazılım kesmesi, örneğin sıfıra bölme
  - bir proses bir başka prosese yollayabilir, kill()
  - kullanıcı bir proses yollayabilir, kill komutu yada CTRL+C
- sinyal alınınca yapılacak işler tanımlanır
- öntanımlı işleri olan sinyaller var
  - öntanımlı işler değiştirilebilir
  - SIGKILL (9 no.lu) sinyali yakalanamaz

Sistem Tarafından Üretilen Sinyaller

- Sistem, proses yanlış bir işlem yürüttüğünde ona bir sinyal gönderir
- Örnekler
  - SIGSEGV – Segmantation Violation
  - SIGBUS – Bus Error (trying to access bad address)
  - SIGFPE – FP Exception (ex. divide by zero)

Kullanıcı Tarafından Üretilen Sinyaller

- Kullanıcı, tuş takımını kullanarak prosese sinyal gönderebilir
- Örnekler
  - SIGINT – Interrupy (Control-C)
  - SIGQUIT – Quit (Control-\)
  - SIGTSTP – Stop (Control-z)

Proses Tarafından Üretilen Sinyaller

- Bir proses başka bir prosese *kill*() sistem çağrısını kullanarak sinyal gönderebilir
- Örnekler
  - SIGKILL – Kill (herzaman prosesin sonlanmasına neden olur)
  - SIGTERM – Software termination (SIGKILL'e göre daha temiz)
  - SIGUSR1 ve SIGUSR2– Kullanıcı tanımlı sinyaller, uygulama geliştirirken faydalı

Prosesin Sinyali Aldığındaki Davranışı

- Varsayılan Davranışlar
  - Boş ver
  - Sonlan
  - Dur/Devam Et
- Sinyali yakala ve işle
- Sinyalleri Maskelemek
- Sinyalleri Bloke Etmek

Sending Signals

- kill()— Sends a signal to a process or process group
- sigsend()— Sends a signal to a process or group of processes
- raise()— Enables a process to send a signal to itself
- abort()— Sends a message to kill the current process

Linux'da Sinyal Mekanizması


```
<sys/types.h>
<signal.h>
int kill(pid_t pid,int sig);
```

Example: kill()

```
#include <sys/types.h>
#include <signal.h>
#include <unistd.h>
main(){
    kill(getppid(),SIGUSR1);
}
```

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;
}
```

Sending Signals to Process Group: sigsend()


```
<sys/types.h>
<signal.h>
int sigsend(idtype_t type,id_t id, int
  sig);
```

Example: sigsend()

```
#include <sys/types.h>
#include <signal.h>
#include <unistd.h>
main(){
    sigsend(P_PGID,getpgid(getpid()),SIGUSR1);
}
```

raise()

- Prosesin kendisine sinyal göndermesini sağlar

```
<sys/types.h>
<signal.h>
int raise(int sig);
```

abort()

- Prosesin kendisine SIGABRT sinyalini göndermesini sağlar

```
<sys/types.h>
<signal.h>
int abort();
```

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
```

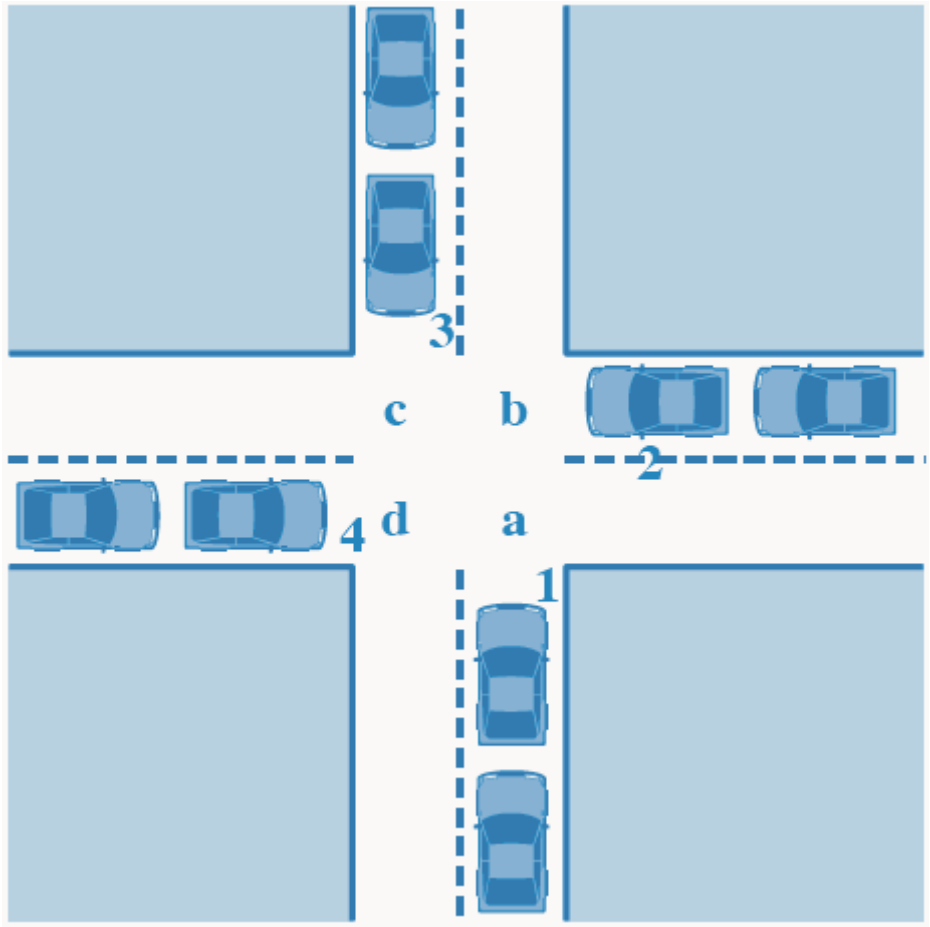
Linux'da Paylaşılan Bellek Mekanizması


```
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;
}
```

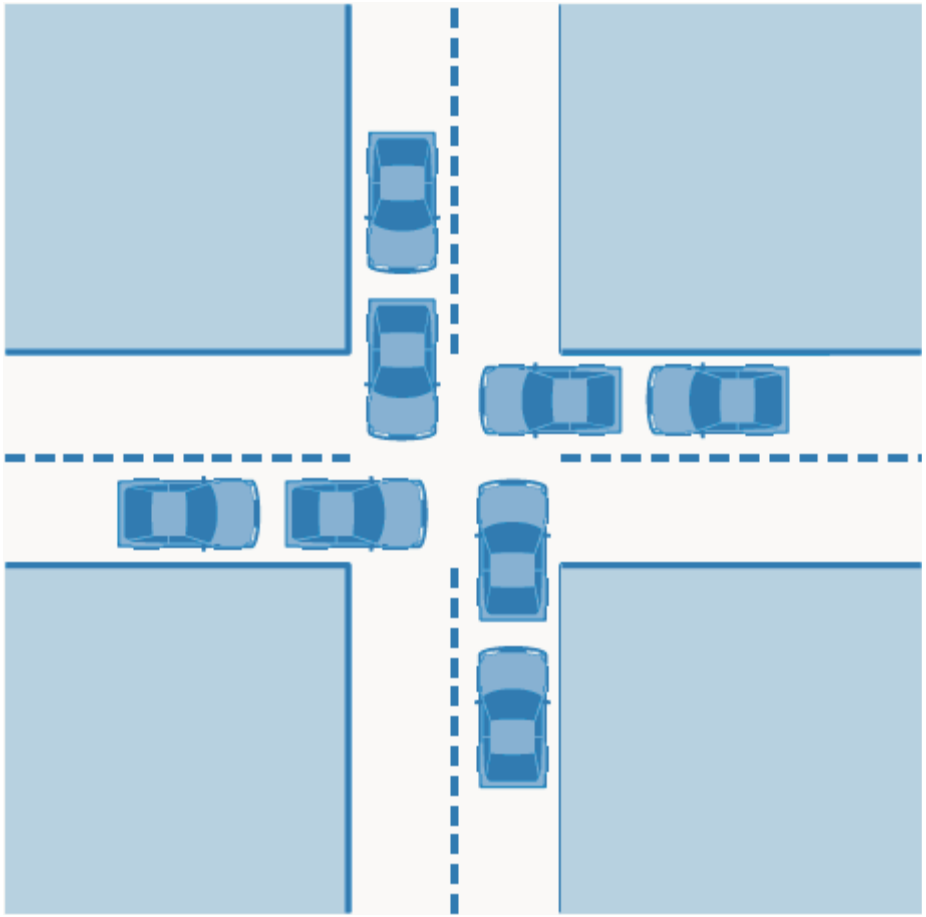
5 ÖLÜMCÜL KİLİTLENME

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

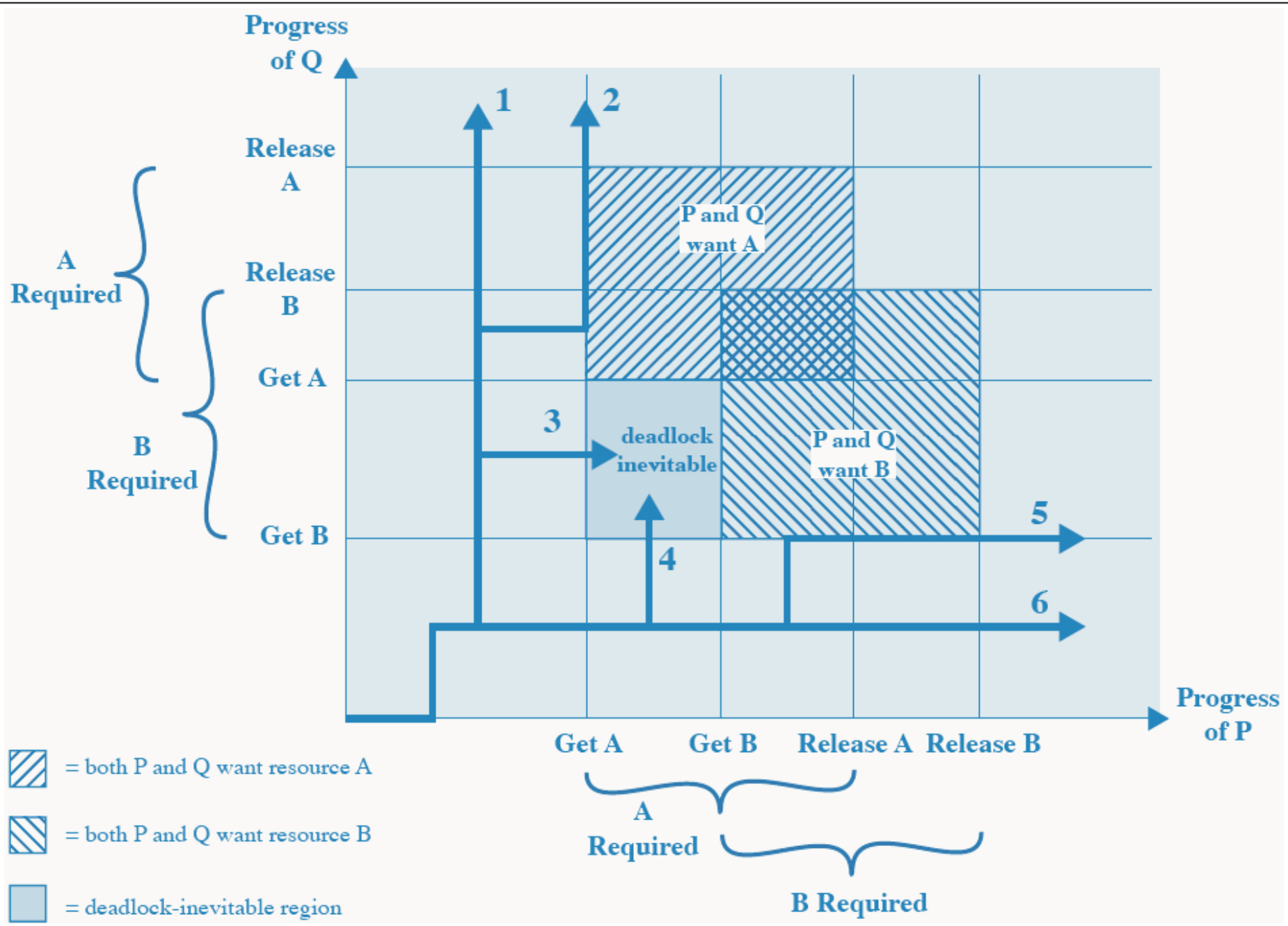
Ölümcül Kilitlenme Örneği - 1

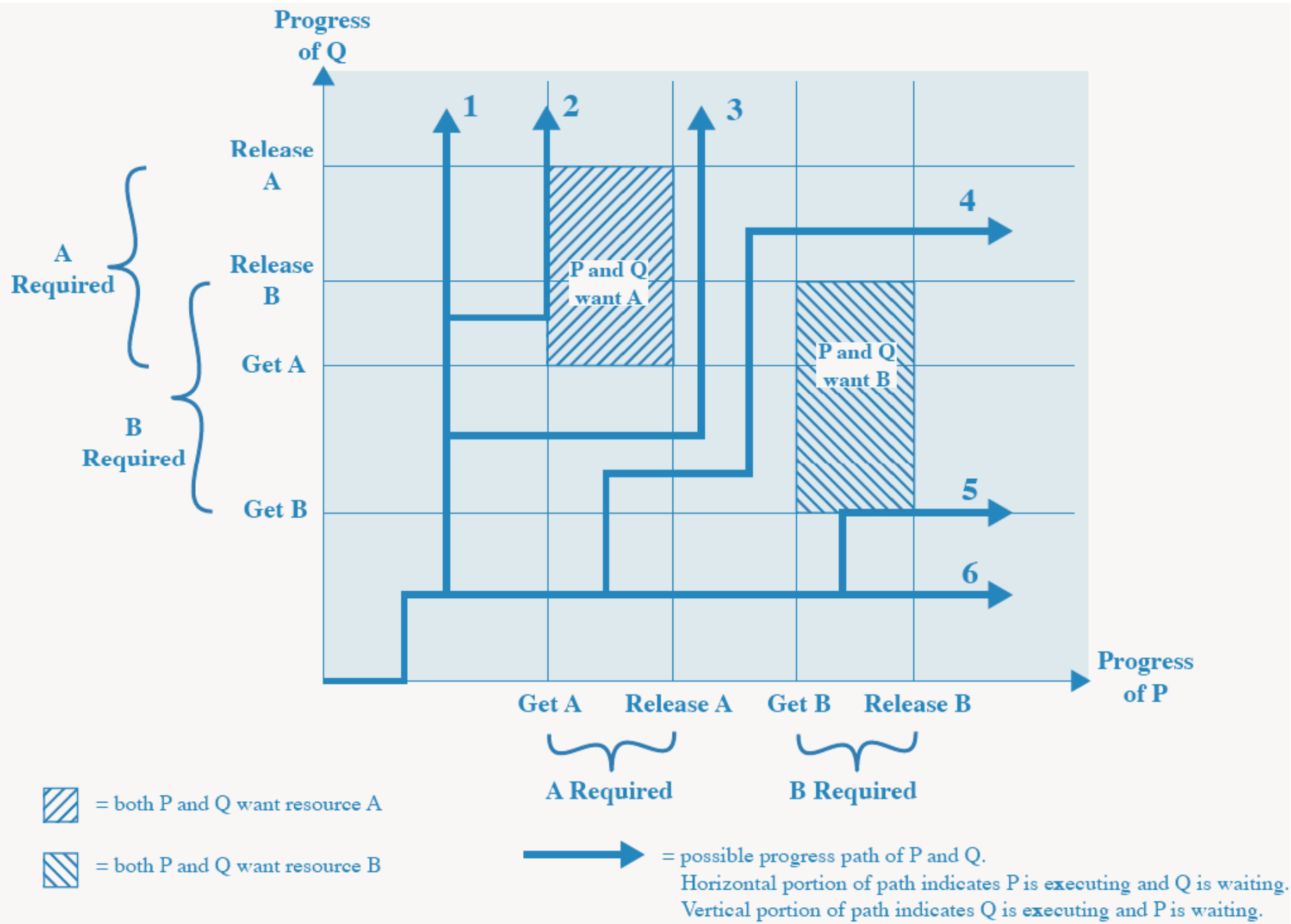
Process P

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
P and Q want B
deadlock inevitable
Get A
Get B
Release A
Release B
Progress of P
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
80K sekizli iste
...
60K sekizli iste

P2 Prosesi
70K sekizli iste
...
80K sekizli iste

Ölümcül Kilitlenme Örneği - 3

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

P1 Prosesi

```
Al (P2);
...
Gönder (P2,M1);
```

P2 Prosesi

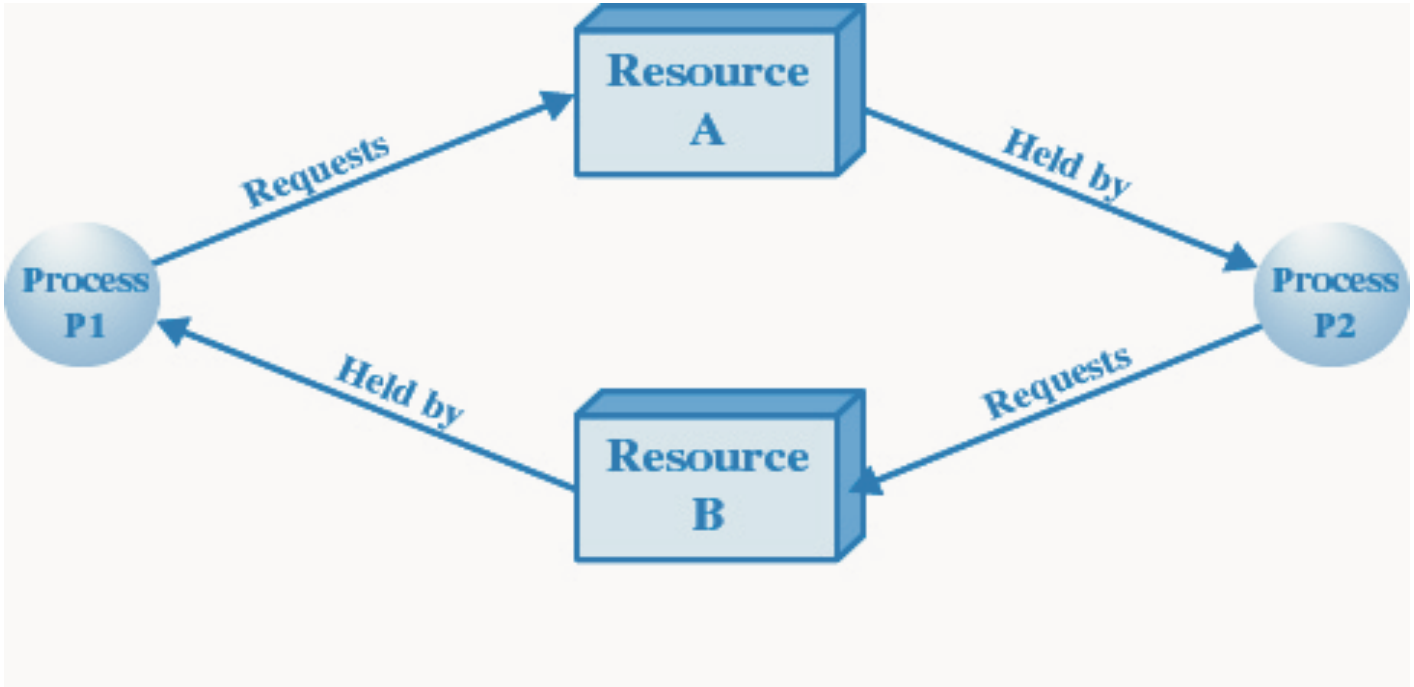
```
Al (P1);
...
Gönder (P1,M2);
```

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={$P_1 \rightarrow R_1$, $P_2 \rightarrow R_3$, $R_1 \rightarrow P_2$, $R_2 \rightarrow P_2$, $R_2 \rightarrow P_1$, $R_3 \rightarrow P_3$}

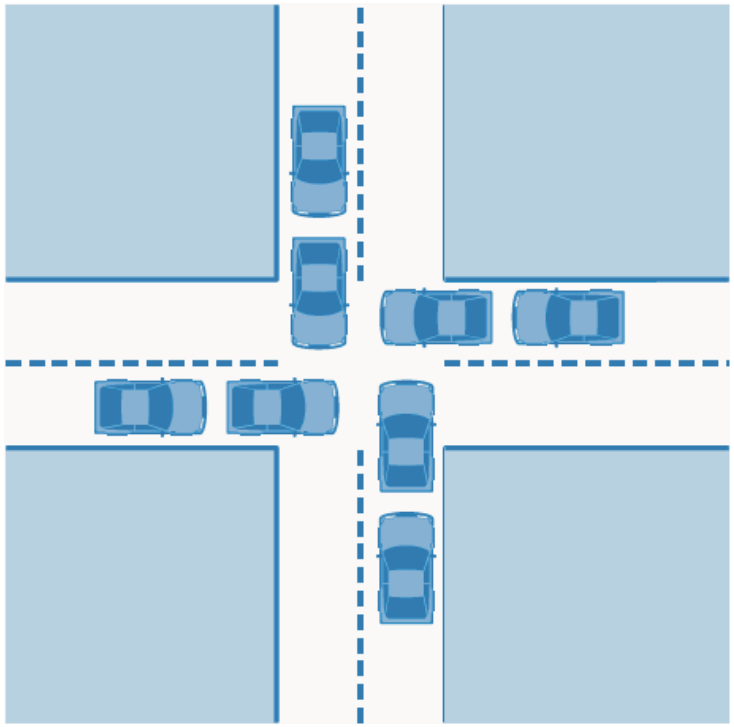
Kaynaklar:

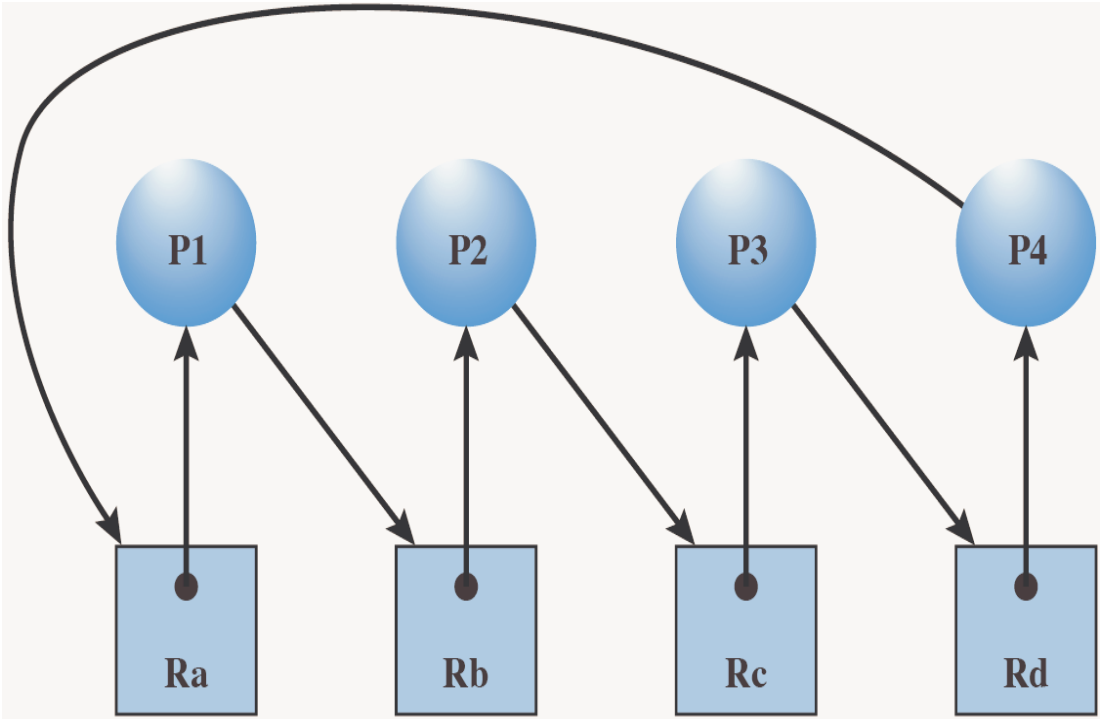
- 1 adet R_1
- 2 adet R_2
- 1 adet R_3
- 3 adet R_4

Proses Durumları:

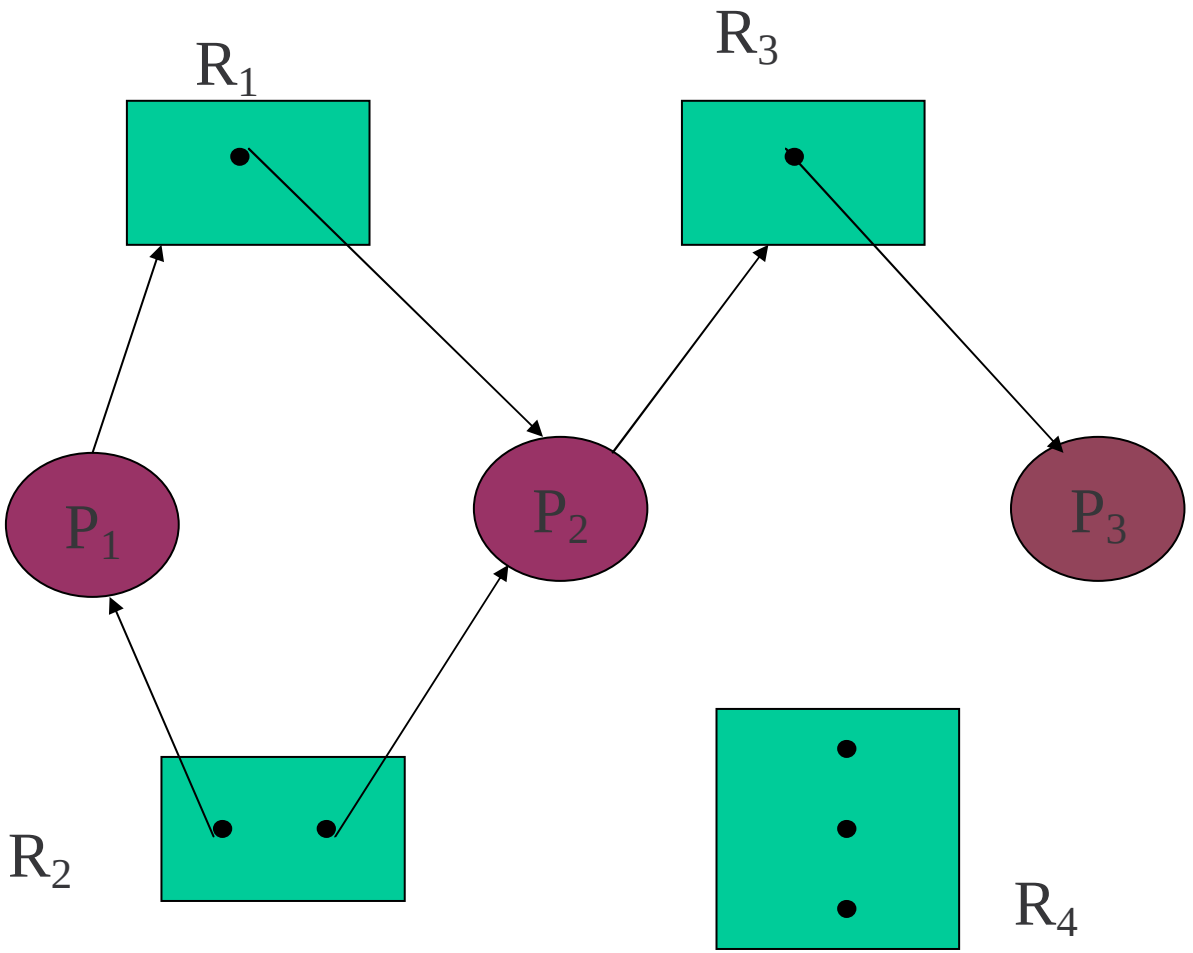
- P_1: bir adet R_2 elinde tutuyor, bir adet R_1 istiyor
- P_2: birer adet R_1 ve R_2 elinde tutuyor, bir adet R_3 istiyor
- P_3: bir adet R_3 elinde tutuyor

Örnek

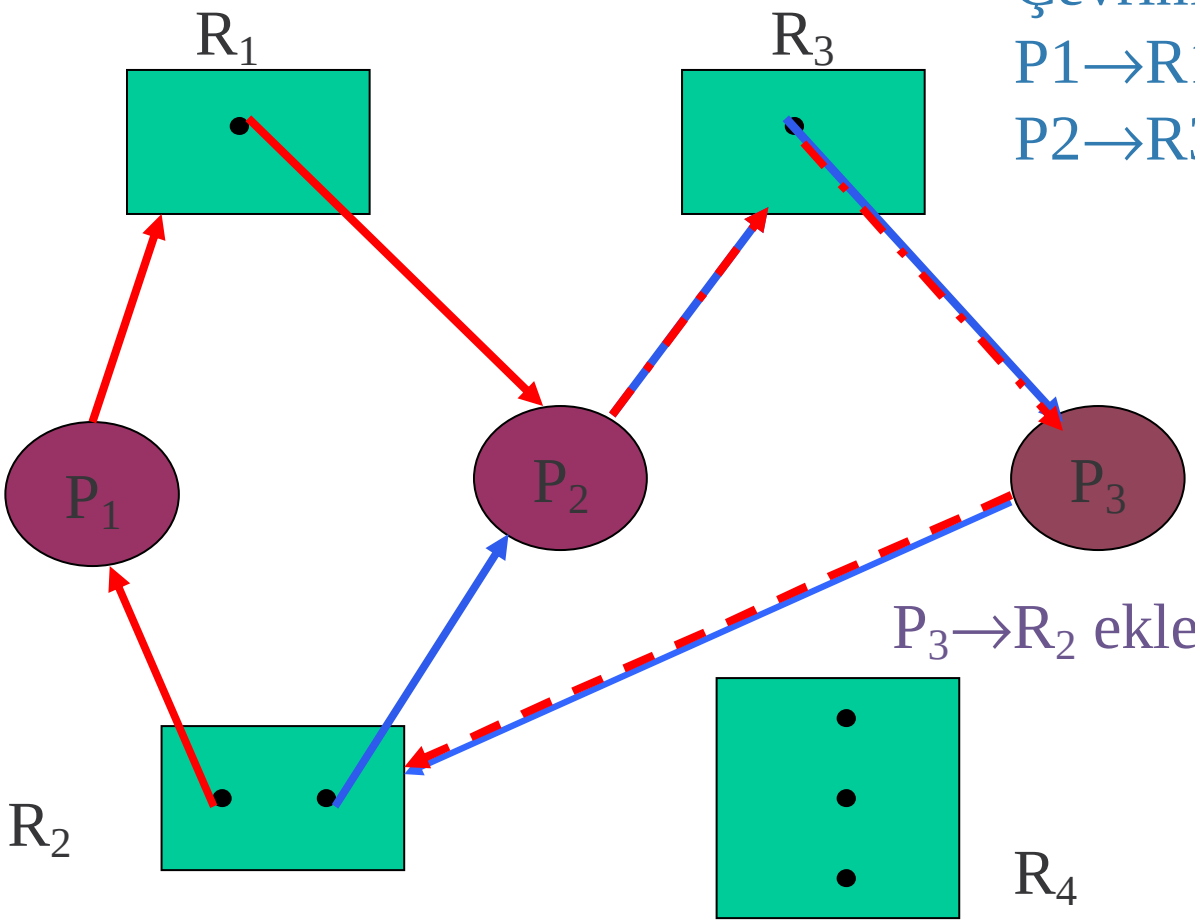




Kaynak Atama Grafı



Kaynak Atama Grafı

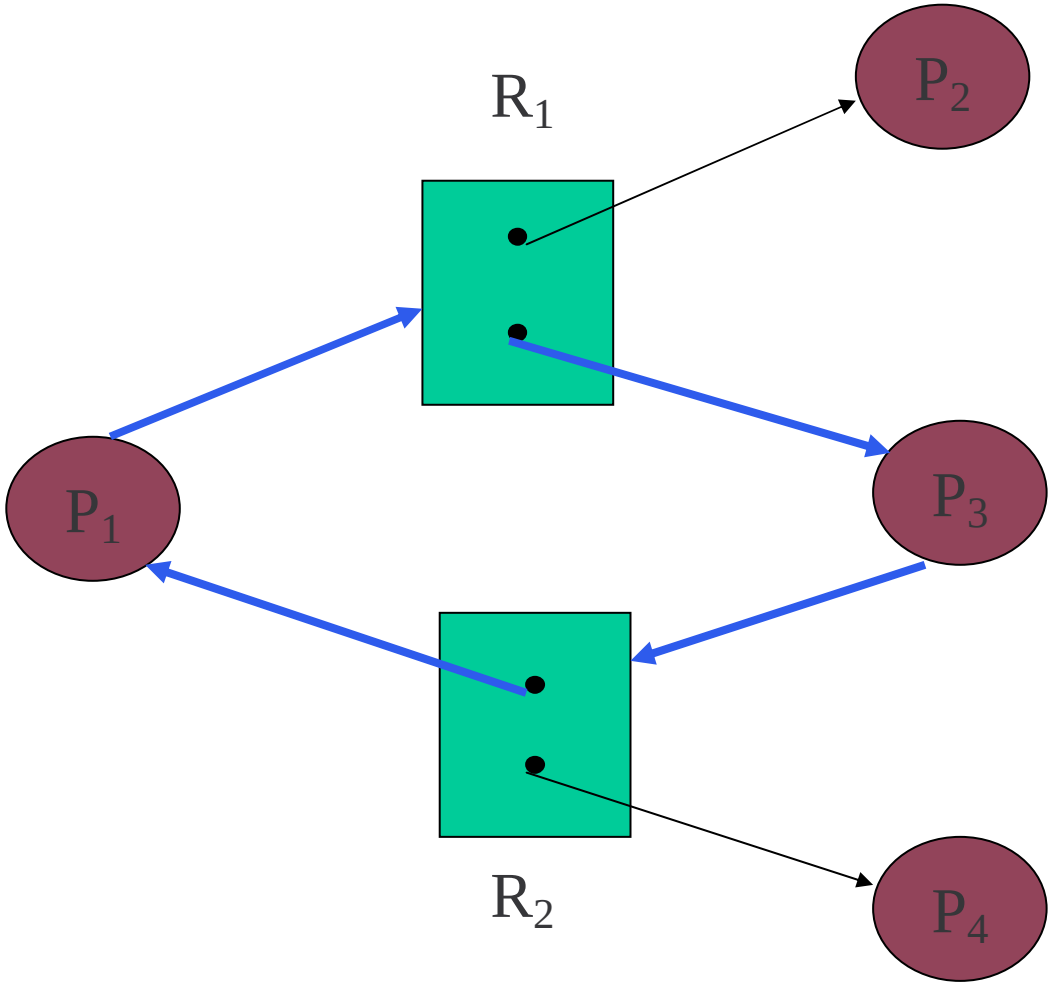


Çevrimler:
P1→R1→P2→R3→P3→R2→P1
P2→R3→P3→R2→P2

$P_3 \rightarrow R_2$ 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?

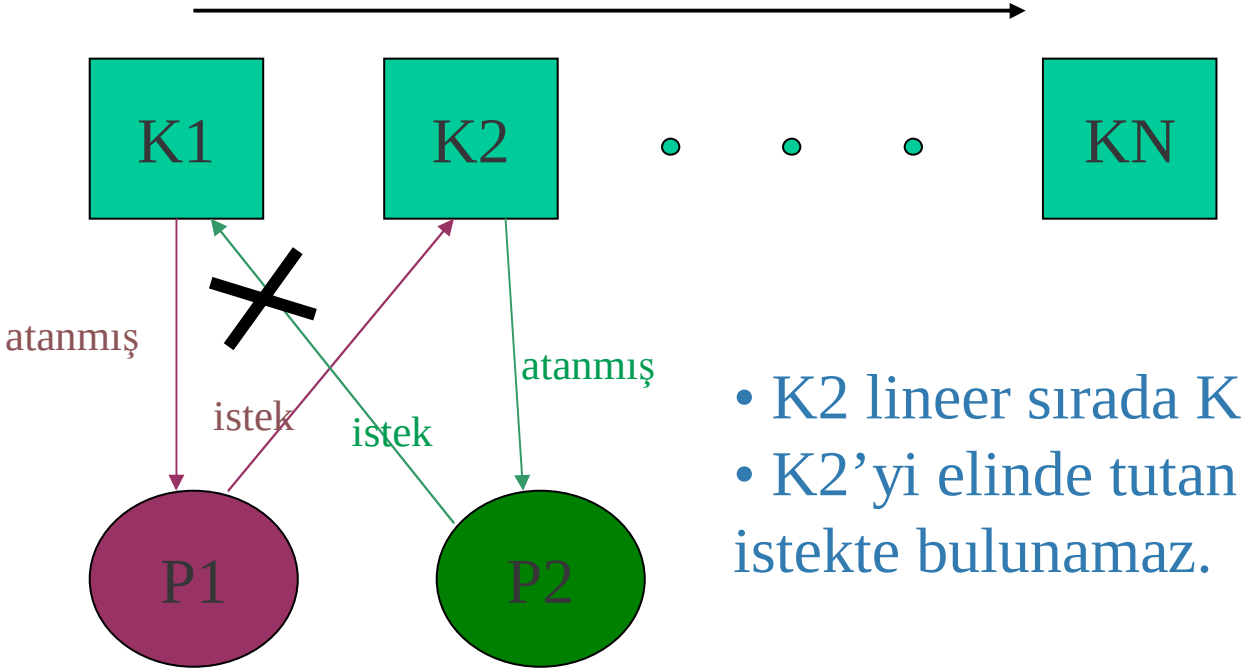
Ölümcül Kilitlenmeyi Önleme

- **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 ⇒ 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)
- **E*min 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 ⇒ Emin Durum √

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.

$$W_k = W_k + A_{ik} \quad , 1 \leq k \leq m$$

- **Adım 6:** Adım 3'e dön.
- Algoritma sonlandığında işaretsiz proses varsa $\Rightarrow$ Ölümcül kilitlenme var.

Ölümcül Kilitlenmeyi Sezme

- Algoritma sonlandığında işaretsiz proses varsa $\Rightarrow$ 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 isaretsiz ⇒ kilitlenmişler

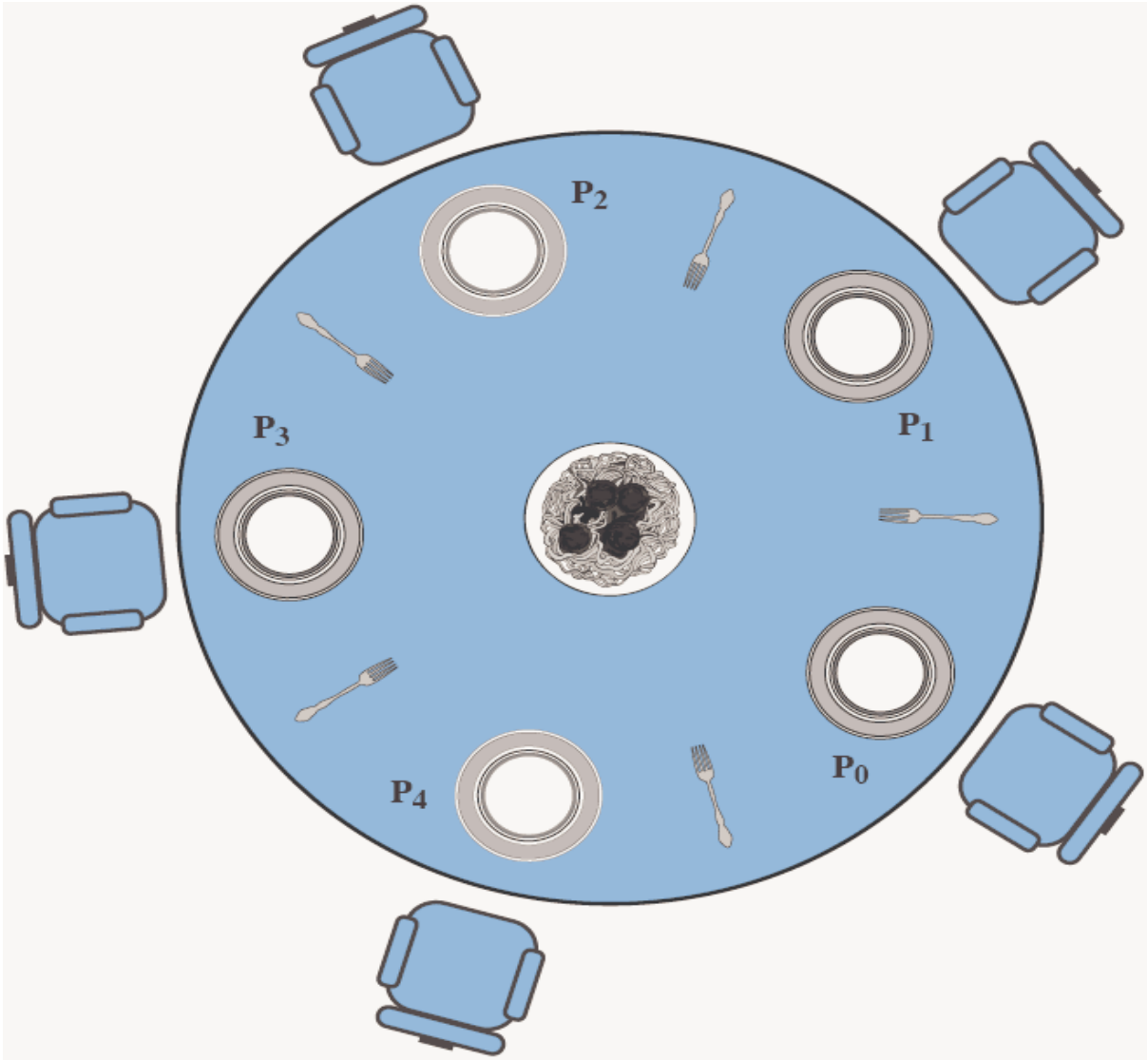
Ölümcül Kilitlenme Sezildikten Sonra Yapılabilecekler

- 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



6 İŞ SIRALAMA

İş Sıralama

- Çok programlı ortamlarda birden fazla proses belirli bir anda bellekte bulunur
- Çok programlı ortamlarda prosesler:
  - işlemciyi kullanır
  - bekler
    - giriş çıkış bekler
    - bir olayın olmasını bekler

İş Sıralama

- Çok programlı ortamlarda işlemcinin etkin kullanımı için iş sıralama yapılır
- İş sıralama işletim sisteminin temel görevleri arasında
  - işlemci de bir sistem kaynağı

İş Sıralamanın Amaçları

- İşleri zaman içinde işlemciye yerleştirmek
- Sistem hedeflerine uygun olarak:
  - İşlemci verimi
  - Cevap süresi (response time)
  - Debi (throughput)

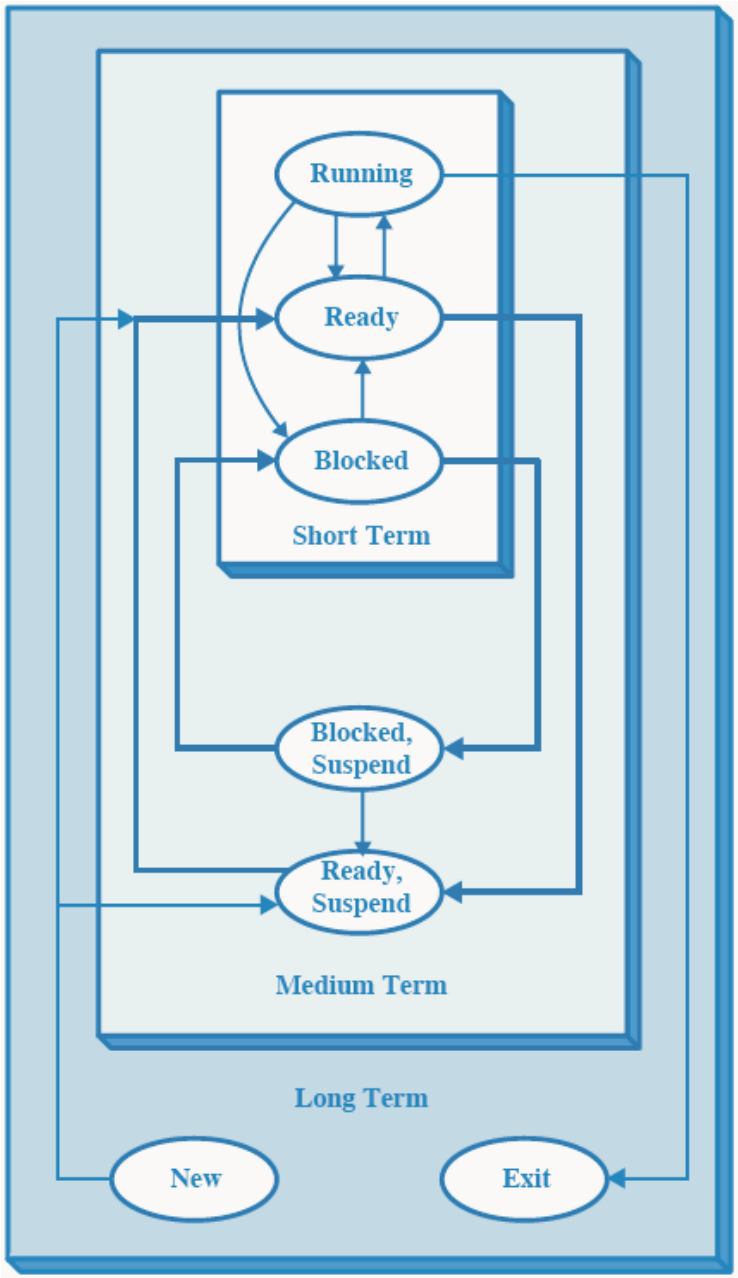
İş Sıralama Türleri - 1

- Uzun vadeli iş sıralama: Yürütülecek prosesler havuzuna ekleme kararı
- Orta vadeli iş sıralama: Kısmen veya tamamen bellekte yer alacak prosesler arasından alma kararı

İş Sıralama Türleri - 2

- Kısa vadeli iş sıralama: Koşabilir durumdaki proseslerden hangisinin seçileceği kararı
- G/Ç sıralama: G/Ç bekleyen hangi prosesin isteğinin karşılanacağı kararı

İş Sıralama Türleri



Uzun Vadeli İş Sıralama

- İşlenmek üzere sisteme hangi proseslerin alınacağına karar verilmesi
- Çoklu programlamanın düzeyini belirler

Uzun Vadeli İş Sıralama

- Sisteme alınan iş:
  - bazı sistemlerde kısa vadeli sıralama için ilgili kuyruğa yerleşir
  - bazı sistemlerde orta vadeli sıralama için ilgili kuyruğa yerleşir (yeni gelen proses doğrudan belleğe alınmaz)

Uzun Vadeli İş Sıralama

- İki tip karar söz konusu:
- sisteme yeni proses eklenebilir mi?
  - çoklu programlama düzeyi tarafından belirlenir
  - Proses sayısı arttıkça her çalışmada proses başına düşen işlemci zamanı azalır : bu nedenle bu sınırlanabilir

Uzun Vadeli İş Sıralama

– uzun vadeli iş sıralama,
  - sistemde herhangi bir iş sonlandığında çalışabilir
  - işlemcinin boş bekleme zamanı bir eşik değerini aşarsa çalışabilir

Uzun Vadeli İş Sıralama

- hangi prosesleri kabul etmeli? hangilerini etmemeli?
  - geliş sıralarına göre seçilebilir
  - bazı kriterlere göre seçilebilir
    - öncelik
    - beklenen toplam çalışma süresi
    - giriş/çıkış istekleri

Uzun Vadeli İş Sıralama

Örneğin gerekli bilgi varsa,

- G/Ç yoğun işlerle işlemci yoğun işleri dengeli tutmak istenebilir
- G/Ç isteklerini dengeli tutmak için istenen kaynaklara göre karar verilebilir

Uzun Vadeli İş Sıralama

- Etkileşimli kullanıcılar (zaman paylaşımlı bir sistemde)
  - sisteme bağlanma istekleri oluşur
  - bekletilmez
  - belirli bir doyma noktasına kadar her gelen kabul edilir
  - doyma değeri belirlenir
  - sistem doluysa kabul edilmeyip hata verilir

Orta Vadeli İş Sıralama

- "Swap" işleminin bir parçası
- Çoklu programlamanın derecesini belirleme gereği
- Sistemin bellek özellikleri ve kısıtları ile ilgilidir
- Belleğe alınacak proseslerin bellek gereksinimlerini göz önüne alır
- Bellek yönetim birimine de bağlıdır

Kısa Vadeli İş Sıralama

- Belirli sistem başarım kriterlerini gerçekleme amacı
- Algoritmaların başarım kıyaslaması için kriterler belirlenir

Kısa Vadeli İş Sıralama

- İş Çalıştırıcı (dispatcher)
- Bir olay olduğunda çalışır
  - Saat kesmesi
  - G/Ç kesmesi
  - İşletim sistemi çağrıları
  - Sinyaller

Kısa Vadeli İş Sıralama Kriterleri

- Kullanıcıya yönelik
  - Yanıt süresi: isteğin iletilmesinden çıkış alınana kadar geçen süre
  - Amaç kabul edilen yanıt süresi ile çalışan iş/kullanıcı sayısını maksimize etmek
  - doğrudan kullanıcılara iyi servis vermek amaçlı

Kısa Vadeli İş Sıralama Kriterleri

- Sisteme yönelik
  - İşlemcinin etkin kullanımı
  - Örnek: debi (işlerin tamamlanma hızı)
    - önemli bir kriter ancak kullanıcı prosesler doğrudan bunu göremez
    - sistem yöneticileri açısımdan daha önemli bir kriter

Kısa Vadeli İş Sıralama Kriterleri

- Başarıma yönelik
  - Nicel
  - Debi ve yanıt süresi gibi ölçülebilir
- Başarım ile ilgili olmayan
  - Nitel ve çoğu durumda ölçülemez
  - Tahmin edilebilirlik: kullanıcılara sunulan hizmetin, diğer kullanıcılardan bağımsız olarak her zaman belirli bir düzeyde olması istenir

Kısa Vadeli İş Sıralama Kriterleri

- Tüm kriterler birden sağlanamayabilir
  - bir kriterin sağlanamaması bir diğerini engelleyebilir
    - örneğin yanıt cevabını kısa tutma amaçlı bir algoritmaya göre işler sık sık değiştirilir. Bu sisteme yük getirir ve debi düşer.

Önceliklerin Kullanımı

- İş sıralayıcı yüksek öncelikli işleri önce seçer
- Her öncelik düzeyine ilişkin *Hazır* kuyruğu tutar
- Düşük öncelikli prosesler *Açlık* durumu ile karşılaşabilir
  - prosesin önceliğinin yaşı ve geçmişi ile bağlantılı olarak değiştirilmesi

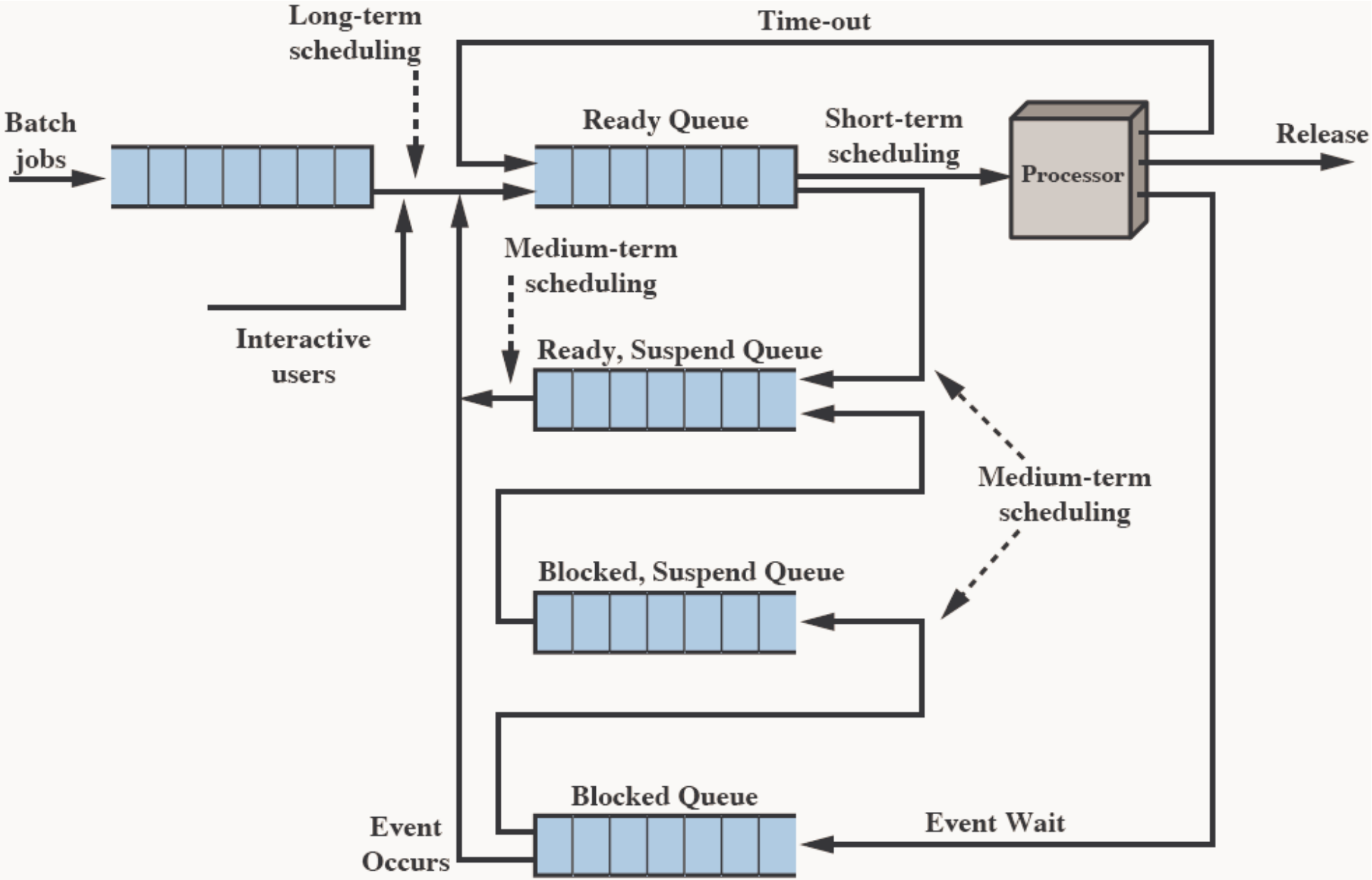
İş Sıralama

- çalışan proses ortasında kesilip “hazır” kuyruğuna eklenebilir (preemptive)
  - çalışma anları:
    - yeni bir iş gelince
    - bloke olan bir iş hazır olunca
    - belirli zaman aralıkları ile
  - sisteme yük getirir ama proseslere hizmet kalitesi artar

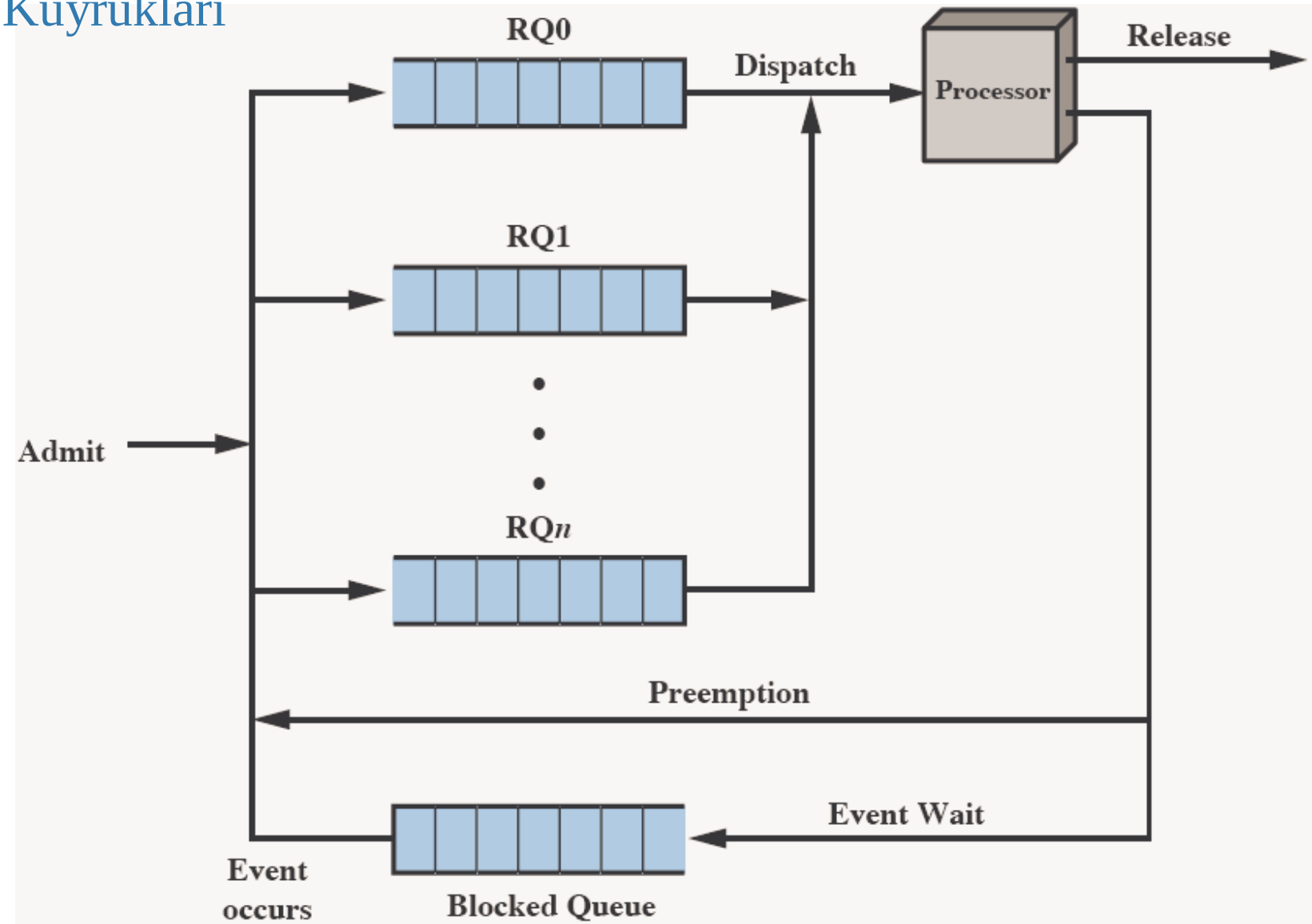
İş Sıralama

- işlemcide koşmakta olan proses
  - sonlanana
  - bloke olana

  kadar kesilmeden koşar (nonpreemptive).


Long-term scheduling
Time-out
Batch jobs
Ready Queue
Short-term scheduling
Processor
Release
Medium-term scheduling
Interactive users
Ready, Suspend Queue
Medium-term scheduling
Blocked, Suspend Queue
Blocked Queue
Event Occurs
Event Wait

Öncelik Kuyrukları



İş Sıralama Örneği

Proses	Varış Zamanı	Servis Süresi
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

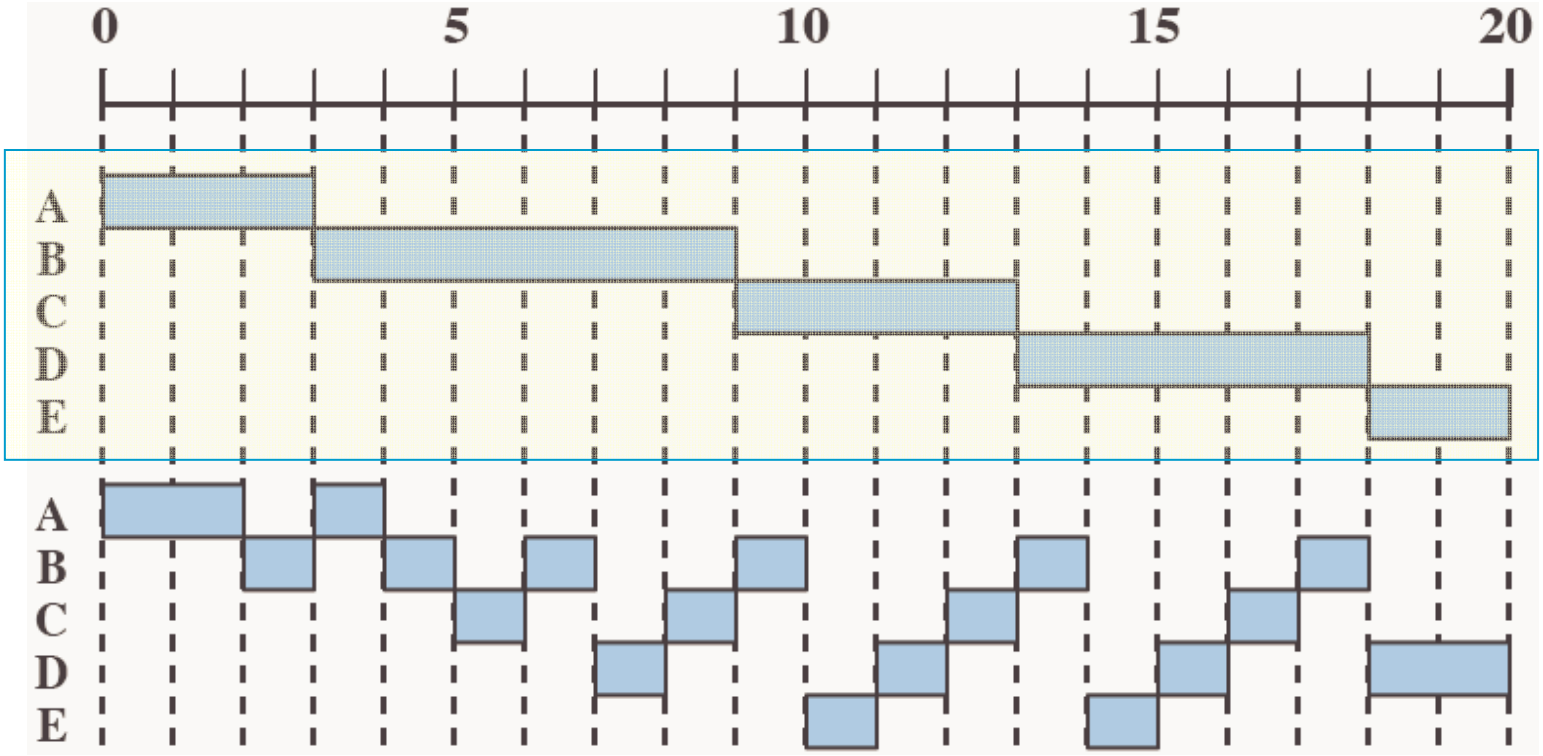
Geliş Sırasına Göre (FCFS)

- Her proses Hazır kuyruğuna girer
- Çalışmakta olan prosesin çalışması durunca *Hazır* kuyruğundaki en eski proses seçilir
- Uzun süre çalışacak prosesler için daha iyi
  - Kısa bir proses çalışmaya başlamadan önce çok bekleyebilir

Geliş Sırasına Göre (FCFS)

- G/Ç yoğun prosesler için uygun değil
  - G/Ç birimleri uygun olsa da işlemciyi kullanan prosesleri beklemek zorundalar
  - kaynakların etkin olmayan kullanımı

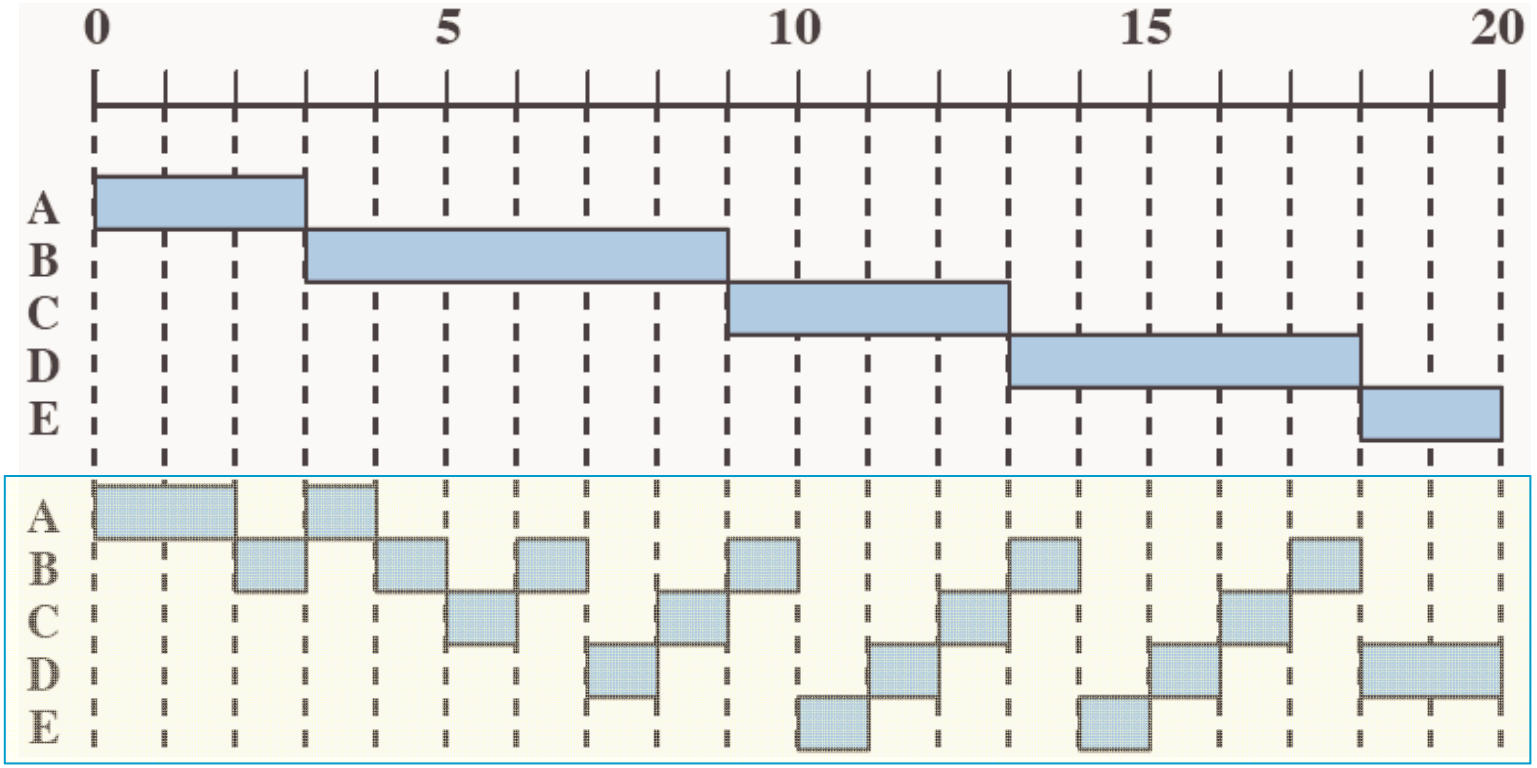
Geliş Sırasına Göre (FCFS)



- Her proses Hazır kuyruğuna girer
- Çalışmakta olan prosesin çalışması durunca *Hazır* kuyruğundaki en eski proses seçilir
- Kısa bir proses çalışmaya başlamadan önce çok bekleyebilir

Taramalı (Round-Robin)

- Saate bağlı olarak prosesleri işlemciden almak (preemptive)
- İşlemcinin her seferde kullanılacağı süre birimi belirlenir



q=1

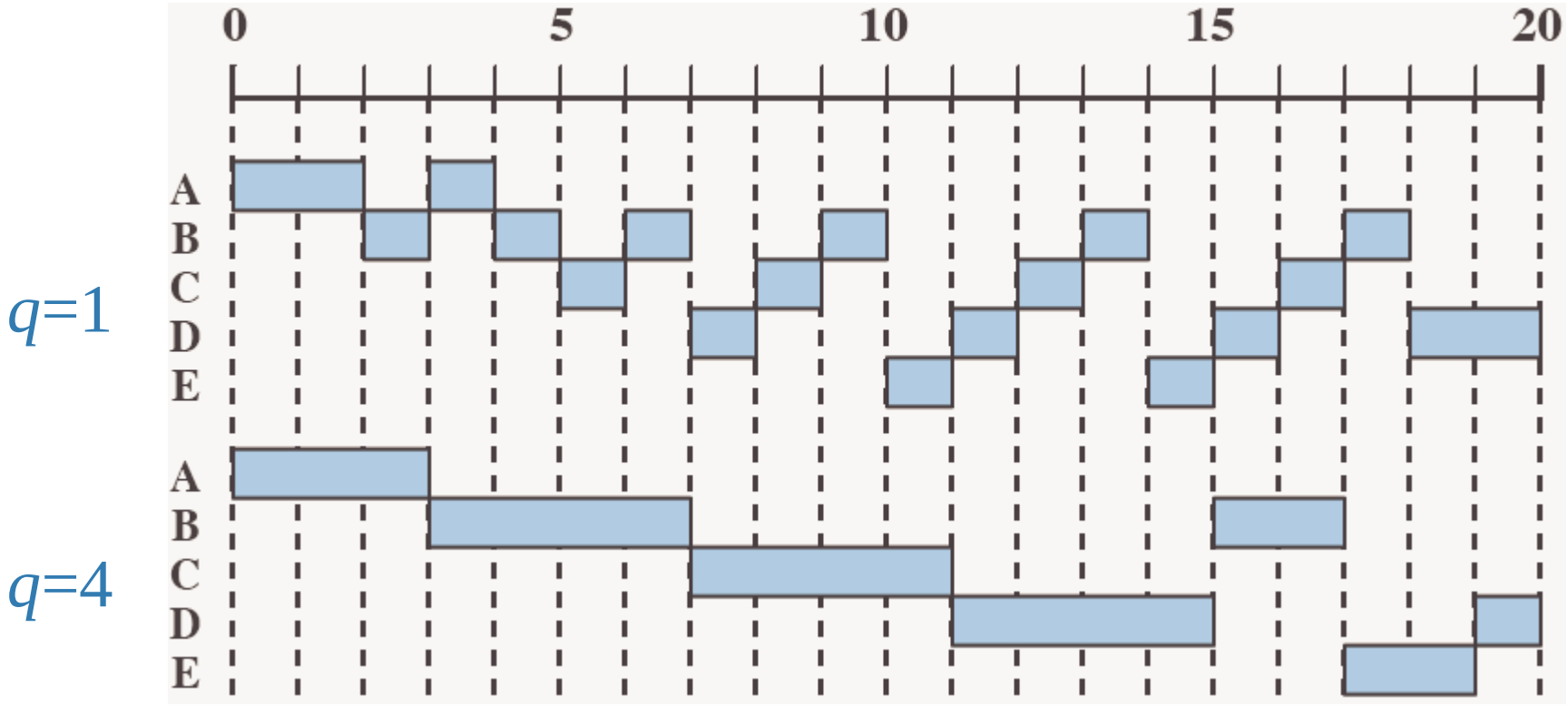
Taramalı (Round-Robin)

- Saat kesmeleri periyodik zaman aralıkları ile üretilir - *zaman dilimi*
- zaman dilimi uzunluğu belirlenmesi gerekir
- kısa zaman dilimleri uygun değil
  - kısa prosesler hızla tamamlanır ama
  - sisteme çok yük oluşur

Taramalı (Round-Robin)

- Zaman dilimi yaklaşık olarak ortalama bir etkileşim süresi kadar olmalı
- Zaman dilimi en uzun prosesten uzun olursa geliş sırasına göre olan yöntemle aynı
- Saat kesmesi geldiğinde koşmakta olan proses *Hazır* kuyruğuna konur ve sıradaki iş seçilir

Taramalı (Round-Robin)



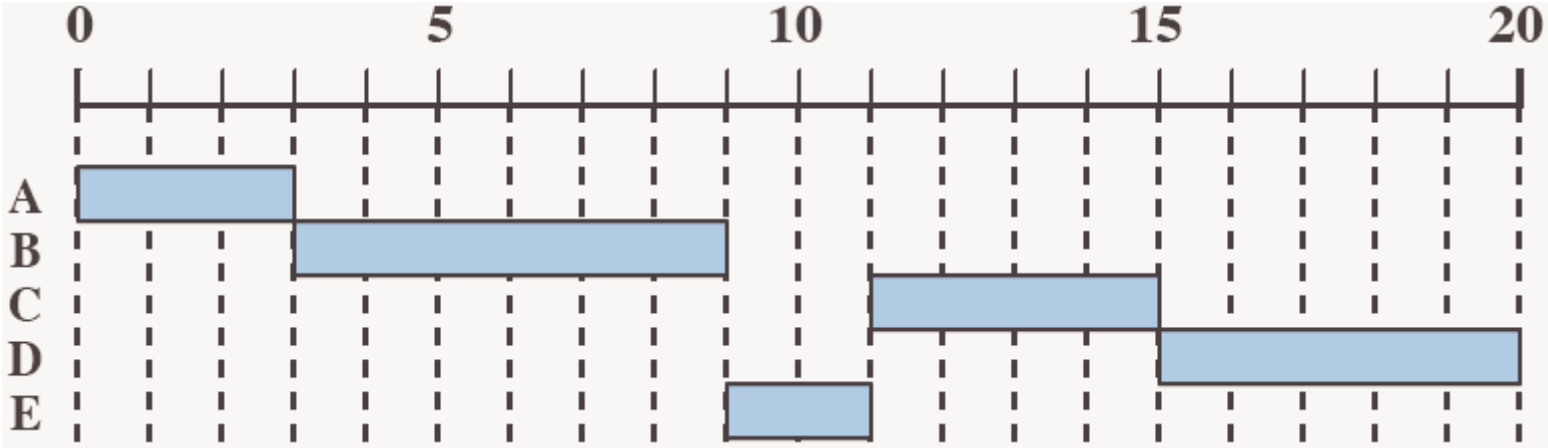
q=1

q=4

En Kısa İş Önce

- Prosesler kesilmeden çalışıyor
- Servis süresi en kısa olacağı bilinen/tahmin edilen proses seçilir
  - süre tahmin edilmesi veya bildirilmesi gerekir; tahmin edilenden çok uzun süren prosesler sonlandırılabilir
- Kısa prosesler önce çalışmış olur
- Uzun prosesler için *Açlık* durumu olası

En Kısa İş Önce

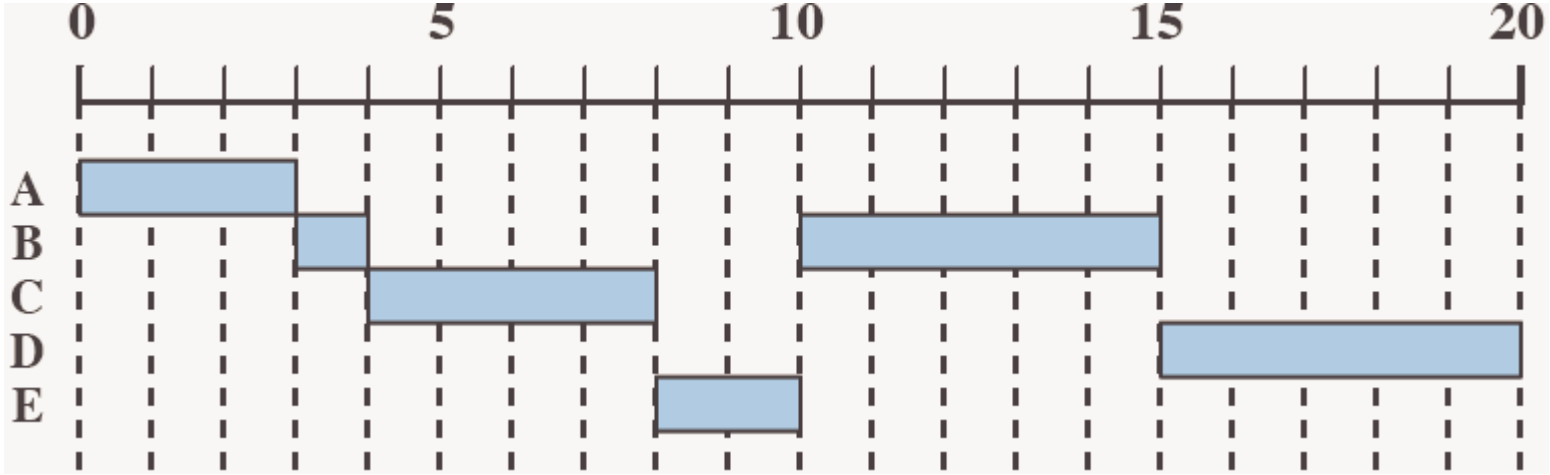


Proses	Varış Zamanı	Servis Süresi
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

En Az Süresi Kalan Önce

- En kısa iş önce yönteminin proseslerin kesilmesine izin veren hali
- İşleme zamanı önceden kestirilmeli
- Proseslerin kalan süreleri tutulmalı

En Az Süresi Kalan Önce



Geri Beslemeli (feedback)

- Proseslerin toplam çalışma süreleri ile ilgili bilgi gerektirmez
- Prosesin daha ne kadar çalışma süresi kaldığı bilinmiyor
- Daha uzun süredir koşan prosesleri cezalandır

Geri Beslemeli (feedback)

- İş sıralama
  - zaman dilimi ve
  - dinamik önceliklere

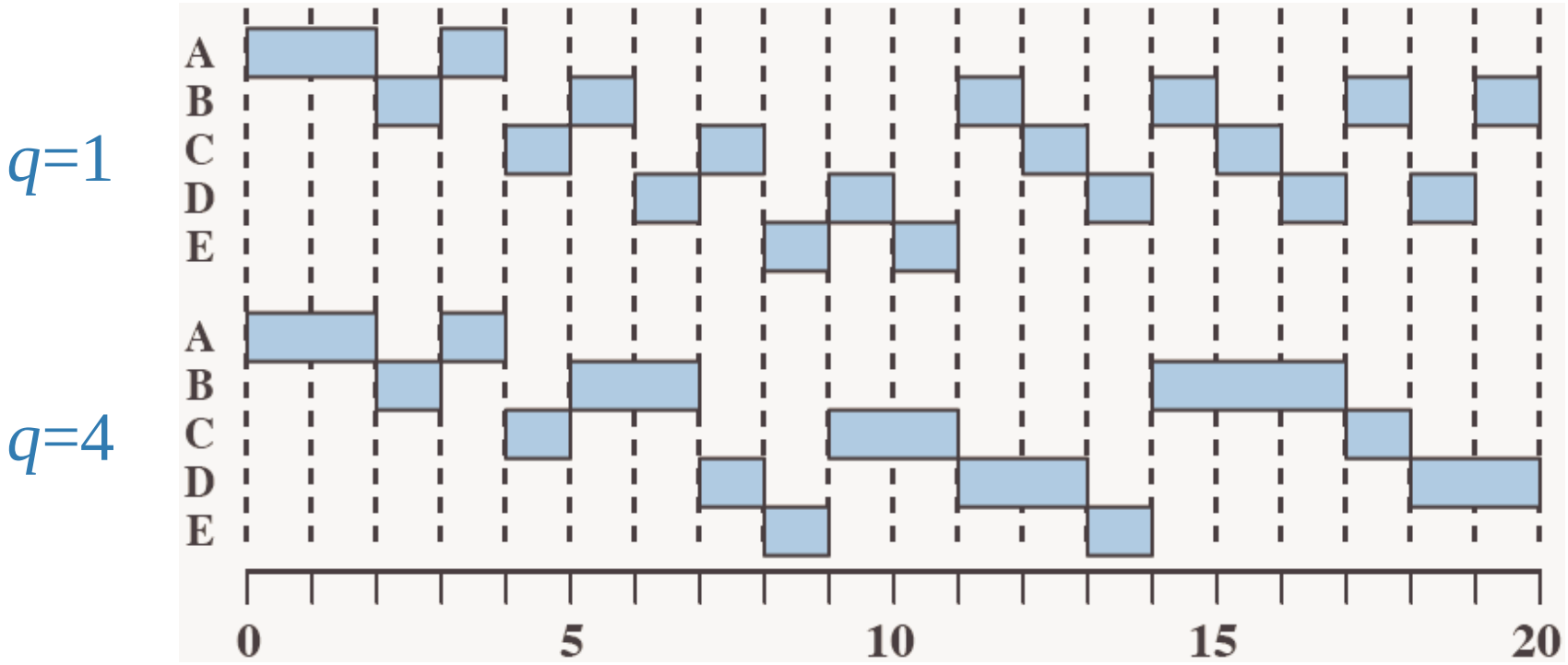
göre yapılır.

- Öncelik kuyrukları
  - kesilen proses bir düşük öncelikli kuyruğa geçer

Geri Beslemeli (feedback)

- Yeni ve kısa işler daha çabuk sonlanır
- Öncelik kuyrukları içinde geliş sırası göz önüne alınır
  - en düşük öncelikli kuyuruktan daha alta inmek mümkün olmadığından bu kuyrukta kalır (taramalı yöntem uygulanır)

Geri Beslemeli



Yöntemlerin Başarım Karşılaştırması

- İş sıralama yöntemi seçilirken yöntemlerin başarım karşılaştırmaları önem kazanır
- Kesin karşılaştırma mümkün değil

Yöntemlerin Başarım Karşılaştırması

- Yöntemlerin bağıl başarımları bazı etkenlere bağlı
  - Proseslerin servis sürelerinin olasılık dağılımı
  - Bağlam anahtarlamanın etkinliği
  - G/Ç isteklerini ve G/Ç alt yapısının özellikleri

Yöntemlerin Başarım Karşılaştırması

- Kuyruk teorisi ile analiz
- Modelleme ve simülasyon ile

İş Sıralama Yöntemleri – Toplu Bir Bakış

	Selection Function	Decision Mode	Throughput	Response Time	Overhead	Effect on Processes	Starvation
FCFS	max[w]	Nonpreemptive	Not emphasized	May be high, especially if there is a large variance in process execution times	Minimum	Penalizes short processes; penalizes I/O bound processes	No
Round Robin	constant	Preemptive (at time quantum)	May be low if quantum is too small	Provides good response time for short processes	Minimum	Fair treatment	No
SPN	min[s]	Nonpreemptive	High	Provides good response time for short processes	Can be high	Penalizes long processes	Possible
SRT	min[$s-e$]	Preemptive (at arrival)	High	Provides good response time	Can be high	Penalizes long processes	Possible
HRRN	$\max\left(\frac{w+s}{s}\right)$	Nonpreemptive	High	Provides good response time	Can be high	Good balance	No
Feedback	(see text)	Preemptive (at time quantum)	Not emphasized	Not emphasized	Can be high	May favor I/O bound processes	Possible

Geleneksel UNIX İş Sıralama Yaklaşımı

- kullanılan iş sıralama yaklaşımı
  - çok seviyeli
  - geri beslemeli
  - her öncelik seviyesi içinde taramalı
- 1 saniyelik dilimler
  - 1 saniyede sonlanmayan proses kesilir (preemptive)

Geleneksel UNIX İş Sıralama Yaklaşımı

- öncelikler
  - prosesin türüne ve
  - çalışma geçmişine bağlı.
- İş sıralama algoritmasının gerçeklenmesinde bazı denklemler kullanılır

Geleneksel UNIX İş Sıralama Yaklaşımı

$$CPU_j(i) = \left\lfloor \frac{CPU_j(i-1)}{2} \right\rfloor$$

$$P_j(i) = Taban_j + \left\lfloor \frac{CPU_j(i)}{2} \right\rfloor + nice_j$$

$CPU_j(i)$: i. zaman diliminde j prosesinin işlemci kullanım miktarı
$P_j(i)$: i. zaman dilimi başında j prosesinin öncelik değeri
(düşük değerler yüksek öncelik anlamına gelir)
$Taban_j$: j prosesinin taban öncelik değeri
$nice_j$: kullanıcı tarafından belirlenen öncelik ayarlama faktörü

Geleneksel UNIX İş Sıralama Yaklaşımı

- Her prosesin önceliği 1 saniyede bir hesaplanır
  - iş sıralama kararı verilir
- Taban öncelik değerinin amacı, prosesleri türlerine göre belirli gruplara ayırmak

Geleneksel UNIX İş Sıralama Yaklaşımı

- Farklı taban öncelikler ile başlayan proses grupları:
  - swapper
  - blok tipi g/ç aygıt kontrolü
  - dosya yönetimi
  - karakter tipi g/ç aygıt kontrolü
  - kullanıcı prosesleri

Örnek: A, B ve C prosesleri aynı anda, 60 taban önceliğine sahip olarak yaratılmış olsunlar.

- ***nice*** değeri göz önüne alınmayacak.
- Saat sistemi saniyede 60 kere kesiyor ve bir sayaç değerini arttırıyor.
- Varsayımlar:
  - Sistemde başka koşmaya hazır proses yok
  - Proseslerin herhangi bir nedenle bloke olmuyorlar.

Proses A, B ve C'nin çalışma sürelerini ve sıralarını gösteren zamanlama tablosunu çizin.

7 BELLEK YÖNETİMİ

Bellek Yönetimi

- Birden fazla prosese yer verilebilecek şekilde belleğin alt birimlere ayrılması
- Belleğin prosesler arasında atanması etkin olmalı:
  - en fazla sayıda proses

Bellek Yönetiminin Gerektirdikleri

- Yeniden yerleştirme (Relocation)
  - programcı çalışan programının bellekte nereye yerleşeceğini bilmez
  - koşan program ikincil belleğe atılıp, ana bellekte farklı bir yere tekrar yüklenebilir
  - Bellek referans adresleri fiziksel adres değerlerine dönüştürülmeli

Bellek Yönetiminin Gerektirdikleri

- Koruma
  - İzni olmadan bir proses bir başka prosesin bellek alanlarına erişemez
  - Programın yeri değişebileceğinden kontrol için programdaki gerçek adresler kullanılamaz
  - Çalışma anında kontrol edilmeli

Bellek Yönetiminin Gerektirdikleri

- Paylaşma
  - Birden fazla prosesin aynı bellek bölgesine erişmesi
    - program kodu
    - ortak veri alanı

Bellek Yönetimi Teknikleri

- Bölmeleme (Partitioning)
  - Sabit
  - Dinamik
- Basit sayfalama (Paging)
- Basit segmanlama (Segmentation)
- Sayfalamalı görüntü bellek (Virtual Memory)
- Segmanlamalı görüntü bellek

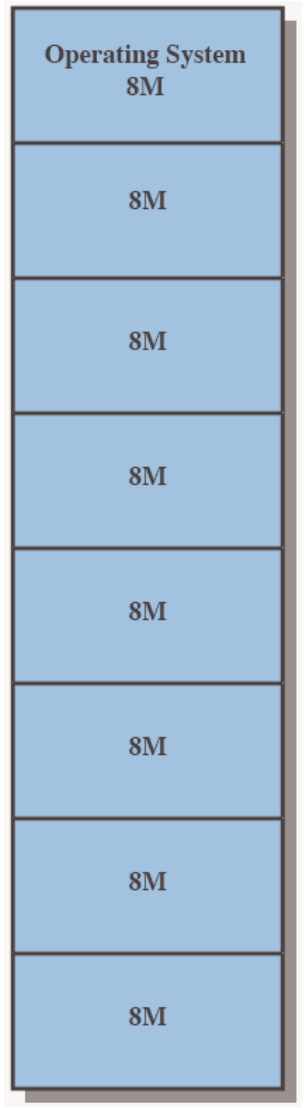
Sabit Bölmeleme

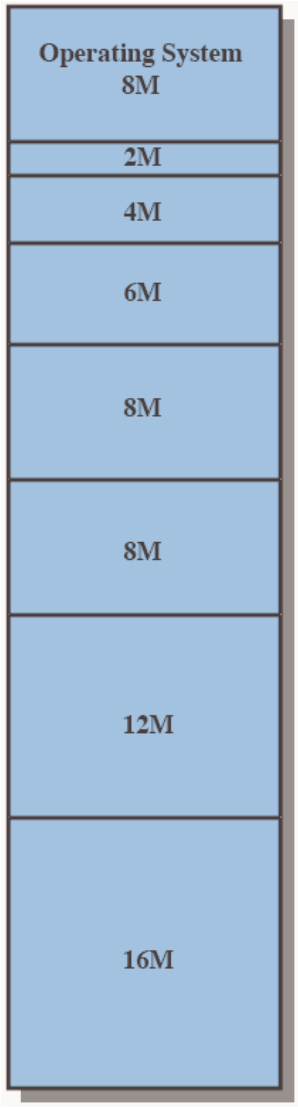
- Bölme boyları eşit
  - boş bölmeye, boyu bölme boyundan küçük ya da eşit prosesler yüklenebilir
  - tüm bölmeler doluysa proseslerden biri bellekten atılır
  - program bölmeye sığmayabilir ⇒ programcı program parçalarını birbirinin üzerini örtecek şekilde (overlay) yazar

Sabit Bölmeleme

- Bellek kullanımı etkin değil:
  - her program ne kadar boyu küçük de olsa tam bir bölmeyi elinde tutar $\Rightarrow$ iç parçalanma (internal fragmentation)
  - eşit boyda olmayan bölmeler kullanılması sorunu bir derece çözer
- Maksimum aktif proses sayısı sınırlı
- İşletim sistemi tarafından gerçeklenmesi kolay
- Getirdiği ek yük az.

Sabit Bölmeleme





Yerleştirme Algoritmaları (Sabit Bölmeleme)

- Bölme boyları eşit
  - prosesin hangi bölmeye yerleştirileceği fark etmez
- Bölme boyları eşit değil
  - her prosesi sığacağı en küçük bölmeye
  - her bölme için kuyruk
  - bölme içi boş kalan yer miktarını en aza indirmek

    (IBM OS/MFT: multiprogramming with a fixed number of tasks)

Dinamik Bölmeleme

- Bölme sayısı ve bölme boyları sabit değil
- Proseslere sadece gerektiği kadar bellek atanır
- Kullanılmayan boş yerler yine de oluşur $\Rightarrow$ dış parçalanma
- Tüm boş alanın bir blok halinde olması için sıkıştırma kullanılır

  (IBM OS/MVT: multiprogramming with a variable number of tasks)

Yerleştirme Algoritmaları (Dinamik Bölmeleme)

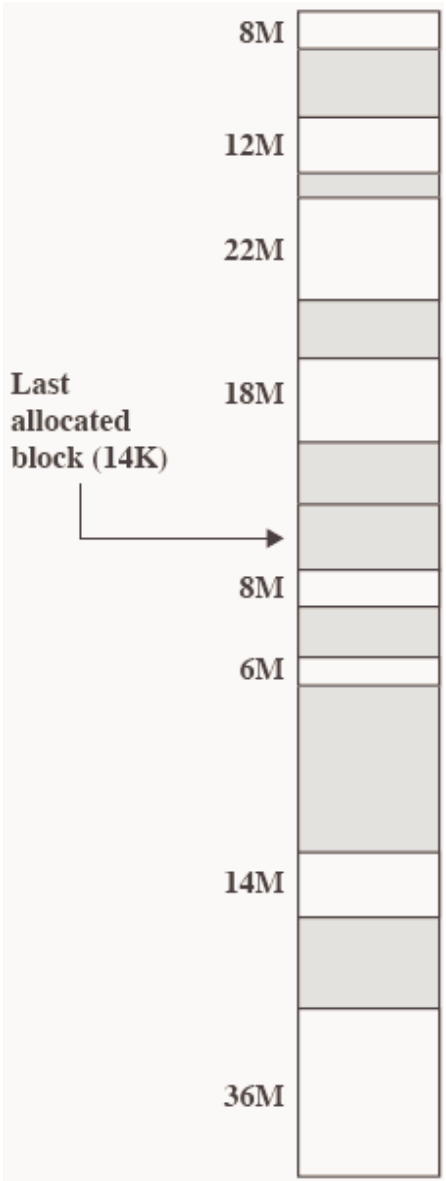
- Hangi boş bloğun hangi proses atanacağına işletim sistemi karar verir
- En-İyi-Sığan Algoritması (Best-Fit)
  - Boyu istenene en yakın olan boşluk seçilir
  - Olası en küçük bölme bulunduğundan artan boş alan az
    $\Rightarrow$ sıkıştırmanın daha sık yapılanması gerekir

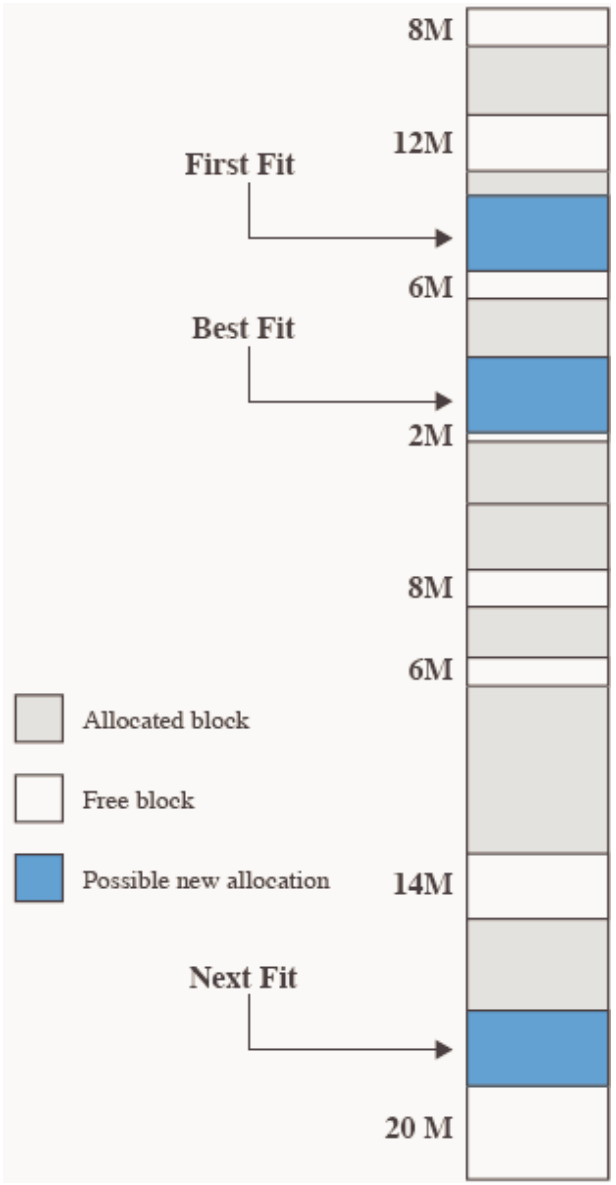
Yerleştirme Algoritmaları (Dinamik Bölmeleme

- İlk-Sığan Algoritması (First-fit)
  - En hızlı
  - Prosesler baş bölgelere yığılır $\Rightarrow$ boş yer ararken üst üste taranır

Yerleştirme Algoritmaları (Dinamik Bölmeleme

- Bir-Sonraki-Sığan Algoritması (Next-fit)
  - Son yerleştirilen yerden itibaren ilk sığan yeri bulur
  - Genellikle atamalar belleğin son kısımlarında yer alan büyük boşluklardan olur
    - En büyük boşluklar küçük parçalara bölünmüş olur
    - Sıkıştırma gerekir

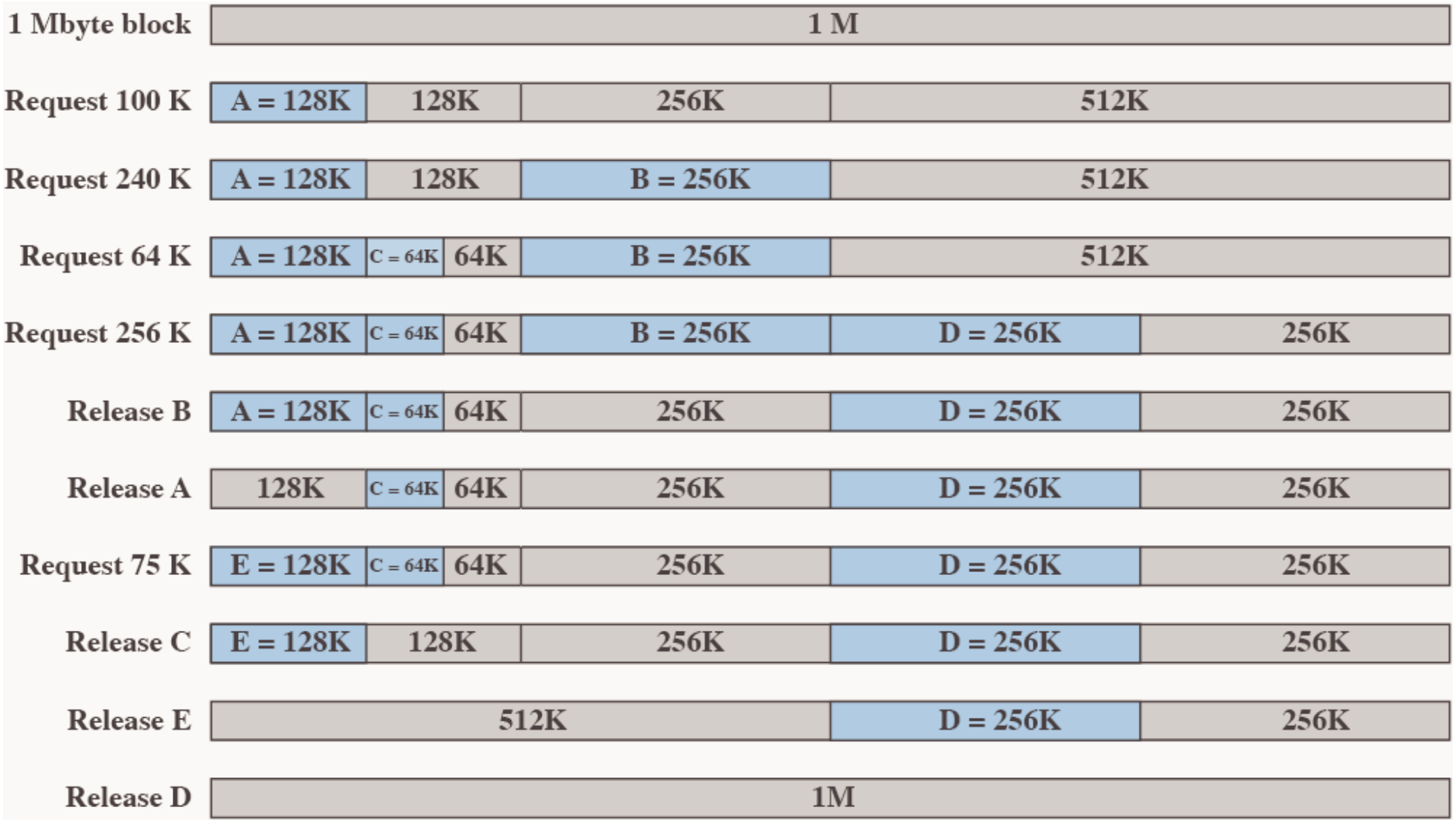

8M
12M
22M
18M
Last allocated block (14K)
8M
6M
14M
36M


8M
12M
First Fit
6M
Best Fit
2M
8M
6M
Allocated block
Free block
Possible new allocation
14M
Next Fit
20 M

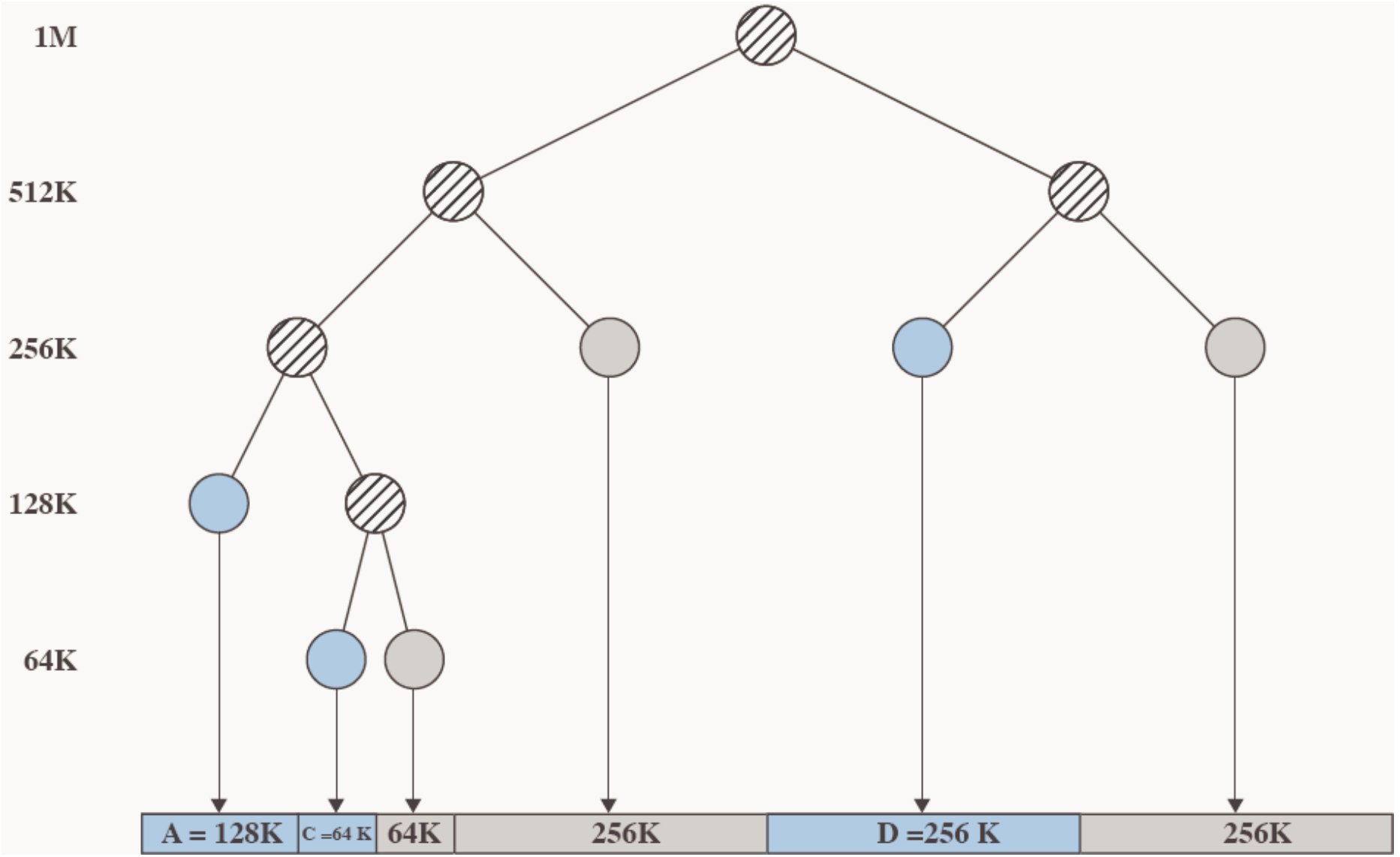
“Buddy” Yöntemi

- Tüm boş alan 2^U boyutunda tek bir alan olarak ele alınır
- s boyutundaki bir istek eğer $2^{U-1} < s <= 2^U$ ise tüm blok atanır
  - Aksi halde blok 2^{U-1} boyutunda iki eş bloğa bölünür (buddy)
  - s’den büyük veya eşit en küçük birim blok oluşturulana kadar işlem devam eder

Buddy Sistem Örneği



Buddy Yönteminin Ağaç ile Temsili



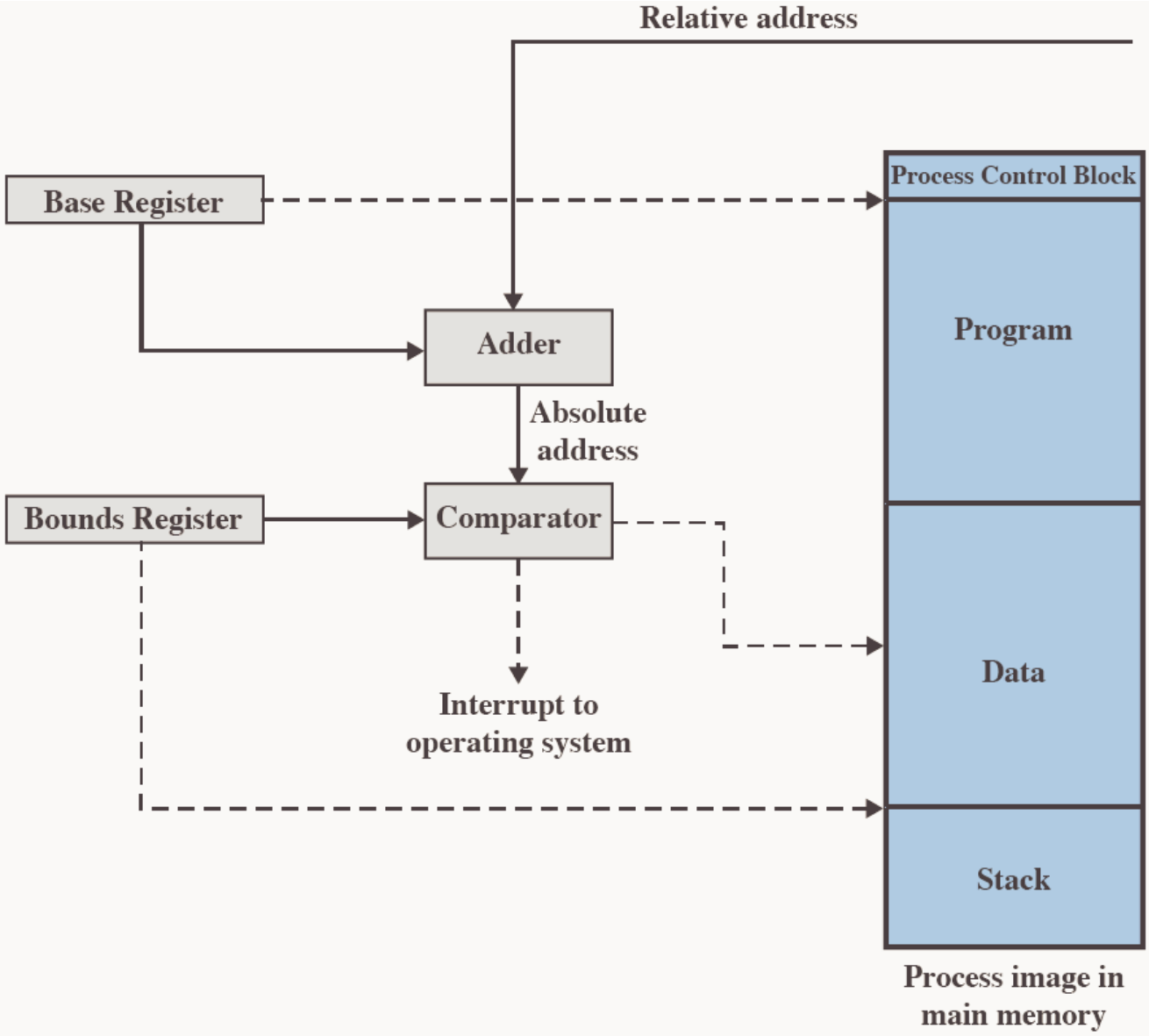
Yeniden Yerleştirme

- Proses belleğe yüklendiği zaman ancak mutlak bellek adresleri belirlenebilir
- Proses çalışması boyunca değişik bölmelere yerleşebilir (swap) $\Rightarrow$ Mutlak adresler değişebilir
- Sıkıştırma nedeni ile de proses farklı bölmelerde yerleşebilir $\Rightarrow$ farklı mutlak bellek adresleri

Adresler

- Mantıksal
  - belleğe erişimde kullanılan adres gerçek fiziksel adreslerden bağımsız
  - adres dönüşümü gerekir
- Bağıl
  - adres bilinen bir noktaya göre bağıl verilir
  - genellikle bu nokta prosesin başıdır
- Fiziksel
  - ana bellekteki gerçek konum adresi

Yeniden yerleştirme için gerekli donanım desteği



Saklayıcılar

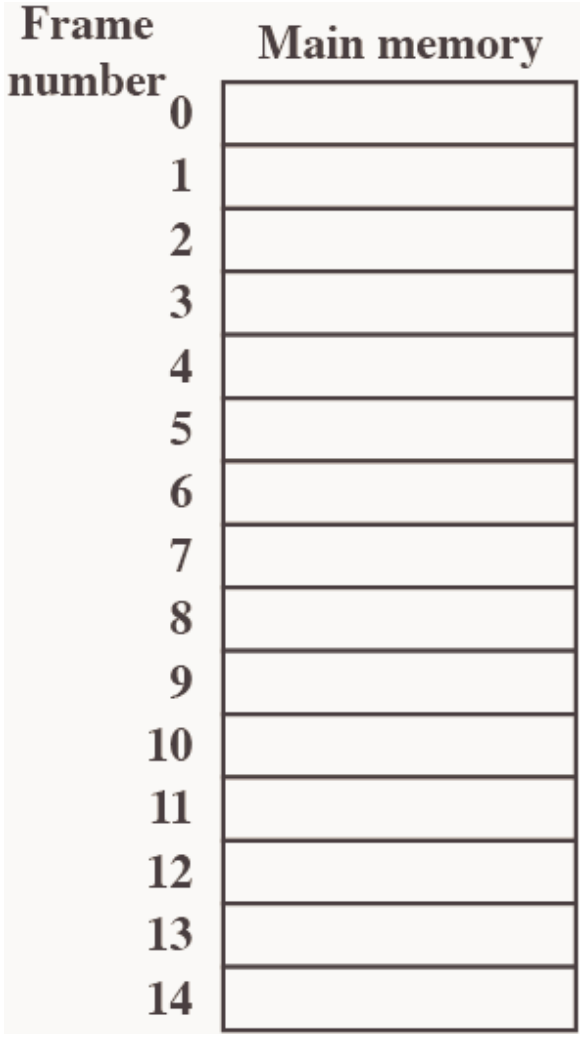
- Taban saklayıcısı
  - prosesin başlangıç adresi
- Sınır saklayıcısı
  - prosesin son adresi
- Bu değerler saklayıcılara proses belleğe yüklendiğinde yazılır

Sayfalama

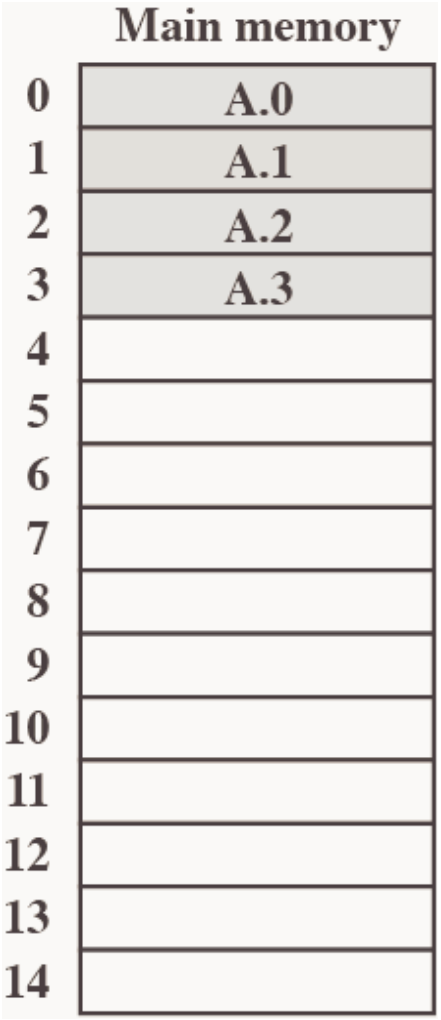
- Belleği küçük, eşit boylu parçalara böl. Prosesleri de aynı boyda parçalara ayır.
- Eşit boylu proses parçaları: *sayfa*
- Eşit boylu bellek parçaları: *çerçeve*
- İşletim sistemi her proses için *sayfa tablosu* tutar
  - prosesin her sayfasının hangi çerçevede olduğu
  - bellek adresi = sayfa numarası ve sayfa içi ofset adresi
- İşletim sistemi hangi çerçevelerin boş olduğunu tutar

Sayfalama

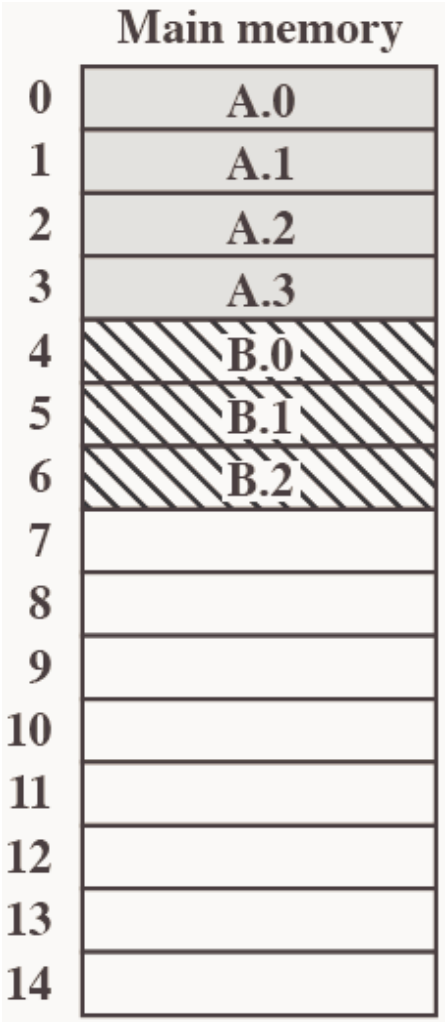
- Sabit bölmeleme benzeri
- Bölmeler küçük
- Bellekte bir prosese ilişkin birden fazla sayfa olabilir
- Sayfa ve çerçeve boylarının ikinin kuvvetleri şeklinde seçilmesi gerekli dönüşüm hesaplamalarını basitleştirir
- Mantıksal adres, sayfa numarası ve sayfa içi kayıklık değerinden oluşur



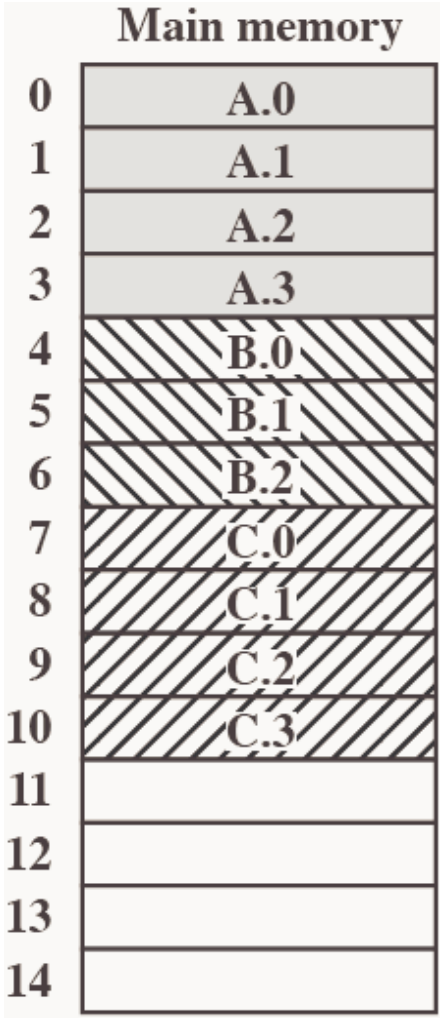
15 available frames



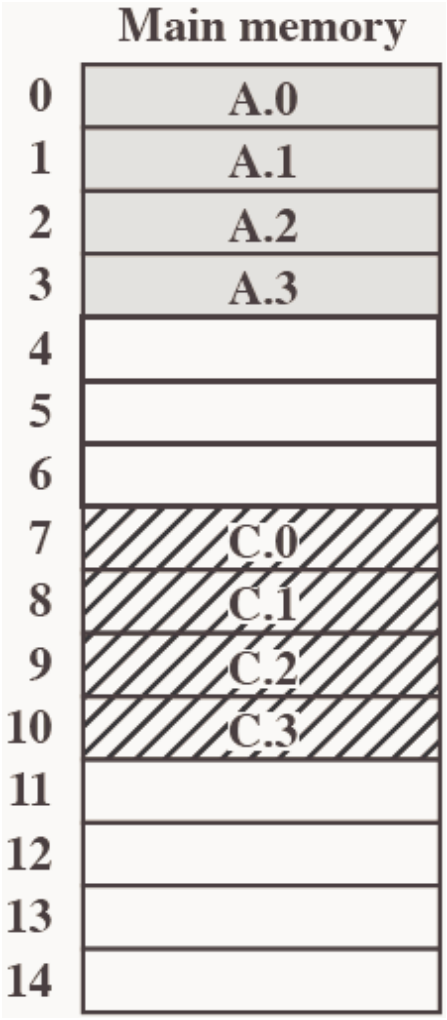
Load process A



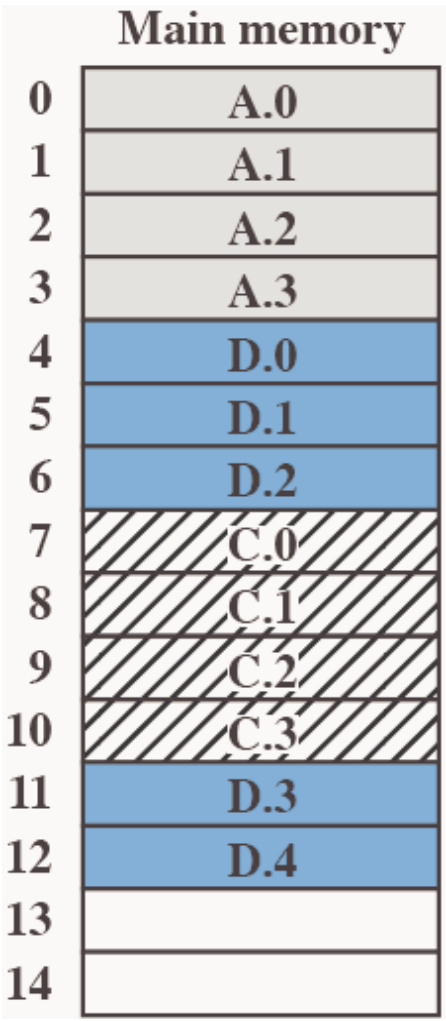
Load process A



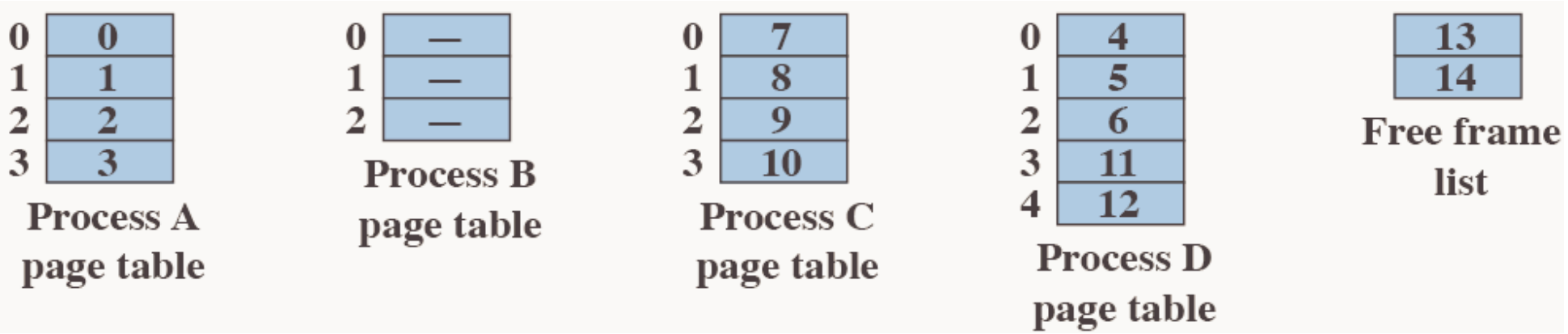
Load process C

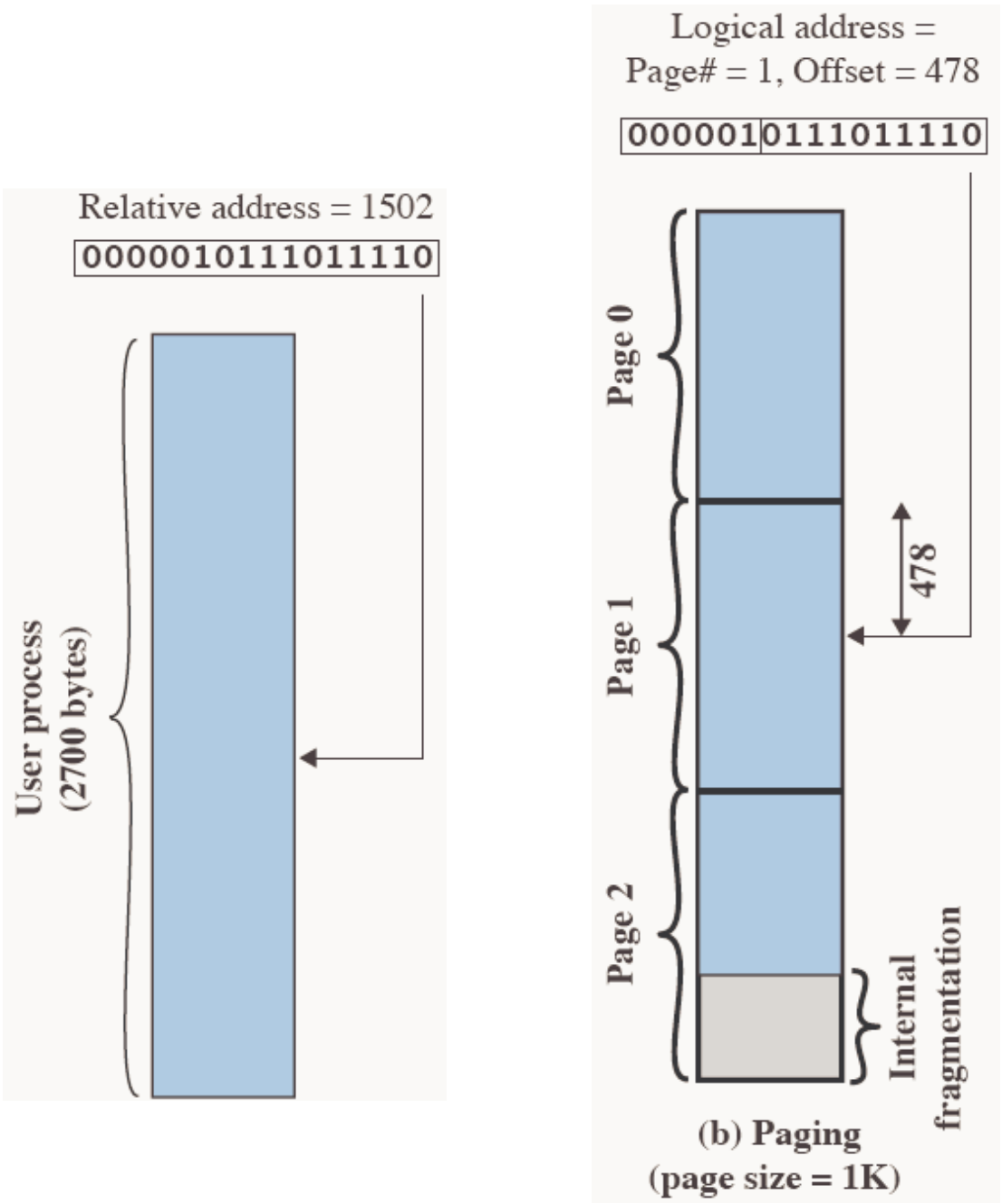


Swap out B

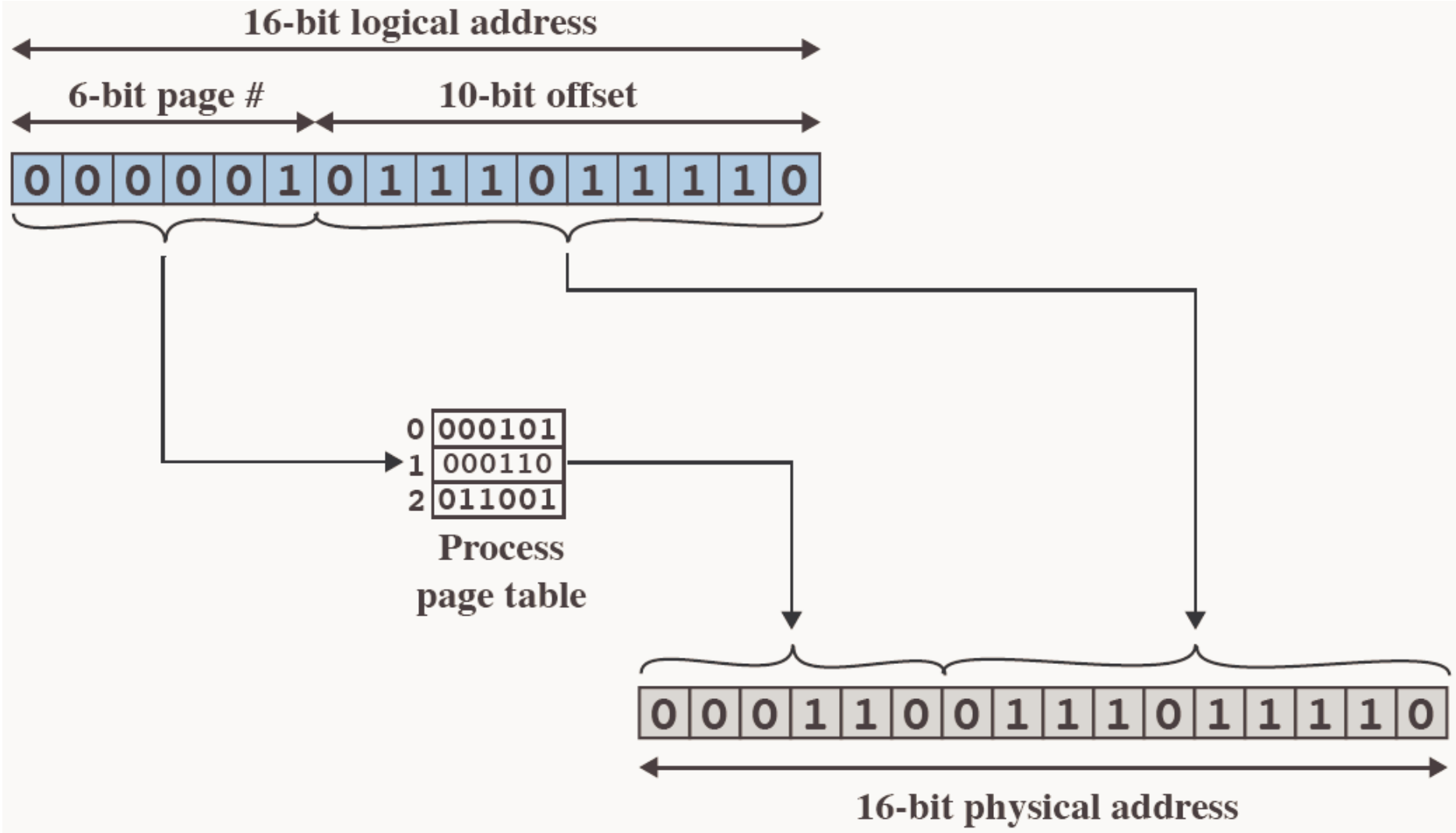


Load process D


0 0
1 1
2 2
3 3
Process A page table
0 —
1 —
2 —
Process B page table
0 7
1 8
2 9
3 10
Process C page table
0 4
1 5
2 6
3 11
4 12
Process D page table
13
14
Free frame list


Relative address = 1502
0000010111011110
User process
(2700 bytes)
Logical address =
Page# = 1, Offset = 478
0000010111011110
Page 0
Page 1
Page 2
478
Internal
fragmentation
(b) Paging
(page size = 1K)

Mantıksal-Fiziksel Adres Dönüşümü

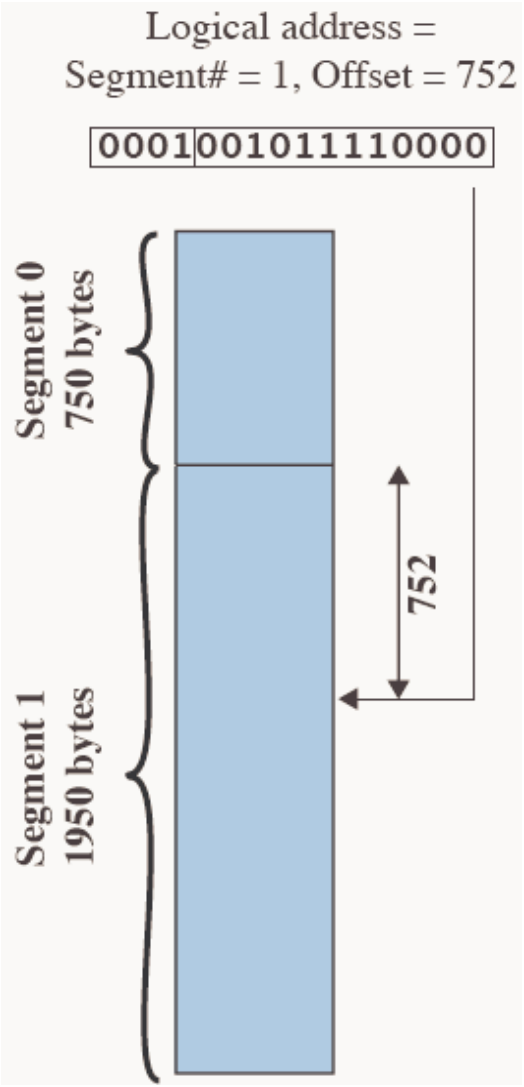


Segmanlama

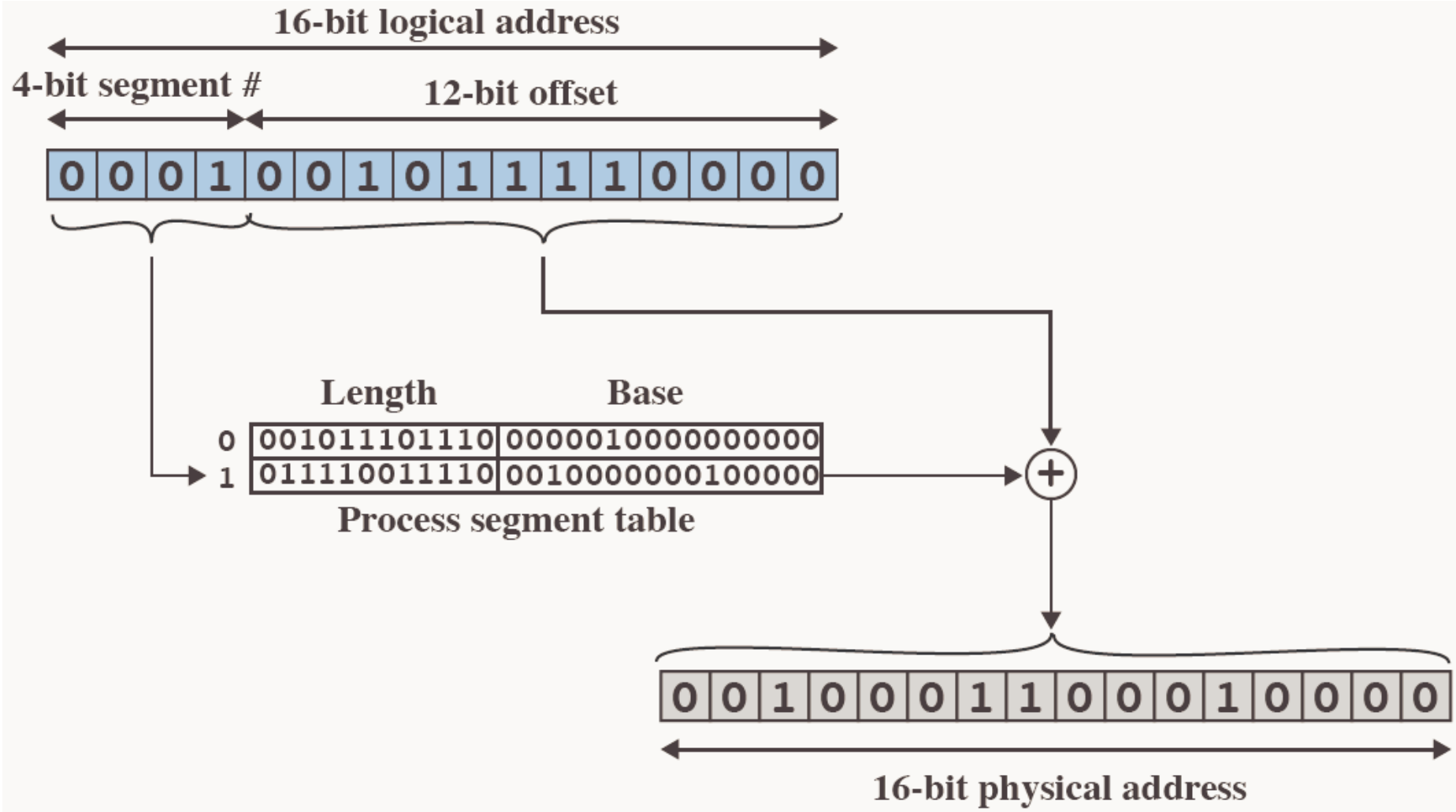
- Program segmanlara ayrılır. Tüm programların tüm segmanları aynı boyda olmak zorunda değil.
- Segman boyunun üst sınırı var
- Mantıksal adresler iki bölümden oluşur -segman numarası ve segman içi ofset
  - segman tablosundan segmanın adresi ve boyu alınır
- Segman boyları eşit olmadığından dinamik bölmelemeye benzer
- Bir program birden fazla segmandan oluşabilir

Segmanlama

- Bir programa ilişkin segmanlar bellekte ardışıl yerleşmek zorunda değil
- Segman tabloları var (yükleme adresi ve segman boyu bilgileri)
- Segmanlar programcıya şeffaf değil
- Kullanıcı / derleyici program text ve veri alanlarını farklı segmanlara atar.
- Maksimum segman boyu bilinmesi gerekir



Mantıksal-Fiziksel Adres Dönüşümü



7.2 GÖRÜNTÜ BELLEK

Program Koşması

- Bellek erişimleri dinamik olarak çalışma anında fiziksel adreslere dönüştürülür
  - Proses çalışması boyunca belleğe alınıp, bellekten atılabilir ve her seferinde farklı bir bölgeye yerleştirilebilir.
- Prosesin parçalarının bellekte birbirini izleyen bölgelerde olması gerekli değil
- Çalışma anında prosesin tüm parçalarının birden bellekte olması gerekmez

Program Koşması

- İşletim sistemi başlangıçta belleğe prosesin bir kısmını yükler
- Yerleşik küme (resident set) - prosesin bellekte bulunan kısmı
- İhtiyaç duyulan bir bölge bellekte yoksa kesme oluşur
- İşletim sistemi prosesi bloke eder

Program Koşması

- İstenen mantıksal adresi içeren parçası belleğe yüklenir
  - İşletim sistemi tarafından disk G/Ç isteği oluşturulur
  - Disk G/Ç işlemi yürütülürken bir başka proses çalışır
  - Disk G/Ç işlemi tamamlanınca kesme oluşur. İşletim sistemi bekleyen prosesi *Hazır* durumuna getirir.

Prosesi Parçalara Ayırmanın Avantajları

- Ana bellekte daha fazla proses bulunabilir
  - Prosesin sadece gerekli parçaları yüklenebilir
  - Bellekte çok proses olduğundan en az birinin *Hazır* durumunda olması olasılığı yüksek
- Bir proses tüm ana bellekten daha büyük olabilir

Bellek

- Gerçek bellek
  - Ana bellek
- Görüntü bellek
  - Disk üzerinde oluşturulan ikincil bellek
  - Çoklu programlamayı daha etkin kılar.
  - Ana bellek boyu kısıtlarından programcıyı kurtarır.

Thrashing

- Bellekten atılan bir parçaya hemen ihtiyaç duyulması durumu
- İşlemci zamanı proses parçalarını ana bellek ve ikincil bellek arasında taşımakla geçer.
- Bu soruna karşılık, işletim sistemi, prosesin geçmişine bakarak hangi parçalara ihtiyaç duyacağını veya duymayacağını tahmin eder

Yerellik Prensibi

- Proses içi program kodu ve veri erişimleri birbirine yakın bölgelerde kalma eğilimindedir
- Kısa bir süre içinde prosesin sadece küçük bir alt kümesi gerekecektir
- Hangi parçaların gerekeceği konusunda tahminde bulunmak mümkün
- Etkin bir çalışma sağlamak mümkün

Görüntü Bellek İçin Gerekli Destek

- Donanım sayfalamaya ve segmanlı yapıya destek vermeli
- İşletim sistemi ana bellek ile ikincil bellek arasında sayfa ve/veya segman aktarımını etkin bir şekilde düzenleyebilmeli.

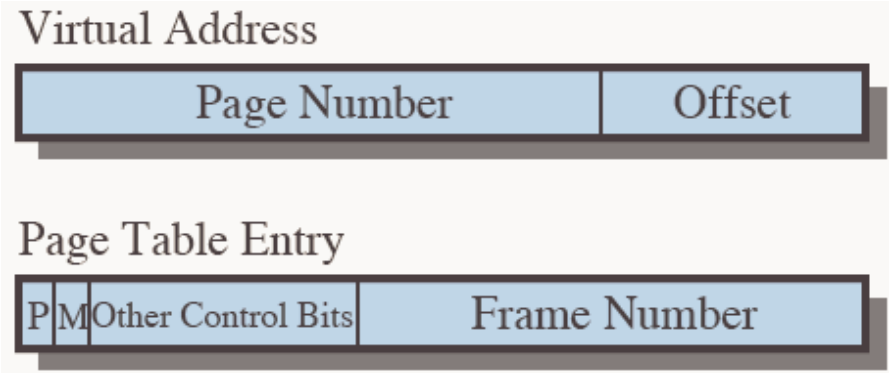
Sayfalama

- Her prosesin kendi sayfa tablosu var
- Tablonun her satırında sayfanın ana bellekte yer aldığı çerçeve numarası bulunur
- Sayfanın ana bellekte olup olmadığını gösteren de bir bit gerekli.

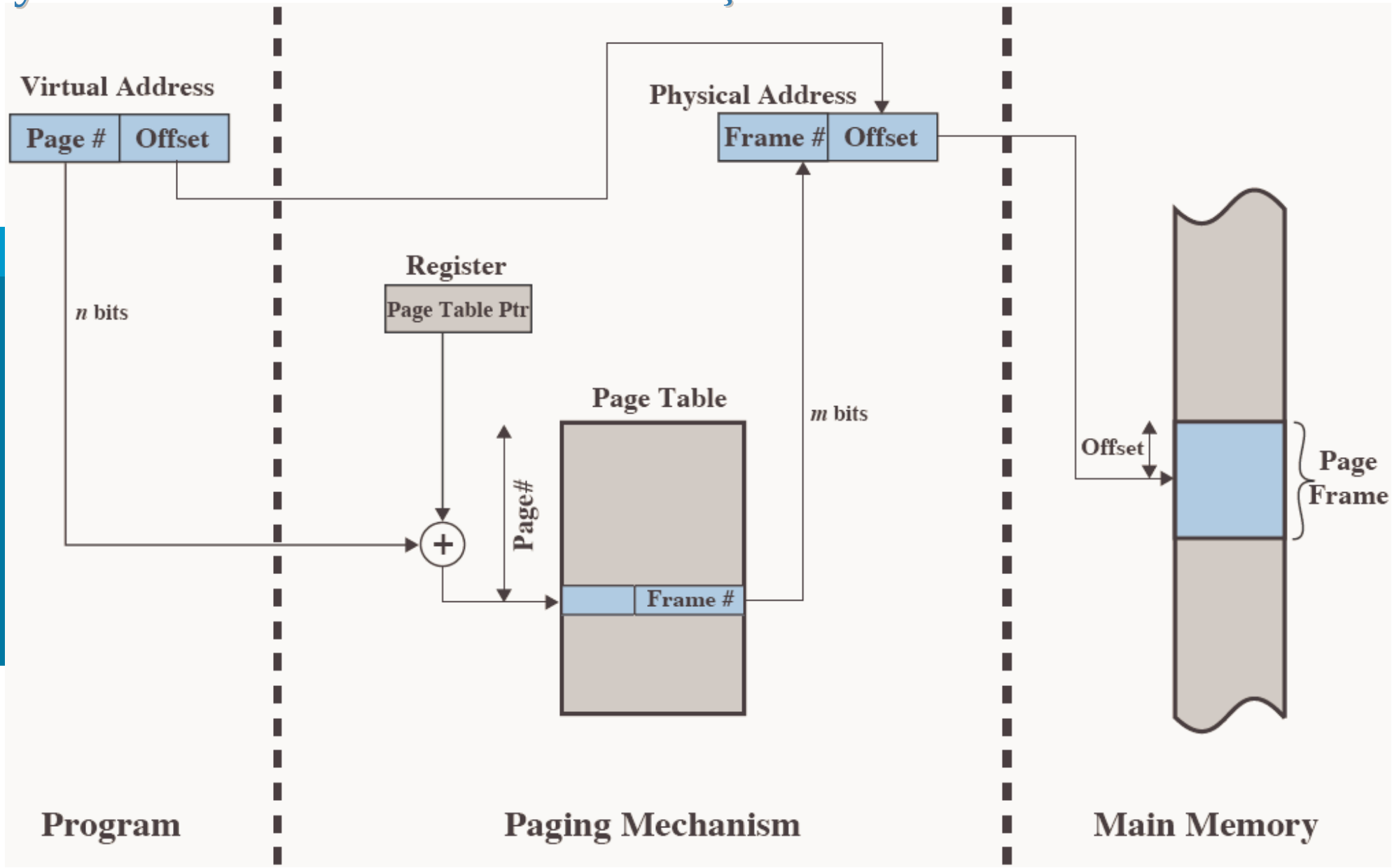
Sayfa Tablosundaki Değişti Biti

- Sayfanın belleğe yüklendikten sonra değişip değişmediğini gösterir
- Değişiklik yoksa ana bellekten ikincil belleğe alınırken yeniden yazmaya gerek yok

Sayfa Tablosu Kayıtları



Sayfalamalı bir sistemde adres dönüşümü



Sayfa Tabloları

- Sayfa tablosunun tamamı çok yer gerektirebilir.
- Sayfa tabloları da ikincil bellekte saklanır
- Koşan prosesin sayfa tablolarının bir kısmı da ana belleğe alınır.

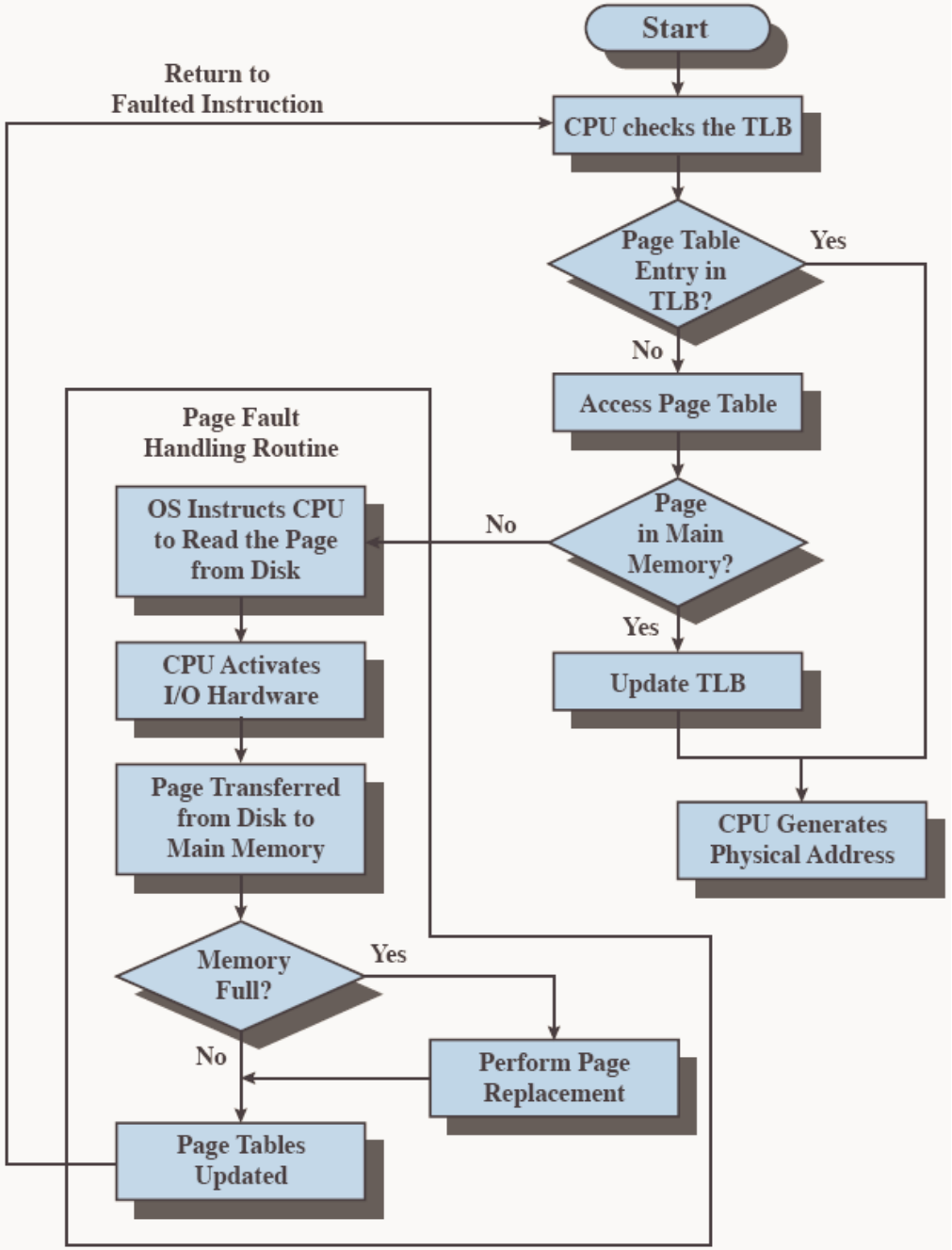
Translation Lookaside Buffer

- Her görüntü bellek erişiminde iki fiziksel bellek erişimi olabilir:
  - sayfa tablosunu getirmek için
  - veriyi getirmek için
- Bu problemin çözümünde sayfa tablosu kayıtlarını tutmak için hızlı bir cep bellek kullanılır:
  - TLB - Translation Lookaside Buffer

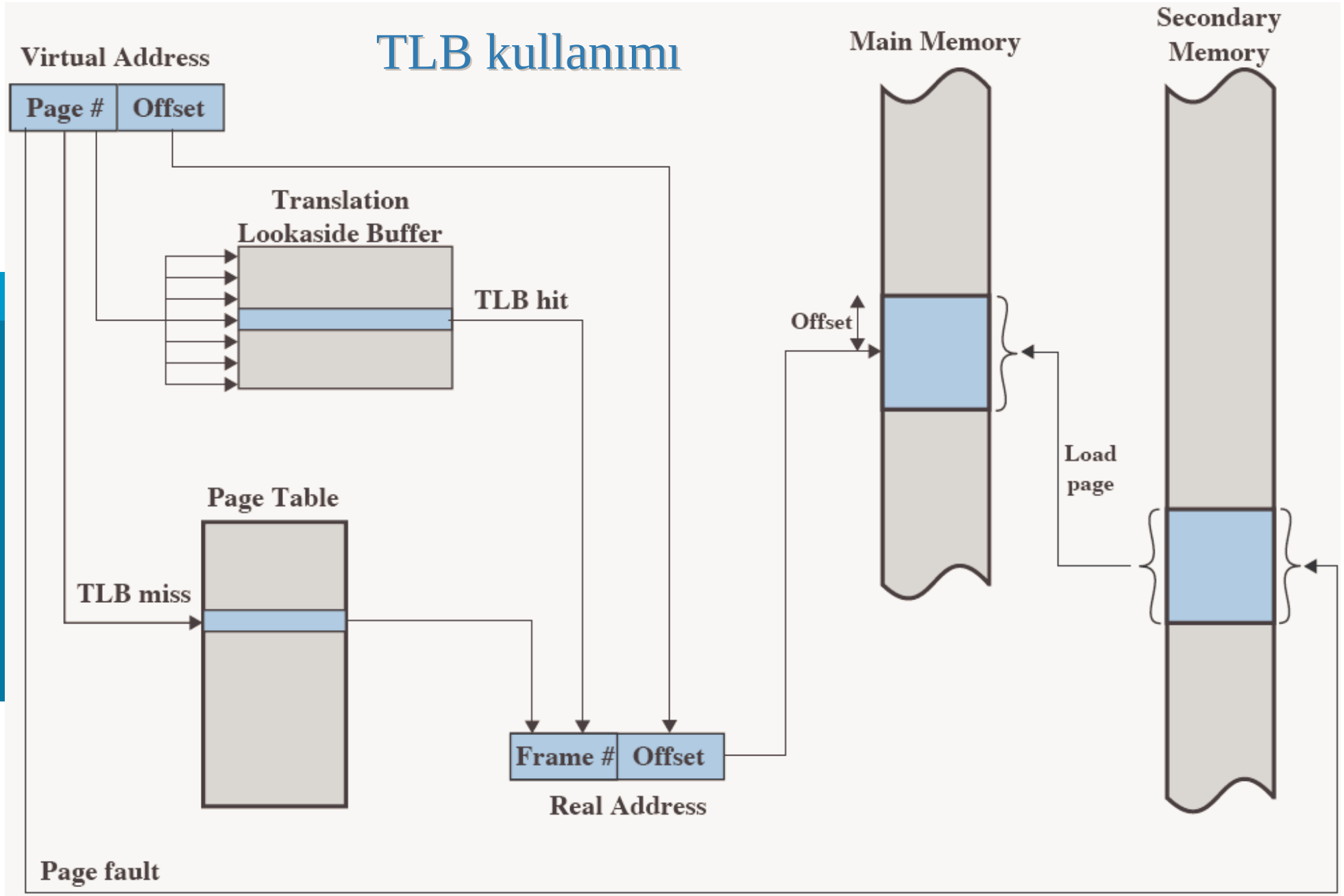
Translation Lookaside Buffer

- En yakın zamanda kullanılmış olan sayfa tablosu kayıtlarını tutar
- Ana bellek için kullanılan cep bellek yapısına benzer bir işlev görür

TLB kullanım akışı



TLB kullanımı



Sayfa Boyu

- Sayfa boyu küçük olursa iç parçalanma daha az
- Küçük sayfa boyları olursa proses başına gereken sayfa sayısı artar.
- Proses başına fazla sayfa olması sonucunda sayfa tablosu boyları büyür.
- Sayfa tablosu boyunun büyük olması sonucu tablonun ikincil bellekte tutulan kısmı daha büyük
- İkincil belleklerin fiziksel özellikleri nedeniyle daha büyük bloklar halinde veri aktarımı daha etkin $\Rightarrow$ sayfa boyunun büyük olması iyi

Sayfa Boyu

- Sayfa boyu küçük olunca bellekteki sayfa sayısı artar
- Zaman içinde proseslerin yakın zamanda eriştikleri sayfaların büyük kısmı bellekte olur. *Sayfa hatası* düşük olur.
- Sayfa boyu büyük olunca sayfalarda yakın zamanlı erişimlere uzak kısımlar da olur. Sayfa hataları artar.

Sayfa Boyu

- Birden fazla sayfa boyu olabilir.
- Büyük sayfa boyları program komut bölümleri için kullanılabilir
- Küçük boylu sayfalar iplikler için kullanılabilir
- Çoğu işletim sistemi tek sayfa boyu destekler

Örnek Sayfa Boyları

Computer	Page Size
Atlas	512 48-bit words
Honeywell-Multics	1024 36-bit word
IBM 370/XA and 370/ESA	4 Kbytes
VAX family	512 bytes
IBM AS/400	512 bytes
DEC Alpha	8 Kbytes
MIPS	4 kbyes to 16 Mbytes
UltraSPARC	8 Kbytes to 4 Mbytes
Pentium	4 Kbytes or 4 Mbytes
PowerPc	4 Kbytes
Itanium	4 Kbytes to 256 Mbytes

Alma Yaklaşımları

- Alma yöntemi
  - Sayfanın belleğe ne zaman alınması gerektiğini belirler
  - "*İsteğe dayalı sayfalama*" kullanılıyorsa ancak sayfaya erişim olduğunda belleğe getirir
    - sayfa hatası başta daha yüksek
  - "Önceden sayfalama" yöntemi kullanıldığında gerektiğinden daha fazla sayfa belleğe alınır
    - Diskte birbirini izleyen konumlarda yer alan sayfaları birlikte belleğe getirmek daha etkin

Yerine Koyma Yaklaşımları

- Yerleştirme yöntemi
  - Hangi sayfanın yerine konacak?
  - Bellekten atılacak sayfa yakın zamanda erişilmesi olasılığı düşük olan bir sayfa olmalı.
  - Çoğu yöntem bir prosesin gelecek davranışını eski davranışına dayanarak kestirmeye çalışır.

Yerine Koyma Yaklaşımları

- Çerçeve kilitleme
  - Bir çerçeve kilitliyse yerine başkası yerleştirilemez
  - İşletim sistemi çekirdeği
  - Kontrol yapıları
  - G/Ç tamponları
  - Her çerçeveye bir kilit biti atanması

Temel Yerine Koyma Algoritmaları

- Optimal yöntem
  - Bir sonraki erişimin olacağı zamanın en uzak olduğu sayfanın seçilmesi
  - Gelecek olaylar hakkında kesin bilgi olması imkansız

Temel Yerine Koyma Algoritmaları

- En Uzun Süredir Kullanılmamış (Least Recently Used (LRU)) yöntemi
  - En uzun zamandır erişim olmamış olan sayfayı seçer
  - Yerellik prensibine göre yakın zamanda da bu sayfaya erişim olmayacaktır
  - Her sayfada en son erişim zamanı bilgisi tutulur. Ek yük getirir.

Temel Yerine Koyma Algoritmaları

- İlk Giren İlk Çıkar (First-in, first-out (FIFO))
  - Prosese atanmış sayfa çerçevelerini çevrel bir kuyruk olarak ele alır.
  - Sayfalar sıralı olarak bellekten atılır
  - Gerçeklenmesi en basit yöntem
  - Bellekte en uzun kalmış sayfanın yerine konur. Ancak bu sayfalara yakın zamanda erişim olabilir!

Temel Yerine Koyma Algoritmaları

- Saat yöntemi
  - Kullanım biti adını alan ek bit
  - Sayfa belleğe yüklendiğinde kullanım bitine 1 yüklenir
  - Sayfaya erişimde kullanım biti bir yapılır
  - Çevrel kuyruk tutulur. İşaretçi en son belleğe alınan sayfanın bir sonrasını gösterir.
  - Bellekten atılacak sayfa belirlenirken bulunan kullanım biti 0 olan ilk sayfa seçilir.
  - Atılacak sayfa belirlenirken 1 olan kullanım bitleri de sıfırlanır.
  - Kullanım biti 0 olan yoksa ilk tur tamamlanır, ikinci turda daha önce sıfır yaptıklarının ilki seçilir.

Temel Yerine Koyma Algoritmaları

- Sayfa tamponlama
  - Bellekten atılan sayfa şu iki listeden birisine eklenir:
    - sayfada değişiklik olmadıysa boş sayfalar listesine
    - değişiklik yapılmış sayfalar listesine

Temizleme Yaklaşımları

- İsteğe dayalı temizlik
  - sayfa ikincil belleğe ancak seçilirse atılır
- Önceden temizlik
  - sayfalar gruplar halinde ikincil belleğe atılır

Temizleme Yaklaşımları

- En iyi yaklaşım sayfa tamponlama ile
  - Atılacak sayfalar iki listeden birisine konulur
    - Değişenler ve değişmeyenler
  - Değişenler listesindeki sayfalar periyodik olarak gruplar halinde ikincil belleğe alınır
  - Değişmeyenler listesindekiler yeniden erişim olursa yeniden kullanılır ya da çerçevesi başka sayfaya verilirse kaybedilir

LINUX'ta Bellek Yöntemi

- Görüntü Bellek Adresleme
  - 3 seviyeli sayfa tablosu yapısı şu tablolardan oluşur: (her tablo bir sayfa boyunda)
    - sayfa kataloğu
      - her aktif prosesin bir sayfa kataloğu var.
      - boyu bir sayfa
      - her kayıt orta aşama sayfa kataloğunun bir sayfasına işaret
      - her aktif proses için bellekte yer almalı

LINUX'ta Bellek Yöntemi

- orta aşama sayfa kataloğu
  - birden fazla sayfadan oluşabilir
  - her kayıt sayfa tablosunda bir sayfaya işaret eder
- sayfa tablosu
  - birden fazla sayfadan oluşabilir
  - her kayıt prosesin bir sanal sayfasına işaret eder

LINUX'ta Bellek Yöntemi

- görüntü adres 4 alandan oluşur
  - en yüksek anlamlı alan sayfa kataloğundaki bir kayda indis
  - ikinci alan orta aşama sayfa kataloğundaki bir kayda indis
  - üçüncü alan sayfa tablosundaki bir kayda indis
  - dördüncü alan seçilen sayfadaki offset adresi
- platformdan bağımsız olarak 64 bitlik adresler

LINUX'ta Bellek Yöntemi

- Sayfa atama
  - buddy sistemi kullanılır
- Sayfa yerine koyma
  - sayfa yöntemine dayalı bir yaklaşım
  - kullanım biti yerine 8 bitlik yaş alanı var
  - yaşlı olan (uzun zamandır erişilmemiş) sayfalar öncelikle bellekten atılır

8 DOSYA SİSTEMİ

Bilgilerin Uzun Vadeli Saklanması

- saklanacak veriler çok fazla olabilir
- veriler proses sonlandıktan sonra da kaybolmamalı
- bilgiye prosesler ortak olarak ulaşabilmeli

Dosya Sistemi Görevleri

- dosya isimlendirme
- dosyalara erişim
- dosyaların kullanımı
- koruma ve paylaşım
- gerçekleme

Dosya Sistemi Özellikleri

- kullanıcı açısından
  - dosyaların içerikleri
  - dosya isimleri
  - dosya koruma ve paylaşma
  - dosya işlemleri
  - ...

⇒ Kullanıcı arayüzü

- tasarımcı açısından
  - dosyaların gerçeklenmesi
  - boş alanların tutulması
  - mantıksal blok boyu
  - ....

⇒ Dosya sistemi gerçeklemesi

Dosya Tipleri

- Dosyalar
  - ASCII dosyalar
  - ikili dosyalar
- Kataloglar
  - çoğu işletim sisteminde katalog = dosya

Dosya İçi Erişim

- sıralı erişim
- rasgele erişim

Dosyaların Özellikleri (Attribute)

- erişim hakları
- parola
- yaratıcı
- sahibi
- salt oku bayrağı
- saklı bayrağı
- sistem bayrağı
- arşiv bayrağı
- ASCII/ikili dosya bayrağı
- rasgele erişim bayrağı
- geçici bayrağı
- kilit bayrakları
- kayıt uzunluğu
- anahtar konumu
- anahtar uzunluğu
- yaratılma zamanı
- son erişim zamanı
- son değişiklik zamanı
- dosya boyu
- maksimum dosya boyu

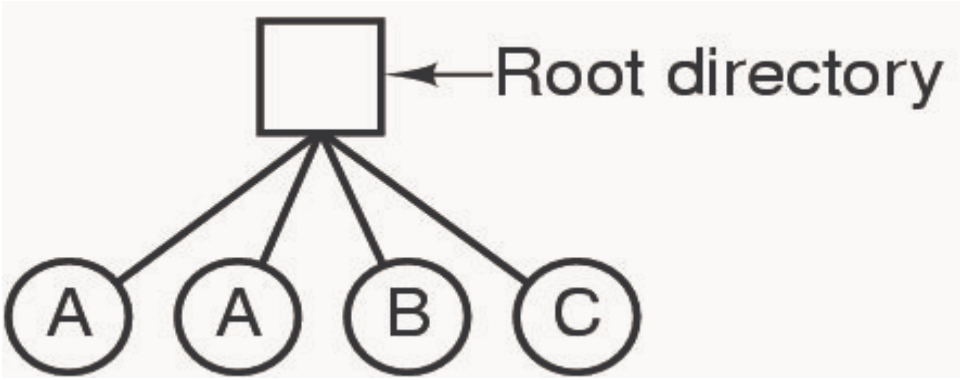
Dosya İşlemleri

- yaratma / silme
- isim değiştirme
- açma / kapama
- yazma / okuma / ekleme
- dosya işaretçisi konumlandırma
- özellik sorgulama / değiştirme

⇒ sistem çağrıları ile (*open*, *creat*, *read*, *write*, *close*, ...)

Tek Seviyeli Katalog Sistemleri

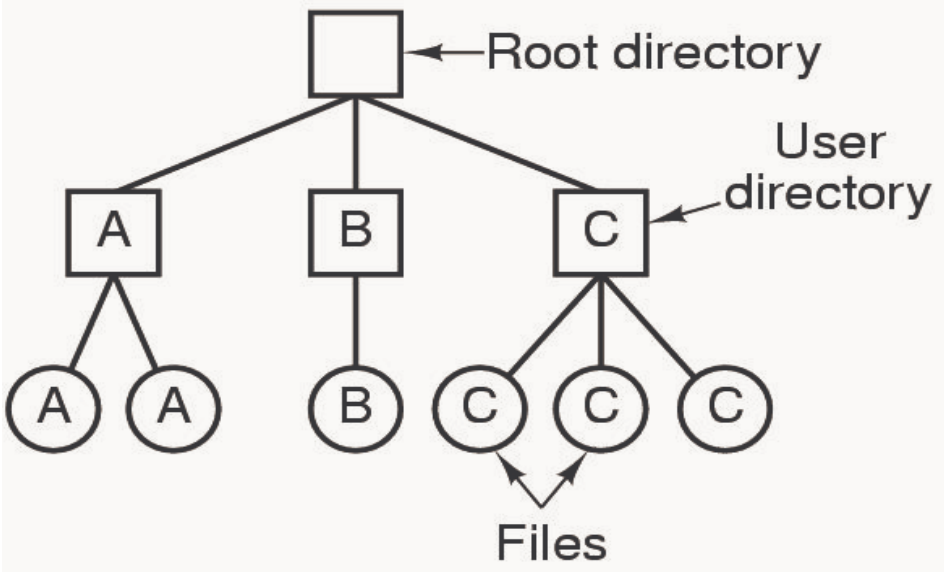
- tek seviyeli katalog
- hızlı erişim
- çok kullanıcılı sistemlere uygun değil
  - farklı kullanıcılar aynı isimli dosya yaratırsa sorun olur
- günümüzde gömülü sistemlerde
  - örneğin arabada kullanıcı profilleri saklanması



(**Not:** örnekte dosya isimleri değil sahipleri gösterilmiştir.)

İki Seviyeli Katalog Sistemleri

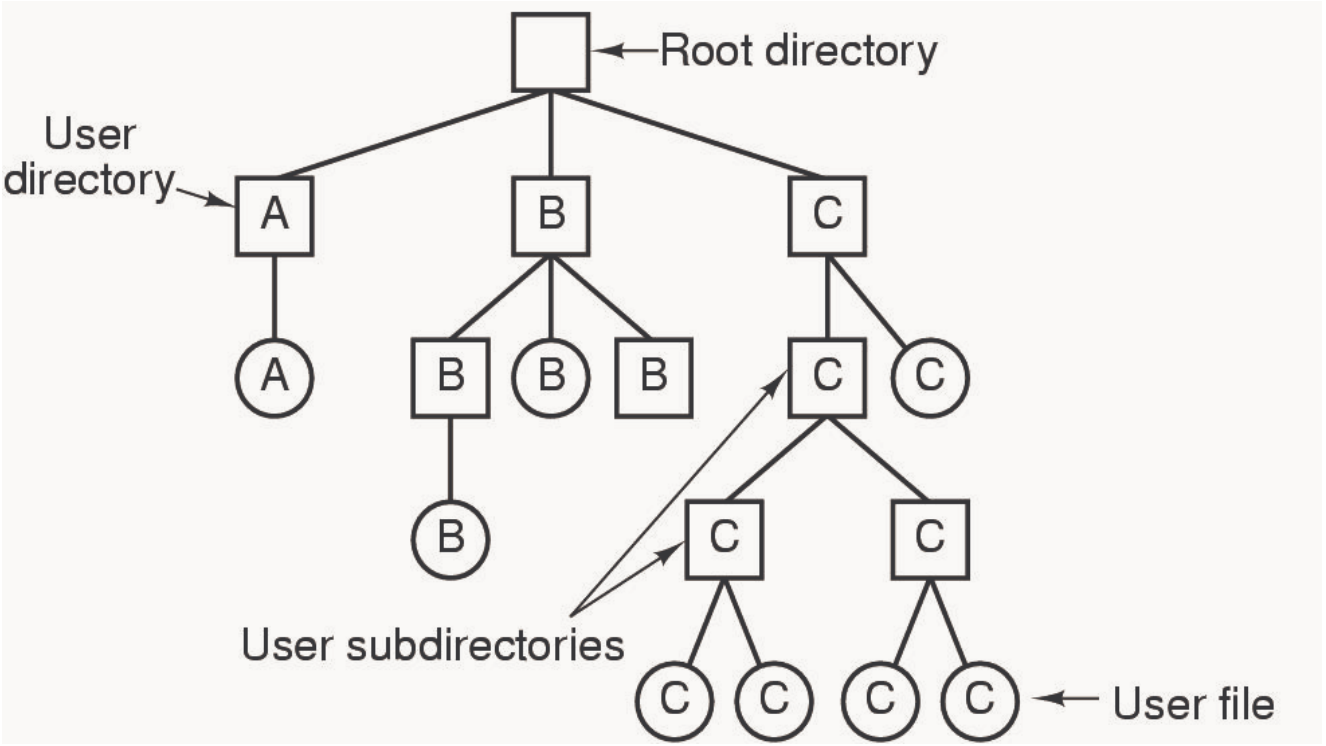
- her kullanıcıya ayrı katalog
  - kullanıcıların aynı isimli dosya sorunu çözülür
- örneğin çok kullanıcılı kişisel bilgisayarlarda
- sisteme kullanıcı adı ve parola ile girme kavramı



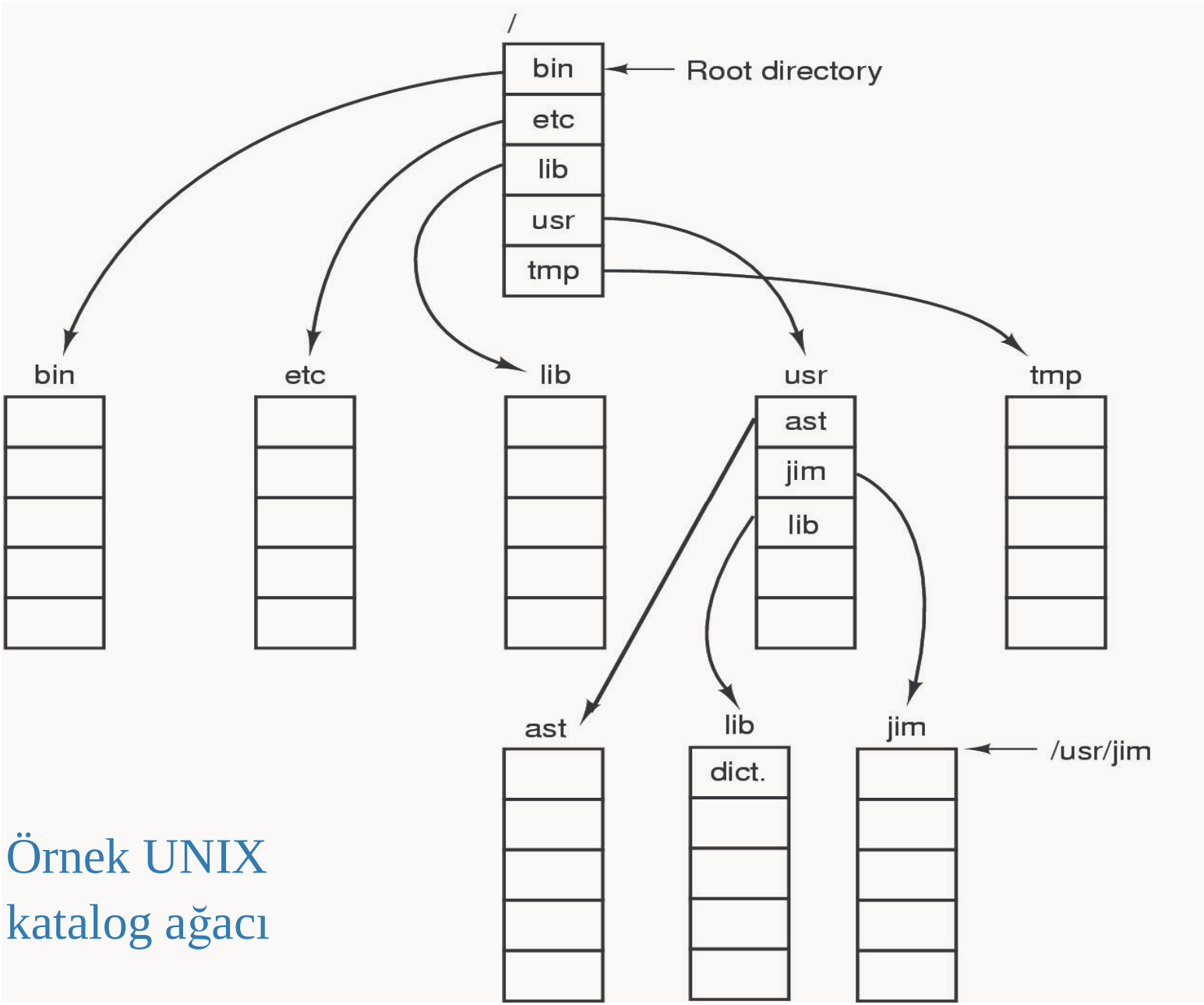
(**Not:** örnekteki harfler katalog ve dosya sahiplerini göstermektedir.)

Hiyerarşik Katalog Sistemleri

- kullanıcıların dosyalarını mantıksal olarak gruplama isteği
- katalog ağacı
- modern işletim sistemlerindeki yapı



(**Not:** örnekteki harfler katalog ve dosya sahiplerini göstermektedir.)



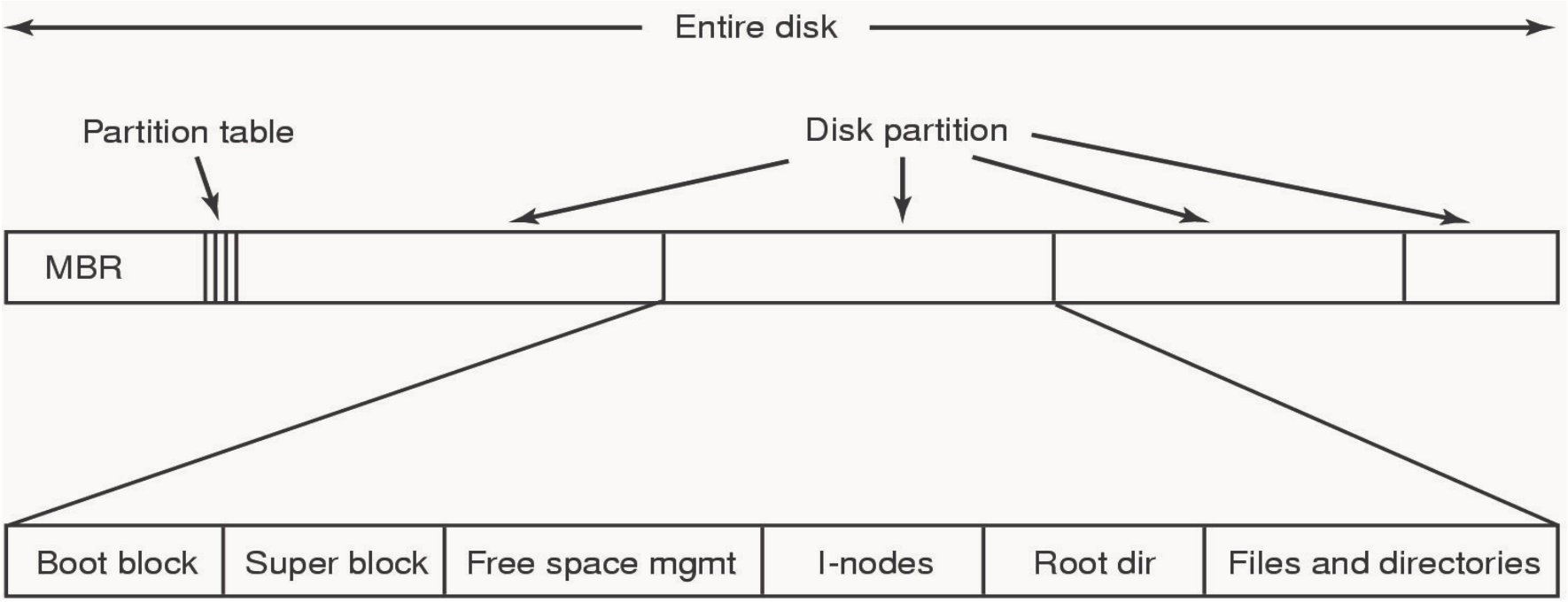
Örnek UNIX katalog ağacı

Katalog İşlemleri

- yaratma / silme
- kataloğu açma / kapama
- kataloğu okuma
  - örneğin dosyaların listelenmesi
  - okumadan önce açılması lazım
- isim değiştirme
- bağla / kopar
  - UNIX'te dosya silmeye özdeş

Dosya Sistemi Gerçeklemesi

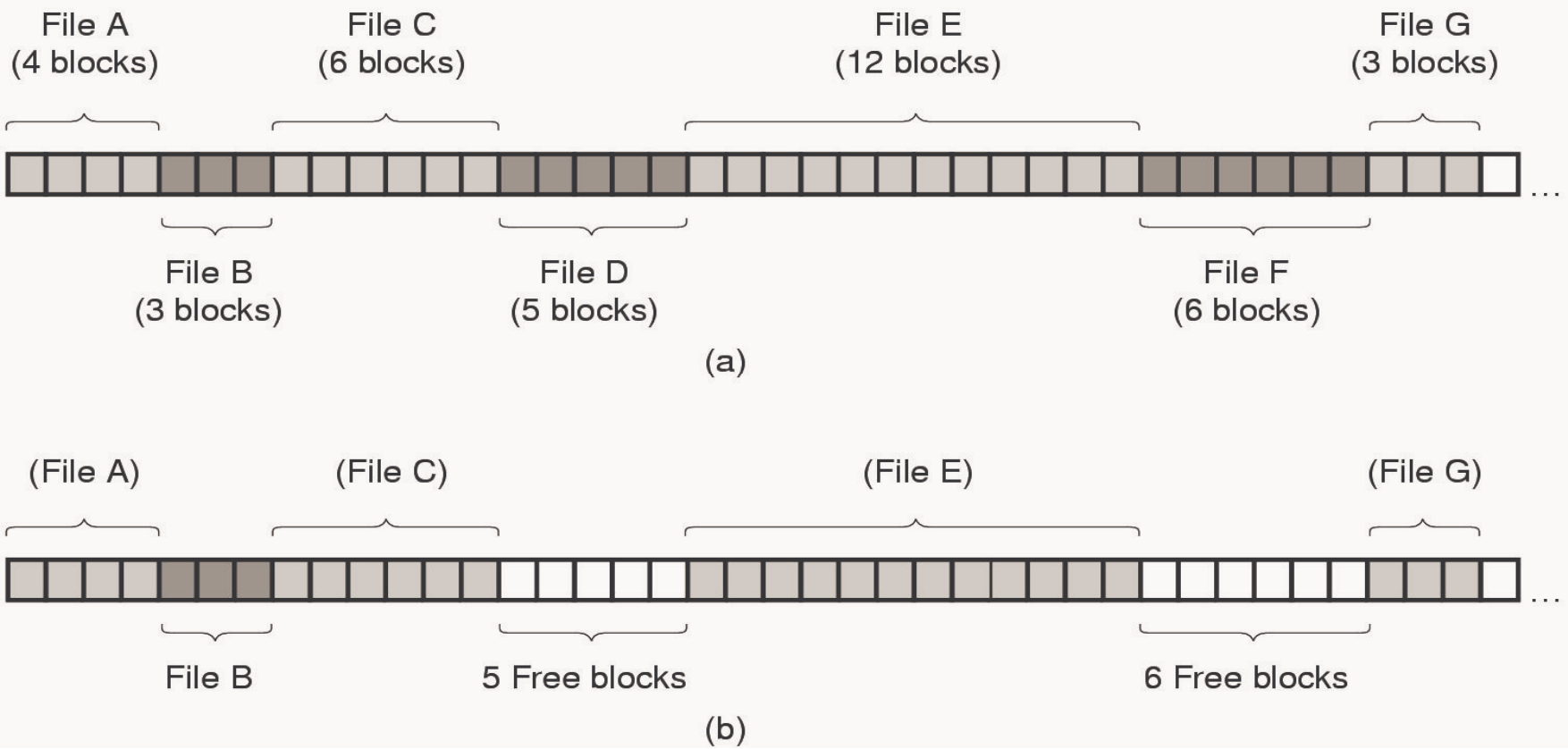
Örnek Dosya Sistemi Yapısı:



Dosya Sistemi Gerçeklemesi (1)

- sürekli yer ayırma ile gerçekleme
  - dosyanın ilk bloğunun adresi ve blok sayısı tutulur
  - avantajları
    - basit gerçekleme
    - daha etkin *okuma* işlemi
  - sorunları
    - diskte parçalanma (fragmentation)
    - sıkıştırma maliyeti yüksek
    - boşluk listesi tutulmalı
      - dosya boyu en baştan bilinmeli ve sonradan değişemez
      - dosyaların maksimum boyları kısıtlı
  - CD-ROM dosya sistemlerine uygun (tek yazımlık)

Dosya Sistemi Gerçeklemesi (1)



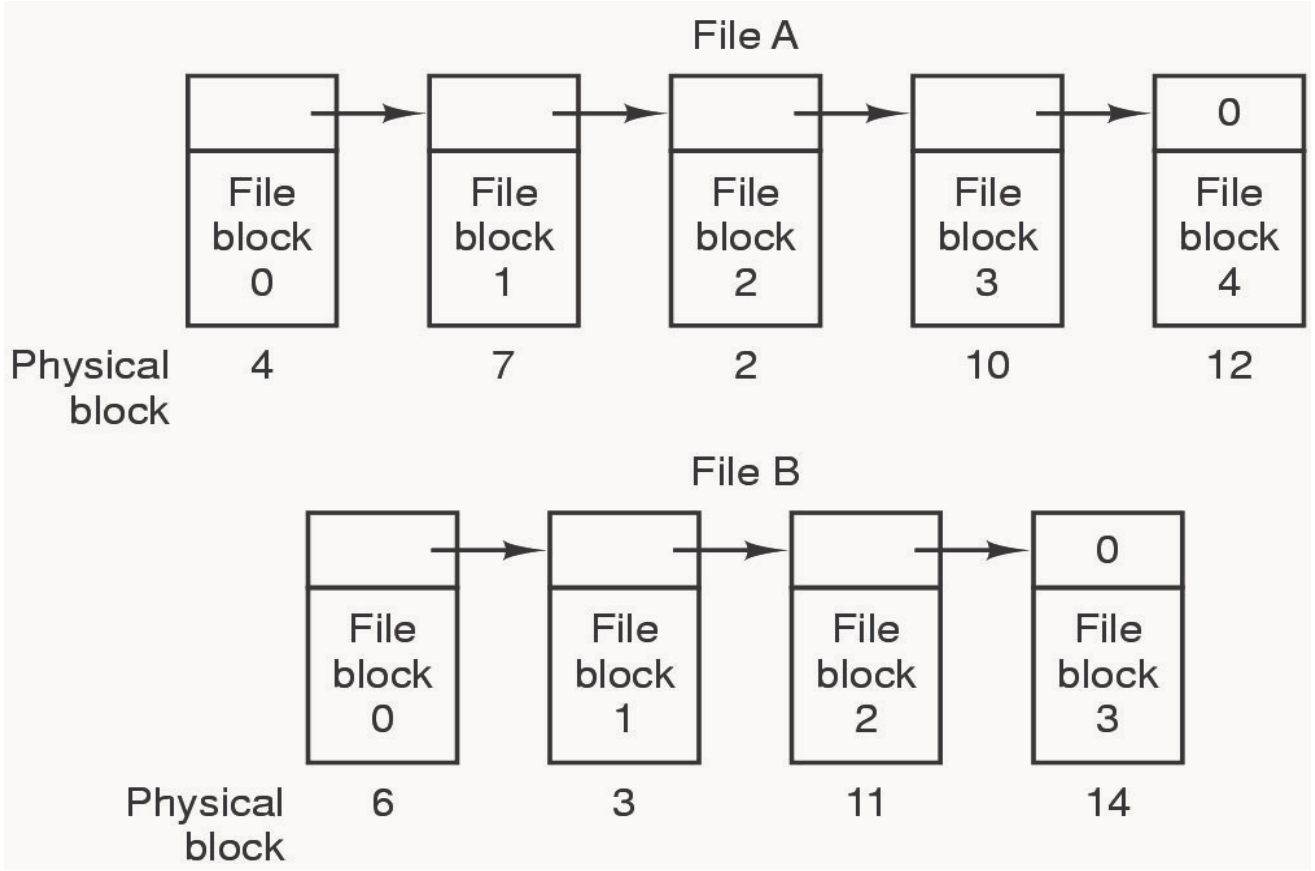
(a) Diskte sürekli yer ayırma: örnek 7 dosya

(b) *D* ve *E* dosyaları silindikten sonra diskin durumu

Dosya Sistemi Gerçeklemesi (2)

- bağlantılı listeler kullanarak gerçekleme
  - her bloğun ilk sözcüğü sıradakine işaretçi
  - parçalanma yok (yanlız son blokta iç parçalanma)
  - yanlız dosyanın ilk bloğunun adresi tutulur
  - dosyadaki verilere erişim
    - sıralı erişim kolay
    - rasgele erişim zor
  - bloktaki veri boyu 2'nin kuvveti değil
    - okumada bloklar genelde 2'nin kuvveti boyunda

Dosya Sistemi Gerçeklemesi (2)



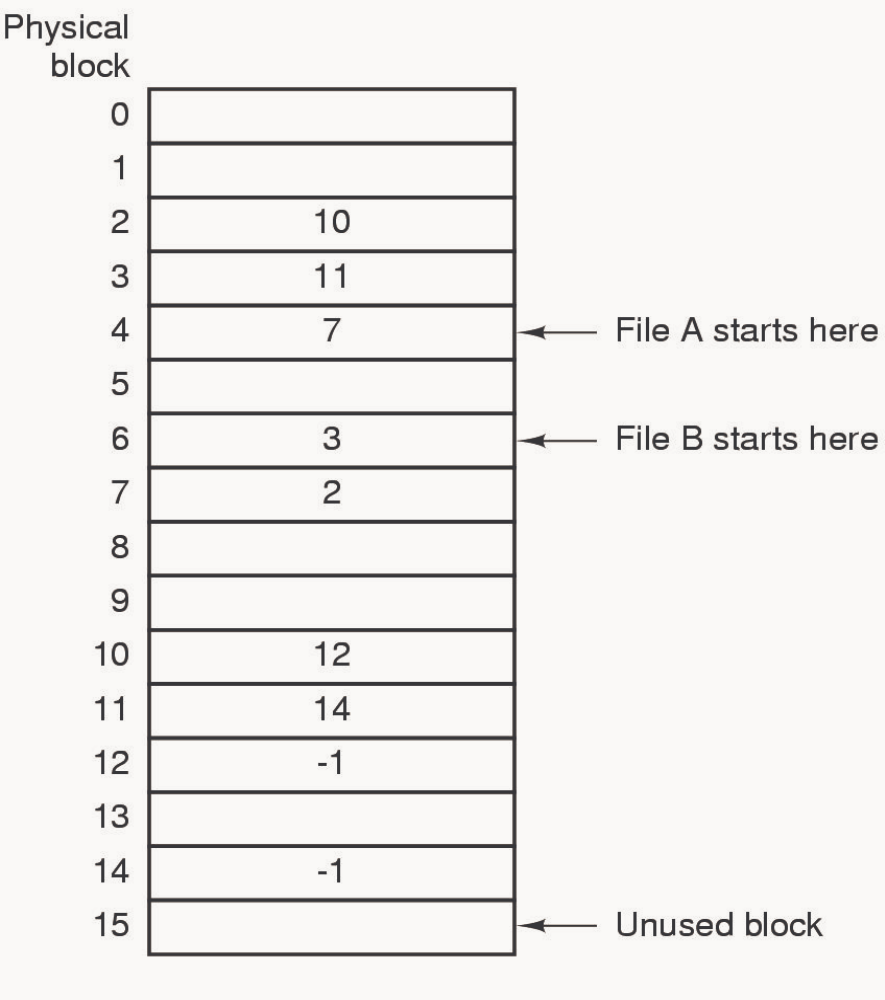
Dosya bloklarının bağlantılı liste yapısında tutulması

Dosya Sistemi Gerçeklemesi (3)

- bellekte dosya tabloları ile gerçekleme
  - işaretçiler diskteki bloklarda değil bellekte tabloda tutulur
  - FAT (File Allocation Table)
  - rasgele erişim daha kolay
    - tablo bellekte
  - başlangıç bloğu bilinmesi yeterli
  - tüm tablo bellekte olmalı!
  - tablo boyu disk boyuna bağlı
    - örnek: 20 GB disk ve blok boyu 1K olsun: tabloda 20 milyon en az 3 sekizli boyunda kayıt gerekli (20MB)

Dosya Sistemi Gerçeklemesi (3)

Bellekte dosya tablosu tutarak gerçekleme

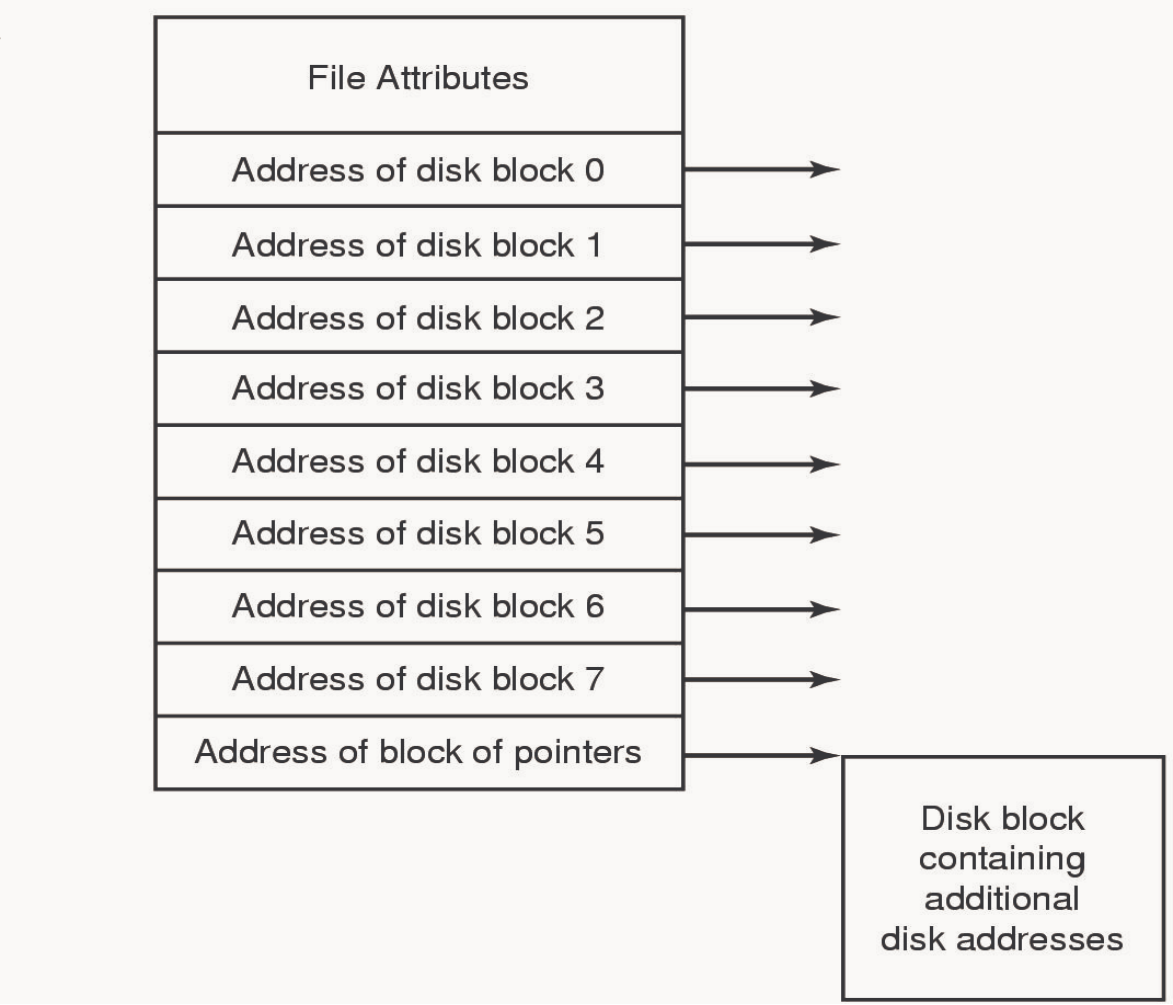


Dosya Sistemi Gerçeklemesi (4)

- her dosyaya ilişkin bir i-node (index-node)
  - dosyanın özellikleri
  - dosyanın bloklarının disk adresleri
- sadece açık dosyaların i-node yapıları bellekte
  - toplam bellek alanı aynı anda açık olmasına izin verilen maksimum dosya sayısı ile orantılı
- basit yapıda dosyanın maksimum blok sayısı kısıtlı
  - çözüm: son gözü ek tabloya işaretçi

Dosya Sistemi Gerçeklemesi (4)

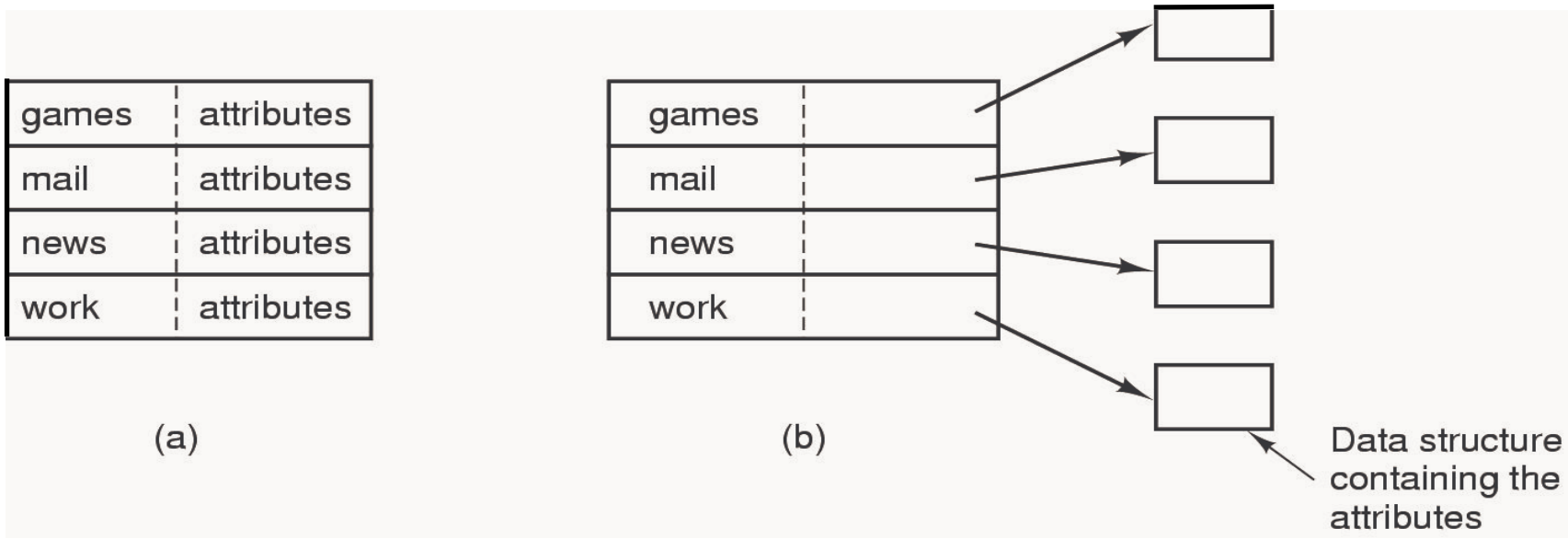
Örnek i-node yapısı



Katalogların Gerçeklenmesi (1)

- dosya adı ile diskteki bloklar ile ilişki kurulması
- dosya özellikleri nerede tutulmalı ?
  - katalog yapısı içinde ?
    - katalog yapısı: sabit boyda dosya adı, dosya özellikleri, disk blokları adresleri
      - MS-DOS / Windows
    - katalog yapısı: dosya adı, i-node numarası
      - dosya özellikleri i-node içinde
      - UNIX
- güncel işletim sistemlerinde dosya isimleri uzun olabilir
  - isim için uzun sabit bir üst sınır tanımla
    - gereksiz yer kaybı

Katalogların Gerçeklenmesi (1)



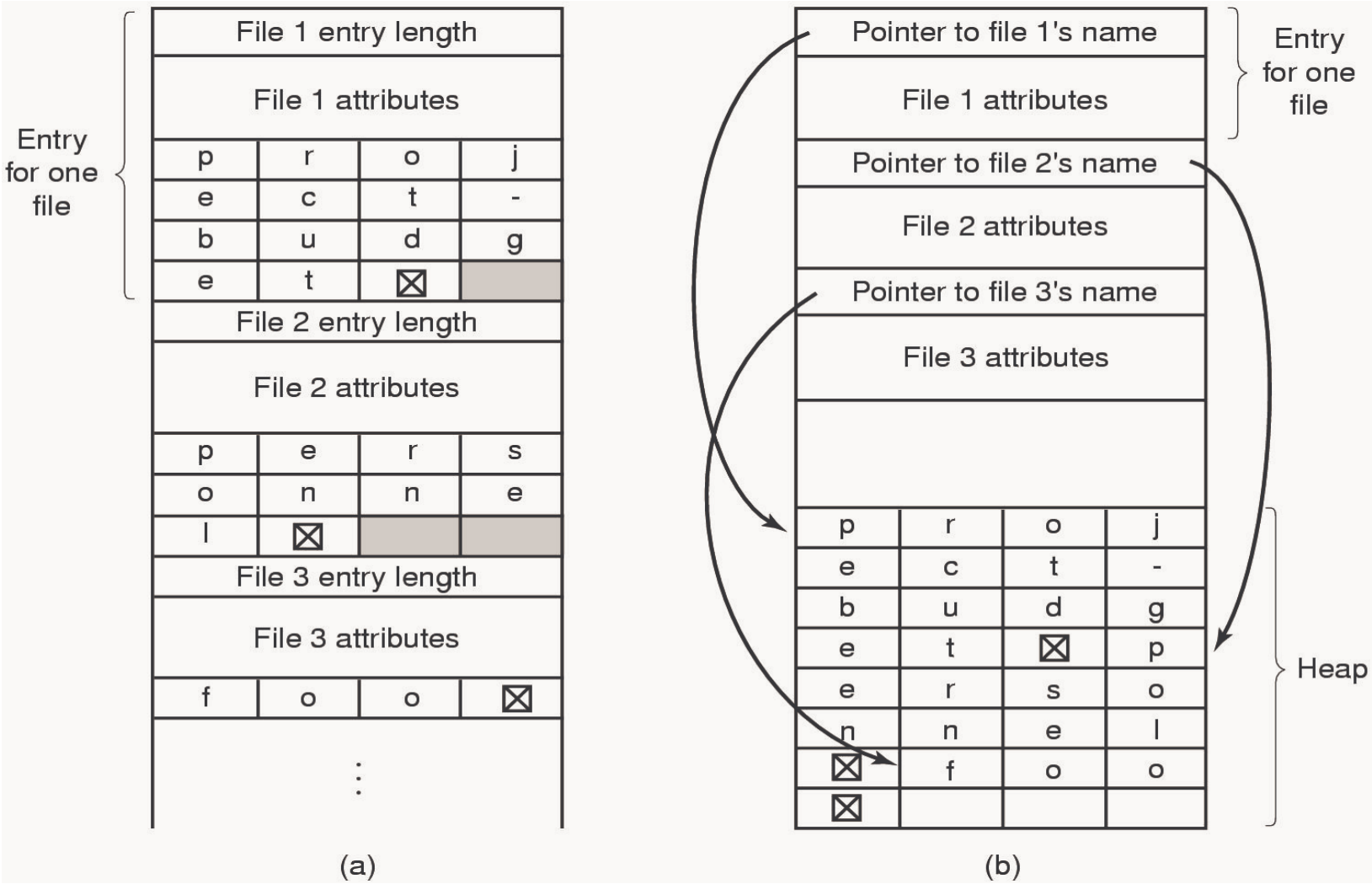
(a) Basit katalog yapısı

sabit uzunluklu dosya adı

disk adres ve dosya özellik bilgileri

(b) Her kayıt bir i-node yapısını gösterir

Katalogların Gerçeklenmesi (2)



Uzun dosya adlarının tutulması:

- (a) sıralı
- (b) en sonda

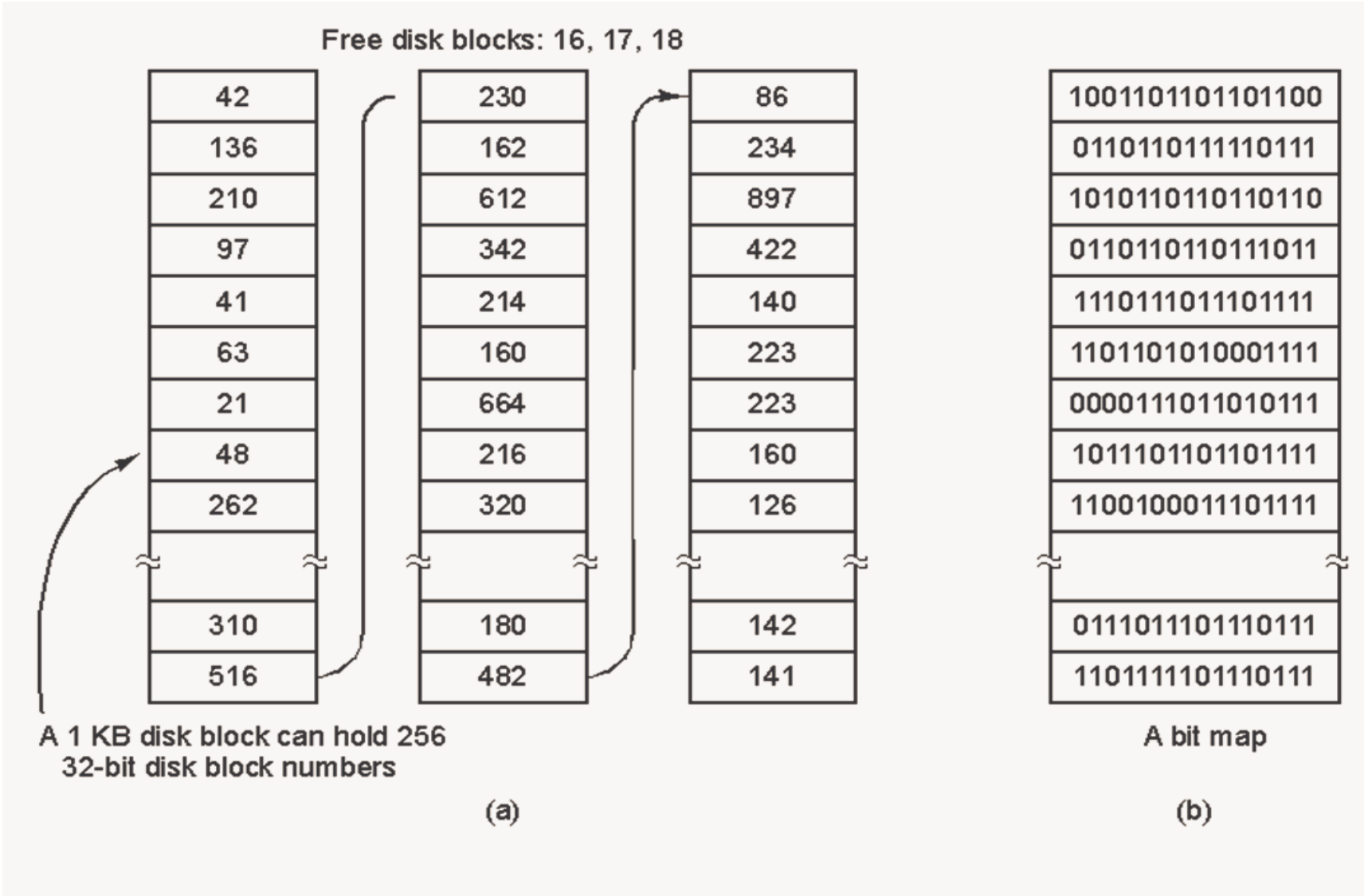
Katalogların Gerçeklenmesi (2)

- dosya isimlerinin sıralı tutulmasının sorunları
  - dosya isimleri sözcük başlarında başlamalı
    - sona boşluk ekle
  - dosya silinince değişken boylu bölgeler boşalır
    - sıkıştırmak kolay
  - bir kayıt birden fazla sayfa boyunda olabilir
    - dosya adını okurken olası sayfa hatası
- dosya isimlerinin en sonda tutulması
  - tüm dosyalar için eş boylu alan
  - yine sayfa hatası olabilir
  - dosya isimleri sözcük başında başlamak zorunda değil

Disk Alanı Yönetimi

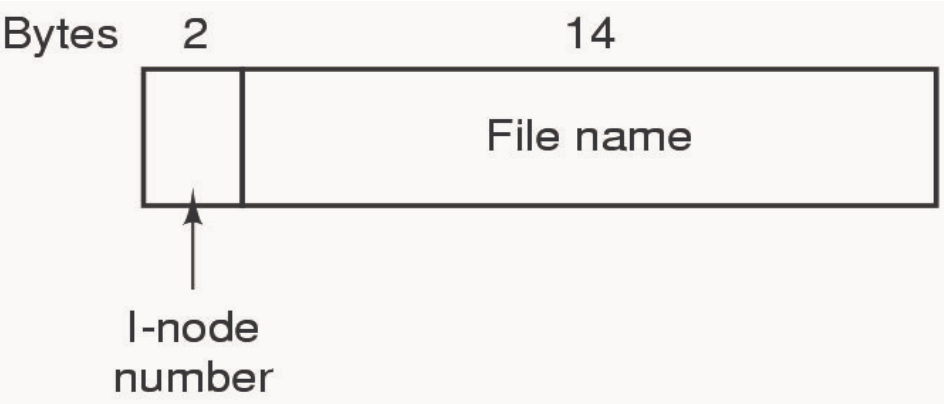
- dosyalar sabit boylu bloklara bölünür
- blok boyu ne olmalı?
  - sektör, iz, silindir boyu?
    - aygıta bağlı
  - boy seçimi önemli
    - başarım ve etkin yer kullanımı çelişen hedefler
    - ortalama dosya boyuna göre seçmek iyi
    - çoğu sistemde çok önceden belirlenmiş
      - UNIX: çoğunlukla 1K

Boş Blokların Yönetimi



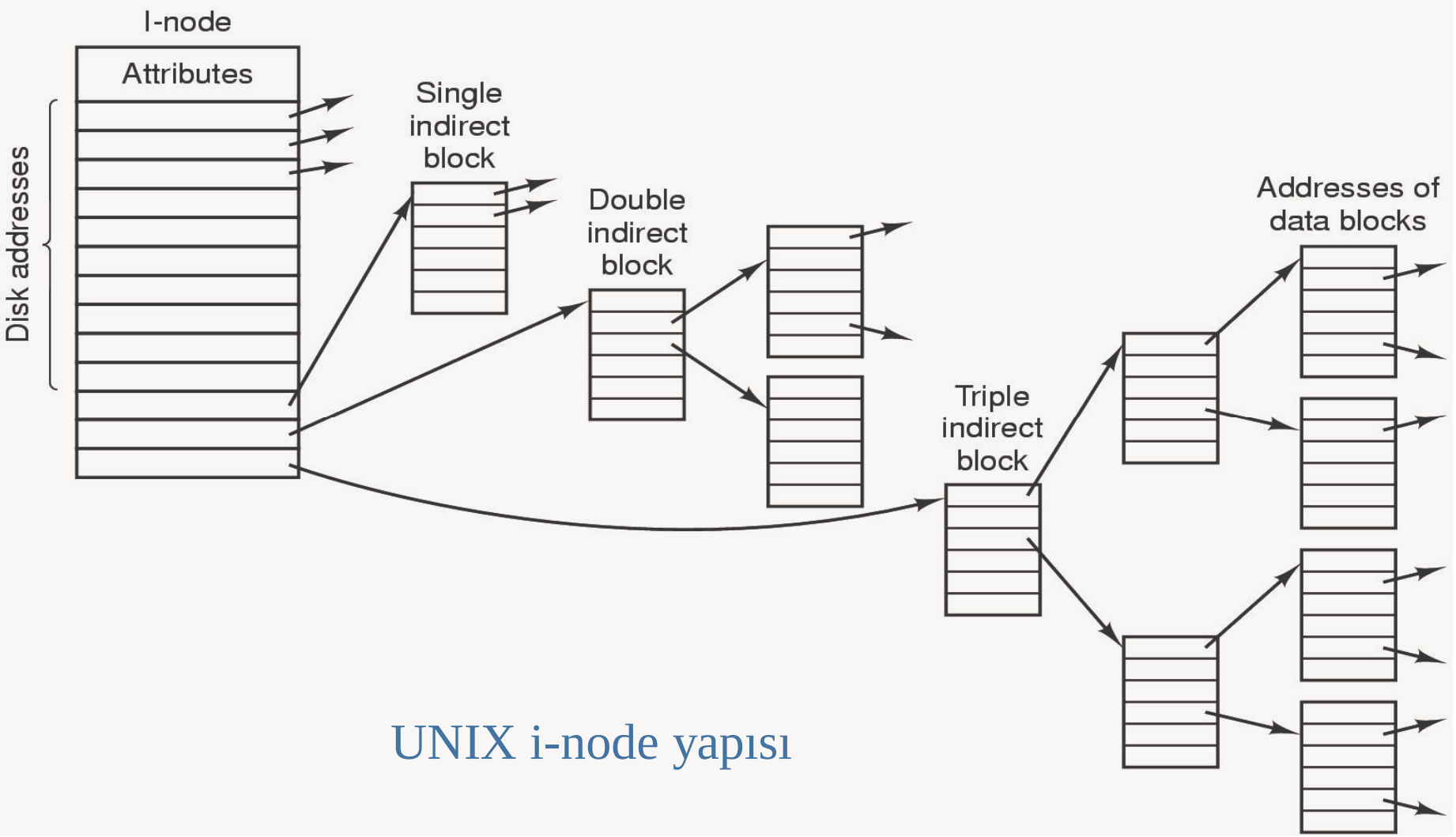
UNIX V7 Dosya Sistemi (1)

- kök kataloğundan başlayan ağaç yapısı
- dosya adı max. 14 karakter
  - / ve NUL olamaz
  - NUL = 0 (isimleri 14 karaktere tamamlamak için)
- katalog yapısında dosya başına bir kayıt
  - dosya adı (14 karakter)
  - i-node numarası (2 sekizli)
- dosya özellikleri i-node yapısında
  - dosya boyu, yaratılma, erişim ve değişim zamanı, sahibi. grubu, koruma bitleri, bağlantı sayısı



UNIX V7 katalog kaydı

UNIX V7 Dosya Sistemi (2)



UNIX i-node yapısı