Bölüm 3: Süreçler

İşletim Sistemleri

Süreç

- Yürütülmekte olan programlar.
- Program belleğe yüklendiğinde süreç halini alır.
- Programlar pasif, süreçler aktif.
- Bir süreç sıralı bir şekilde yürütülür,
  - paralel yürütülemez.

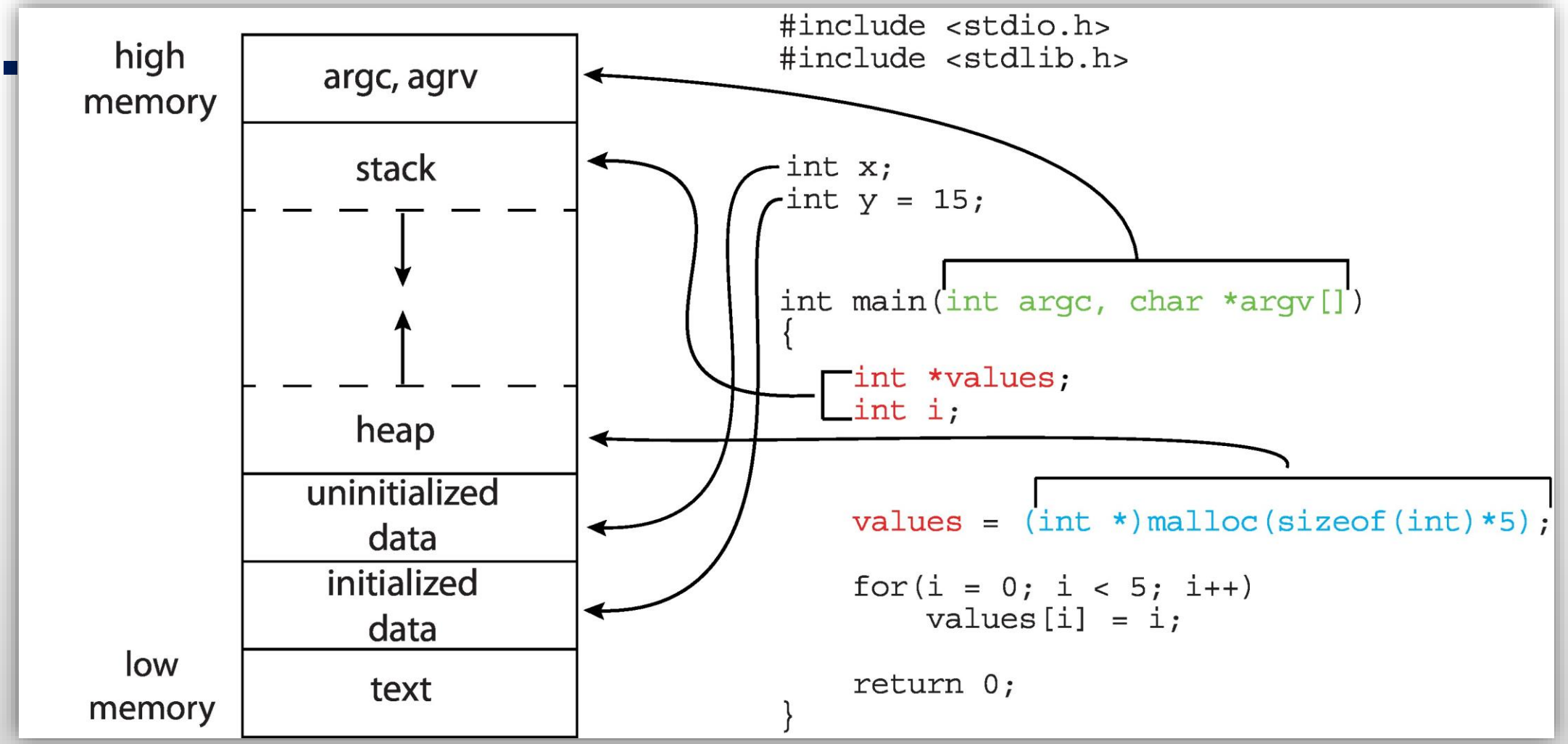
Süreç ve Program Arasındaki Farklar

- Program, bilgisayar komutlarının bir derlemesi. (*compilation*)
- Yürütülebilir dosya halindedir. (ikili, *binary, executable*)
- Program çalıştırıldığında, bir süreç oluşturulur.
- Program diskte, Süreçler bellekte yer alır.
- Bir programdan birden fazla süreç oluşturulabilir.
- Her süreç farklı sistem kaynakları kullanır.
- Süreçler arasında haberleşme, veri paylaşımı ve iş bölümü gerçekleşebilir.

Bellekte Süreç Yerleşimi

- **Metin (text)** bölümü, program kodunu içerir.
- **Program sayacı**, mevcut etkinliği tutar, işlemci içerisinde bir yazmaç.
- **Yığıt (stack),** geçici verileri tutar.
  - Fonksiyon parametreleri, dönüş adresleri, yerel değişkenler.
- **Veri (data)** bölümü, genel (*global*) değişkenleri içerir.
- **Yığın (heap)**, çalışma süresi boyunca dinamik olarak ayrılan belleği içerir.

Bellekte Süreç Yerleşimi

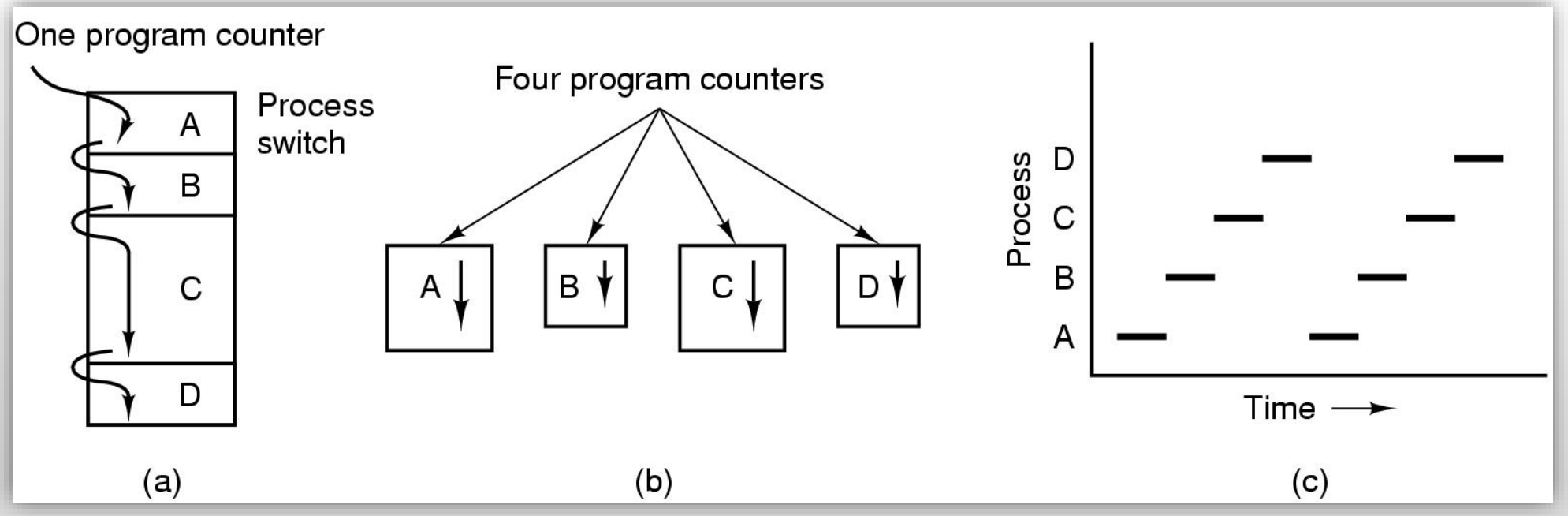


Sözde Paralellik

- Modern bir bilgisayardan aynı anda birçok işlem yürütmesi beklenir.
- Tek işlemcili sistemde, herhangi bir anda, sadece bir işlem yürütülebilir.
- Çoklu programlama sisteminde işlemci,
  - yüzlerce *ms* süresince çalışan süreçler arasında hızlıca geçiş yapar.
- Sözde paralellik kullanıcılar için faydalıdır.
  - Ancak; yönetimi zordur.

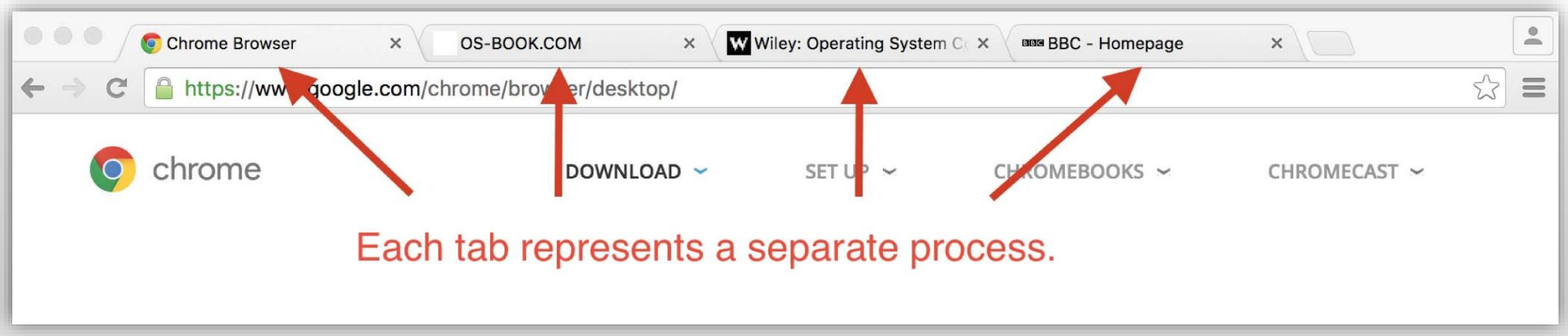
Çoklu Programlama Süreç Modeli

(a) Dört sürecin çoklu programlanması. (b) Birbirinden bağımsız dört sürecin kavramsal modeli. (c) Aynı anda bir süreç etkin.



Chrome

- **Browser**, kullanıcı arayüzü, disk ve ağ arayüzlerini yönetir.
- **Renderer**, web sayfası, HTML, javascript kodları ile ilgilenir.
- **Plug-in**, her bir plug-in için.

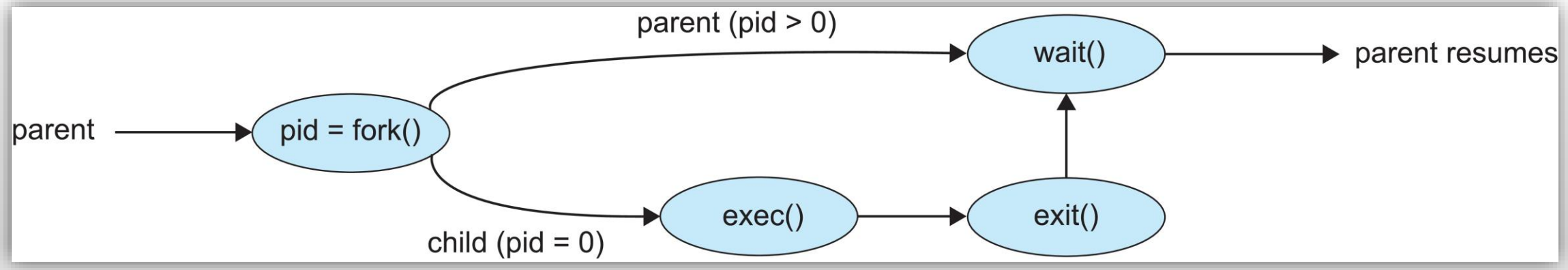


Süreç Başlatma

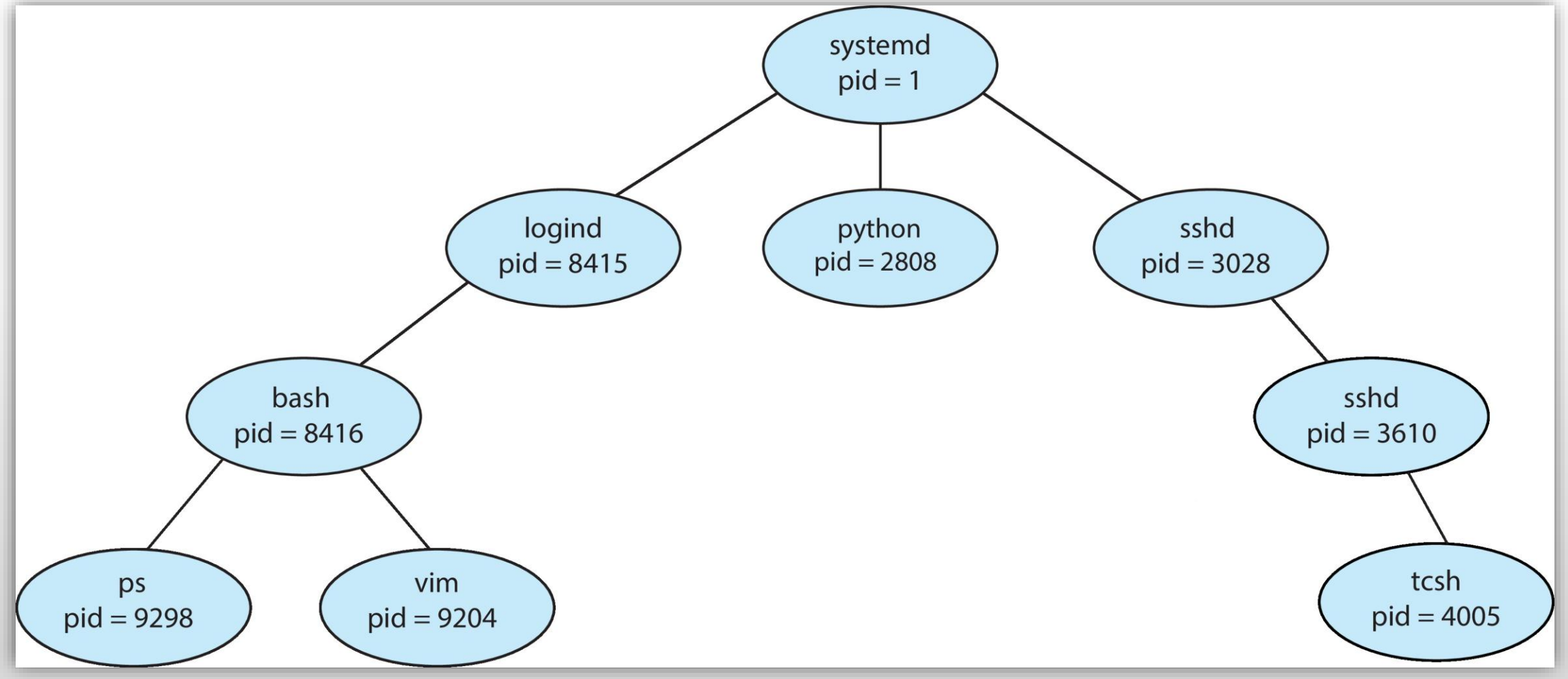
- Süreç oluşturmaya neden olan olaylar:
  - Sistem başladığında otomatik olarak.
  - Çalışan bir süreç tarafından sistem çağrısının yürütülmesi.
  - Yeni bir süreç oluşturmak için kullanıcı isteği.
  - Toplu işin başlatılması. (*batch*)

Süreç Başlatma

- Ata süreç, çocuk süreç başlatabilir. Tüm kaynakları paylaşabilirler. Paralel olarak çalışırlar. Ata süreç, çocuk sürecin sonlanmasını bekler. Çocuk süreç, ebeveyn sürecin adres uzayını kullanır.



Süreç Başlatma



Süreç Sonlandırma

- Süreç sonlandırılmasına neden olan olaylar:
  - Normal çıkış (*gönüllü*).
  - Hata sonrası çıkış (*gönüllü*).
  - Ölümcül (*fatal*) hata sonrası çıkış (*istemsiz*).
  - Başka bir süreç tarafından sonlandırılma (*kill*) (*istemsiz*).

Ölümcül Hatalar (Fatal Errors)

- Ölümcül hatalar;
  - Bellek bölgesine izinsiz erişim.
  - Bellek taşmaları veya bellek sızıntıları.
  - Sıfıra bölme.
  - Dosya, ağ veya diğer kaynaklara hatalı erişim.
  - Kaynakların doğru şekilde serbest bırakılmaması.
- Ölümcül bir hata durumunda, işletim sistemi ilgili süreci sonlandırır.
- Süreç bellekten temizlenir ve sistem kaynakları serbest bırakılır.
- **Crash Dump Analizi**: hatanın nedenlerini belirlemek için kullanılır.

Süreç Sonlandırma (Kill Process)

- Süreci aniden durdurma ve sonlandırma işlemidir.
- *kill*, terminalden kullanılan bir komuttur. *PID* ile kullanılır.
  - Her süreç, tekil bir kimliğe (*Process ID - PID*) sahiptir.
  - `kill -9 1234`
- Süreç sonlandırma bir sinyal göndererek yapılır.
  - SIGTERM (15): Süreç kendi kaynaklarını temizleyebilir.
  - SIGKILL (9): Süreç kaynak temizliği yapamaz, aniden zorla kapatılır.
- Aktif terminaldeki süreci durdurmak için *Ctrl*+*C* kısayolu kullanılır.
- **pkill, killall**: İsimle süreç sonlandırma araçları.

Süreç Sonlandırma

- Sürecin sonlanmasını bekleyen başka bir süreç yoksa,
  - (`wait()` çağrılmamışsa) **zombie** olarak adlandırılır.
- Ata süreç, `wait()` çağıramadan sonlanmışsa,
  - **orphan** olarak adlandırılır.
- `exit()` sistem çağrısı ile süreç sonlanabilir.
- `abort()` sistem çağrısı ile süreç sonlandırılabilir.

Zombi Süreç (Zombie Process)

- İşletim sistemi görevini tamamlayan süreci hemen temizlemez.
- Bu süreç, ölü bir süreçtir. Ancak, PID'si hala süreç tablosunda bulunur.
- Zombi süreçler genellikle ataları tarafından beklenen çocuk süreçleridir.
- Ata süreç, çocuğunun tamamlandığını öğrenmek için bekleyebilir.
- İşletim sistemi, çocuk sürecin *geri dönüş durumu*nu ata süreçten alıp kaynakları temizler.

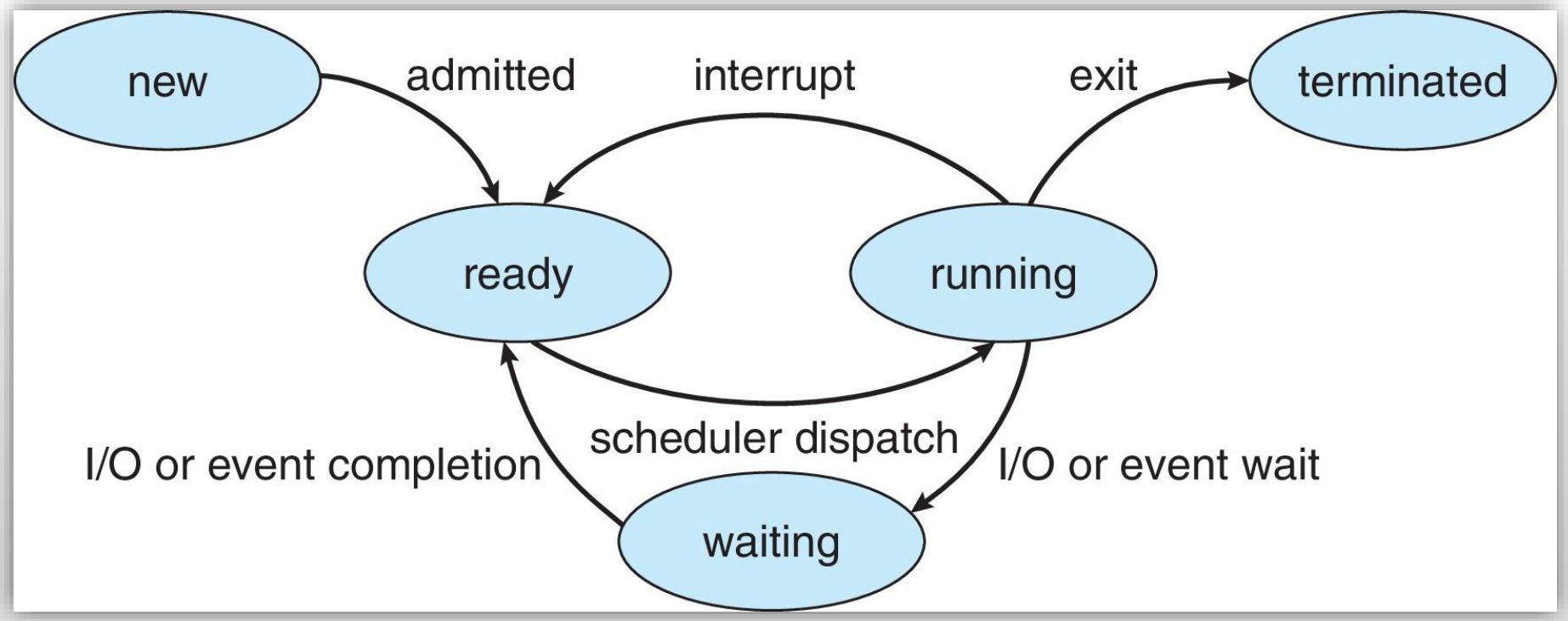
Yetim Süreç (Orphan Process)

- Bir sürecin atası sonlandığında yetim kalır.
- Bu durumda, işletim sistemi yetim sürecin atasını *init* (*PID 1*) olarak atar.
- Yetim süreçler, ata sürecin sonlandığı, ancak; çocuk sürecin tamamlanmadığı durumları ifade eder.
- Yetim süreçler, sistem kaynaklarını tutmaya devam eder.

Süreç Durumları

- Bir süreç yürütülürken durum (*status*) değiştirir.
  - **Yeni**: Süreç oluşturuldu. (*new*)
  - **Çalışıyor**: Talimatlar yürütülüyor. (*running*)
  - **Bekliyor**: Bir olayın gerçekleşmesi bekleniyor. (*waiting*)
  - **Hazır**: İşlemciye atanma bekleniyor. (*ready*)
  - **Sonlandırıldı**: Yürütme tamamlandı. (*terminated*)

Süreç Durumları



Süreç Tablosu

- Süreç tablosunda (*process control block*) bulunan alanlar.
- **Süreç yönetimi**
  - *Registers, Program counter, Program status word, Stack pointer, Process state, Priority, Scheduling parameters, Process ID, Parent process, Process group, Signals, Time when process started, CPU time used, Children's CPU time, Time of next alarm*
- **Bellek yönetimi**
  - *Pointer to {text, data, stack} segment info*
- **Dosya yönetimi**
  - *Root directory, Working directory, File descriptors, User ID, Group ID*

Süreç Tablosu

- **Yazmaçlar** (*registers*): CPU içinde bulunan hızlı ve küçük depolama alanları; geçici veri saklama ve işleme için kullanılır.
- **Program Sayacı** (*counter*): İşletilecek bir sonraki komutun bellek adresini tutar.
- **Program Durum Sözcüğü** (*status word*): İşlemcinin mevcut durumu hakkında durum bayrakları ve kontrol bilgilerini içerir.
- **Yığın İşaretçisi** (*stack pointer*): Bellekteki yığının üstünü işaret eder; fonksiyon çağrıları ve yerel değişken yönetimi için kullanılır.
- **Süreç Durumu** (*process state*): Bir sürecin mevcut durumunu temsil eder (örneğin, çalışıyor, bekliyor veya sonlandırıldı).

Süreç Tablosu

- **Öncelik** (*priority*): Bir sürece atanmış öncelik seviyesi; çoklu görev sistemlerinde çizelgelemeyi etkiler.
- **Çizelgeleme Parametreleri** (*scheduling parameters*): Sürecin çizelgeleme algoritmasını etkileyen parametreler; öncelik, işlem süresi vb.
- **Süreç Kimliği** (*process identifier*): İşletim sistemi tarafından her sürece atanan benzersiz (*unique*) tekil tanımlayıcı.
- **Ata Süreç** (*parent process*): Mevcut süreci oluşturan veya başlatan süreç.
- **Süreç Grubu** (*process group*): Bir veya daha fazla süreci tek bir varlık gibi yönetebilen bir grup.

Süreç Tablosu

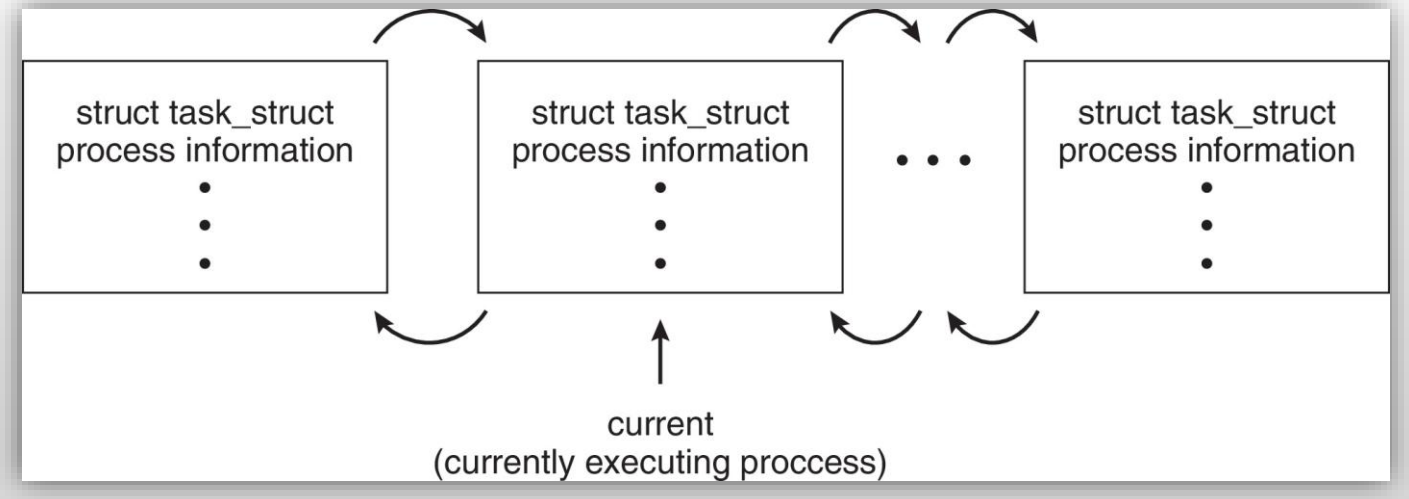
- **Sinyaller** (*signals*): Süreçler arasında iletişim veya bir sürece belirli olayları bildirme mekanizmaları.
- **Başlama Zamanı** (*time when started*): Sürecin başlatıldığı anı gösteren bir zaman damgası.
- **Kullanılan CPU Zamanı** (*CPU time used)*: Süreç tarafından tüketilen toplam CPU zamanı.
- **Çocukların CPU Zamanı** (*children's CPU time*): Mevcut sürecin tüm çocuk süreçlerince tüketilen toplam CPU zamanı.
- **Sonraki Alarm Zamanı** (*time of next alarm*): Sürecin belirli bir sinyal alması veya bir olayı tetiklemesi için planlanan zaman.

Süreç Tablosu

- **{Text, Data, Stack} Segment İşaretçisi**: kod, veri ve yığını gösteren bellek kesimlerinin işaretçileri.
- **Kök Dizin** (*root directory*): Dosya sisteminde en üst düzey dizin.
- **Çalışma Dizini** (*working directory*): Bir sürecin çalıştığı mevcut dizin.
- **Dosya Tanımlayıcıları** (*file descriptors*): Açık dosyalarla ilişkilendirilen tanımlayıcılar; giriş/çıkış akışlarını temsil eder.
- **Kullanıcı Kimliği** (*User ID*): İşletim sistemi tarafından her kullanıcıya atanan tekil tanımlayıcı.
- **Grup Kimliği** (*Group ID*): İşletim sistemi tarafından her kullanıcı grubuna atanan tekil tanımlayıcı.

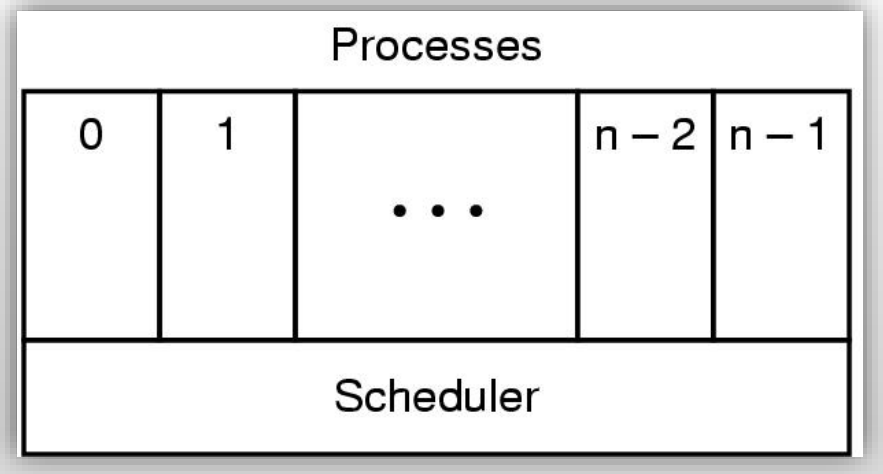
Süreçlerin Çizelgelenmesi

```
pid t_pid;                      /* process identifier */
long state;                     /* state of the process */
unsigned int time_slice         /* scheduling information */
struct task_struct *parent;     /* this process's parent */
struct list_head children;      /* this process's children */
struct files_struct *files;     /* list of open files */
struct mm_struct *mm;           /* address space of this process */
```

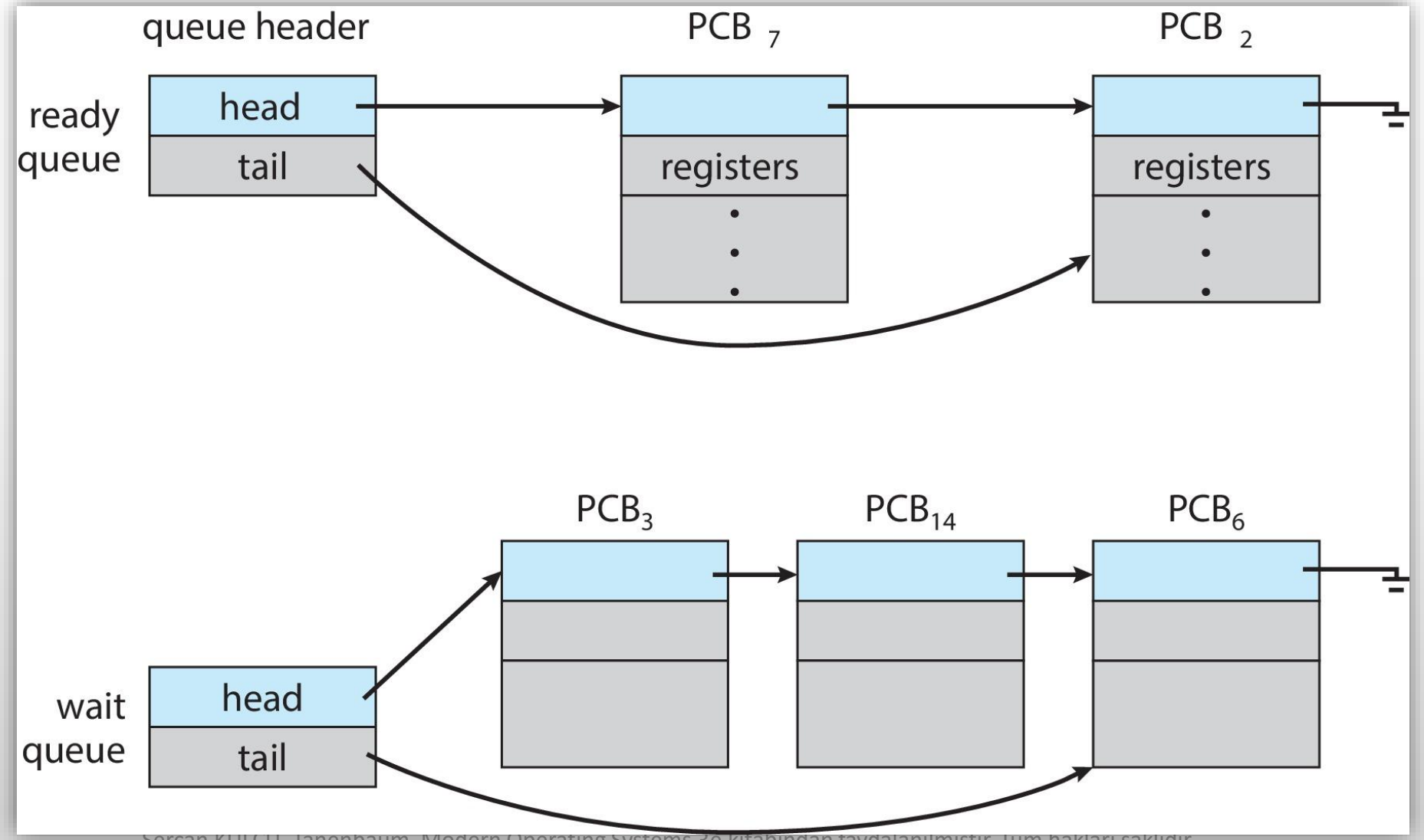


Süreçlerin Çizelgelenmesi

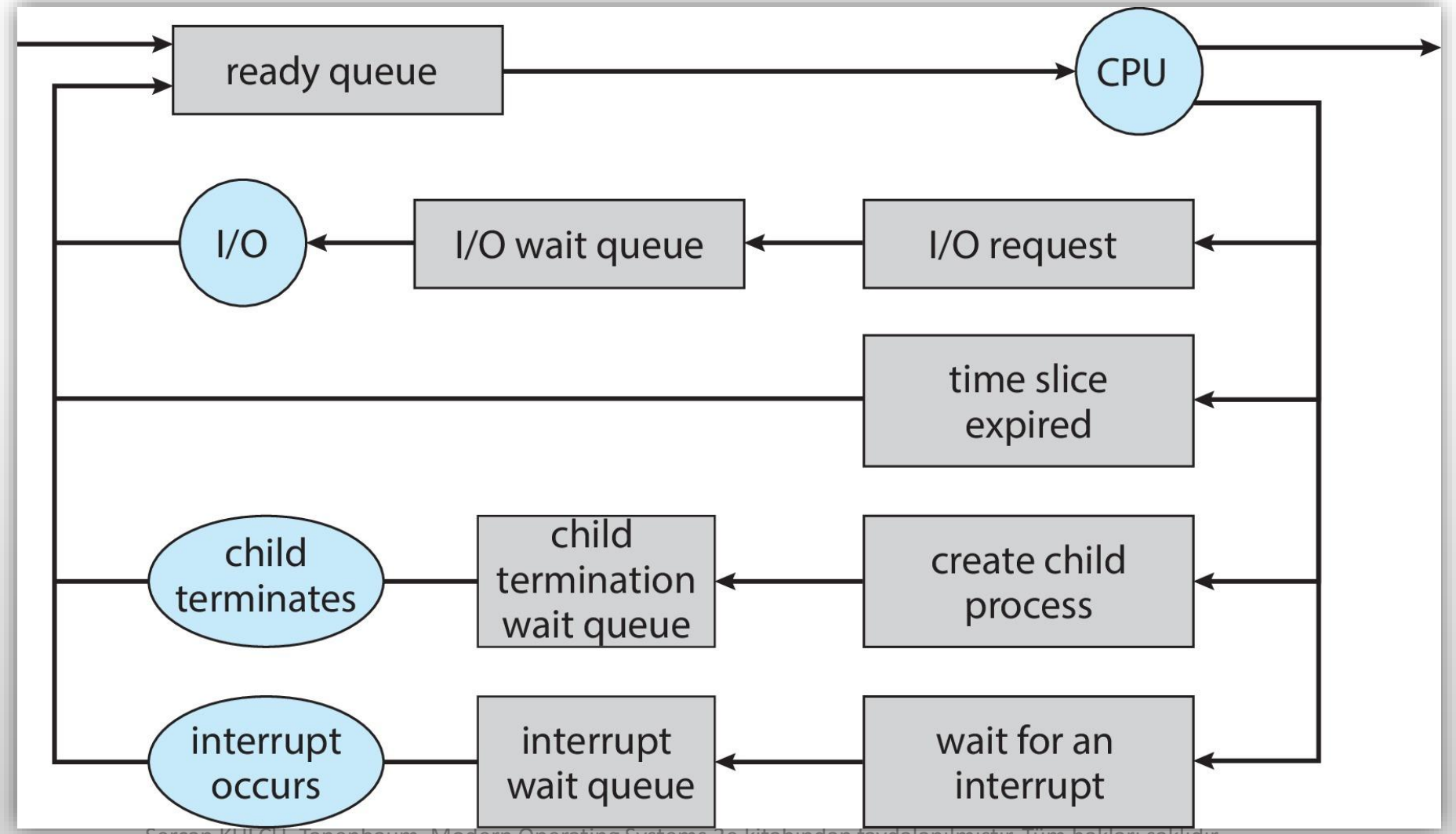
- İşletim sistemi kesme (*interrupt*) ve çizelgelemeden sorumludur.
- Çizelgeleyici, sonraki yürütme için mevcut süreçler arasından seçim yapar.
- **Ready queue**, ana bellekte hazır bulunan ve yürütmeyi bekleyen süreçler.
- **Wait queue**, bir olayı bekleyen süreçler kümesi (G/Ç gibi).



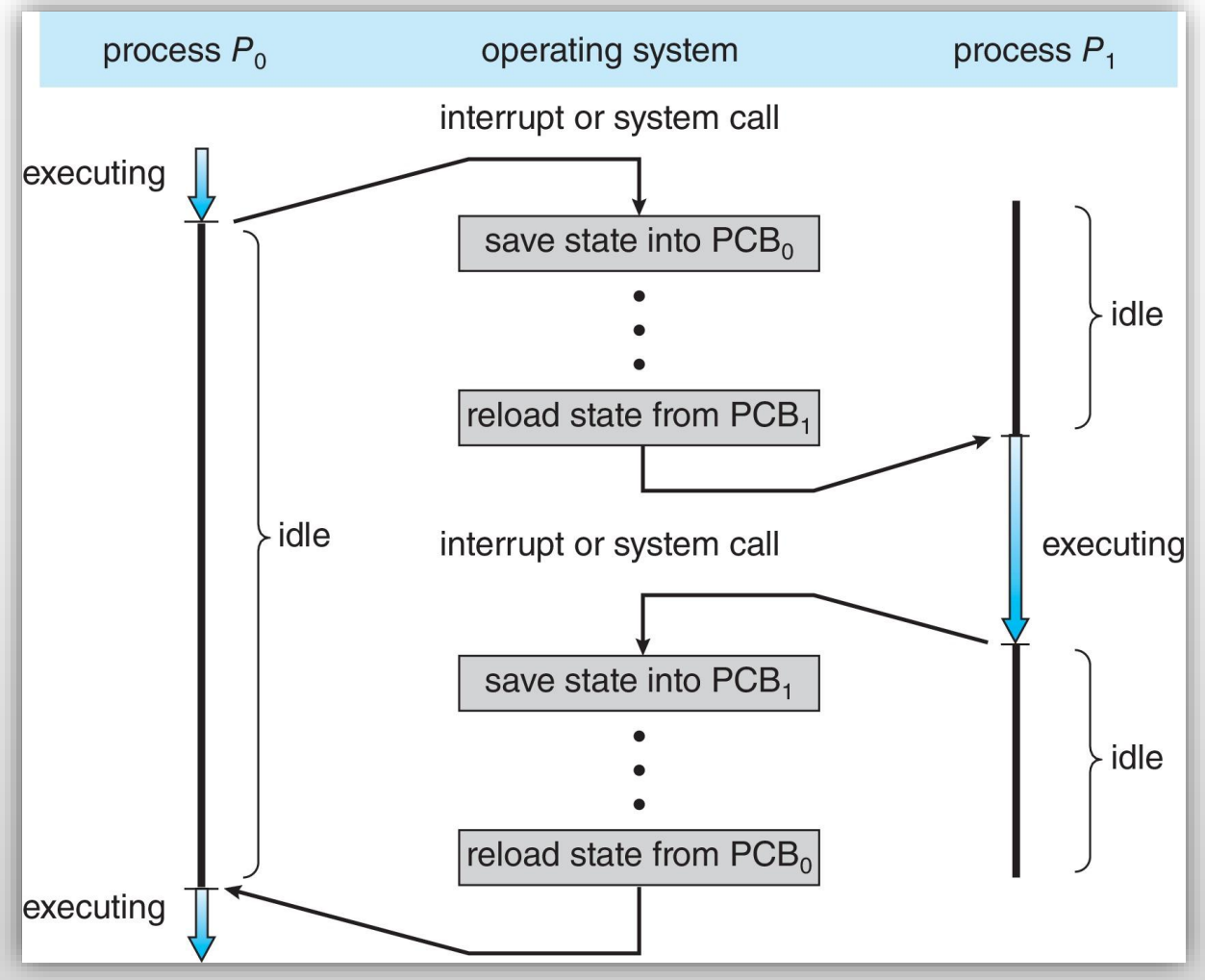
Hazır ve Bekleme Kuyrukları



Süreçlerin Çizelgelenmesi



Bağlam Anahtarlama (Context Switch)



Bağlam Anahtarlama (Context Switch)

- İşlemci başka bir sürece geçtiğinde,
  - eski sürecin durumunu kaydetmeli ve
  - yeni süreç için durum bilgisini yeniden yüklemelidir.
- Bağlam, süreç için süreç tablosunda (*PCB*) tutulan bilgilerdir.
- Bağlam anahtarlama süresi sisteme ek yük (*overhead*) oluşturur.
- Sistem, anahtarlama yaparken başka bir iş yapamaz.
- İşletim sistemi ve PCB ne kadar *karmaşıksa*,
  - bağlam anahtarlama o kadar *uzun* sürer.
- Donanım desteği bu süreyi kısaltabilir. (ekstra yazmaçlar gibi)

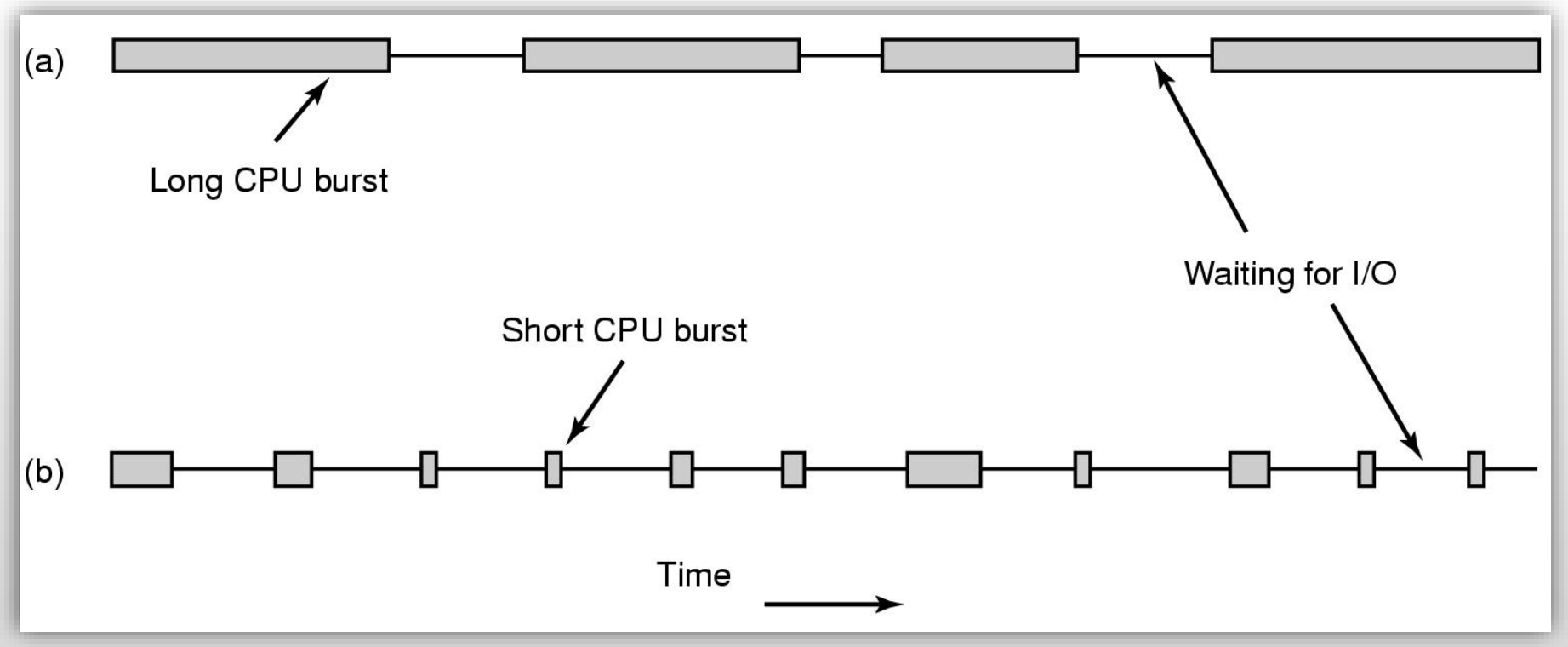
Kesmenin Ele Alınması

Bir kesme oluştuğunda;

1. Donanım, program sayacı vb. verileri kaydeder.
2. Donanım, kesme vektöründen yeni program sayacı yükler.
3. Assembly dili prosedürü, yazmaçları kaydeder.
4. Assembly dili prosedürü, yeni yığın hazırlar.
5. C kesme hizmeti çalışır.
6. Çizelgeleyici hangi sürecin çalıştırılacağına karar verir.
7. C kesme hizmeti, Assembly dili prosedürüne geri döner.
8. Assembly dili prosedürü seçilen süreci başlatır.

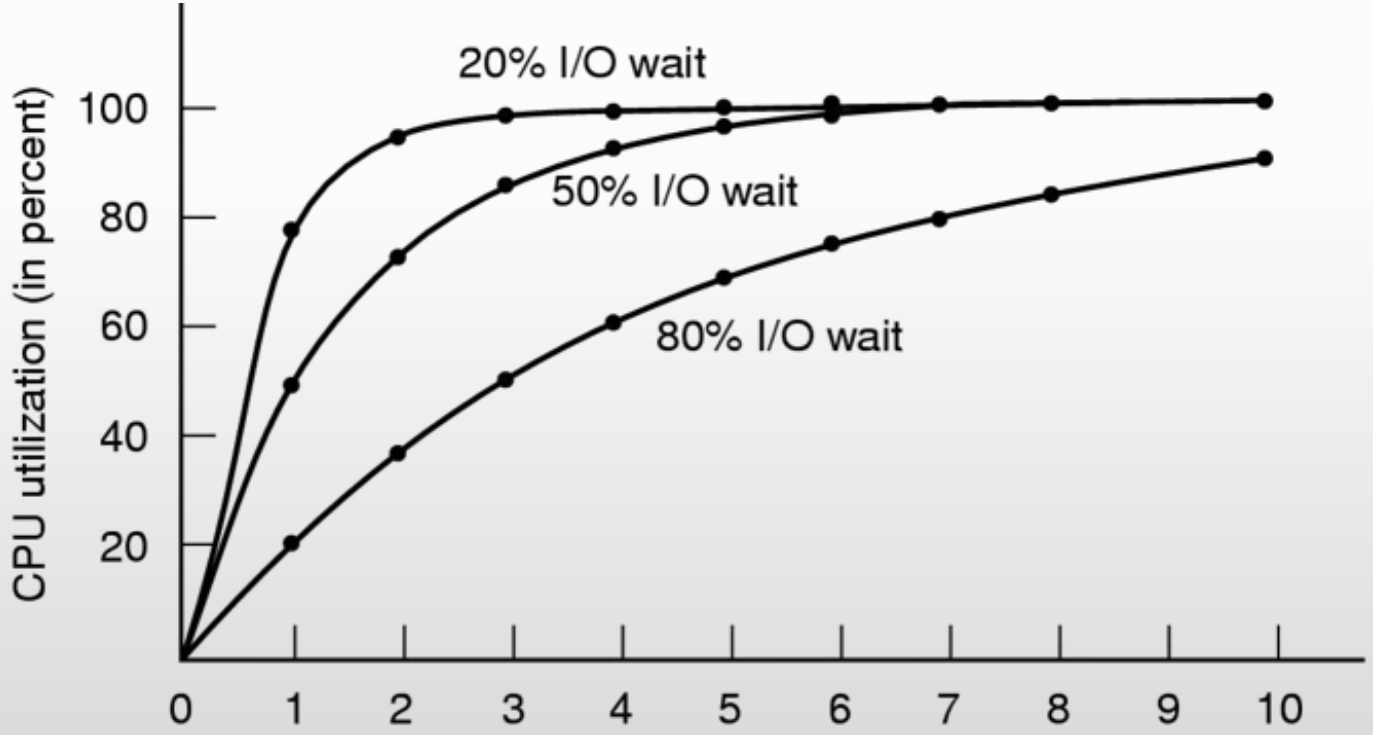
İşlemci Kullanımı

(a) CPU'ya bağlı (*bound*) bir süreç. (b) G/Ç'ye bağlı bir süreç.



Çoklu Programlama Modellemesi

- Bellekte bulunan süreç sayısına bağlı olarak CPU kullanımı grafiği.



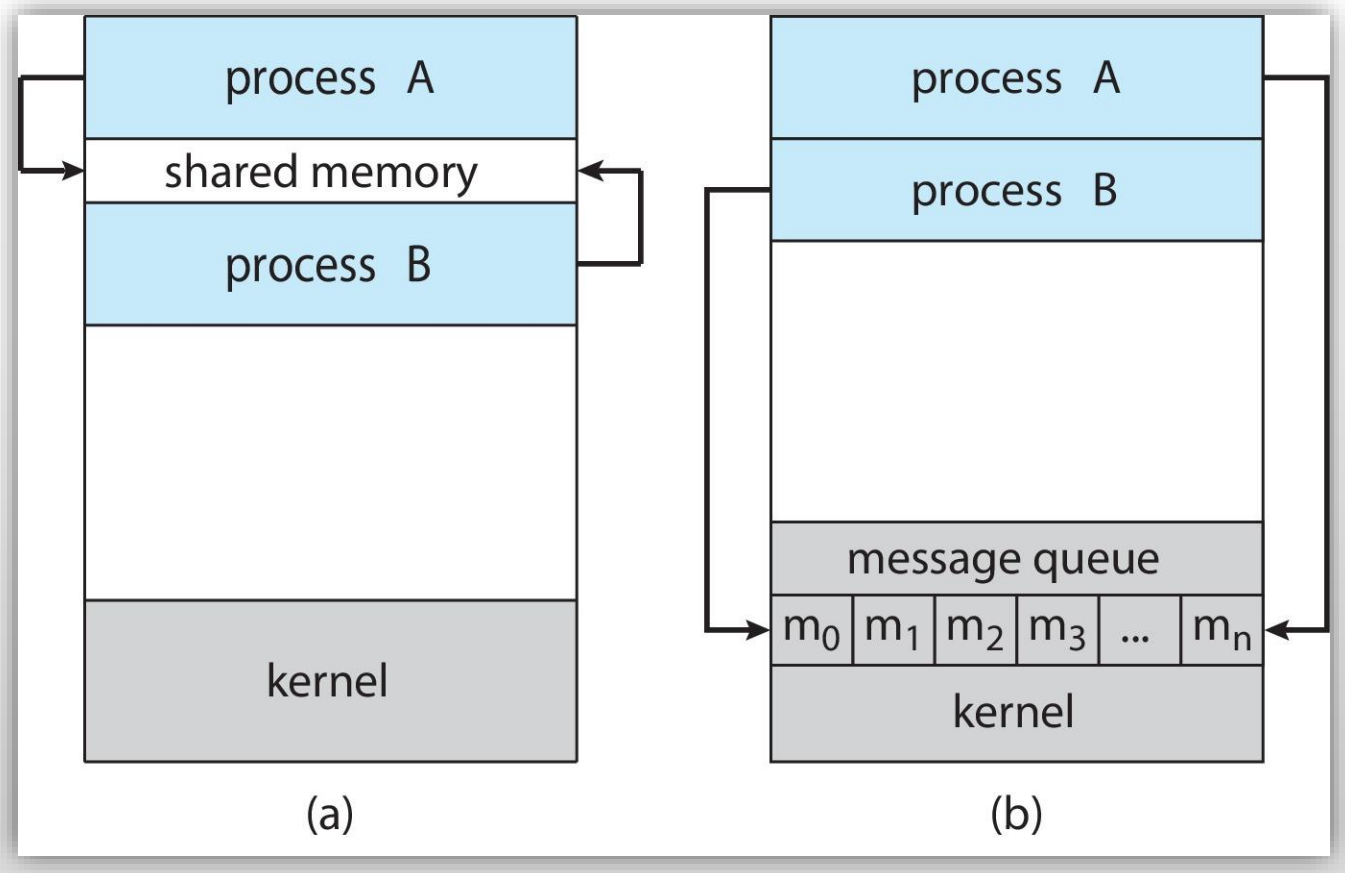
İşlemci Kullanımı

- G/Ç bağlı süreçler, işlemci hızı arttıkça daha belirgin hale gelir.
  - Veri okuma/yazma, ağ iletişimi gibi işlemleri içeren süreçler.
  - İşlemci yoğunluğu düşük, beklemeye eğilimli.
- G/Ç bağlı süreçlerin CPU'ya geçişi hızlı olmalıdır.
  - Bağlam anahtarlamayı hızlı yapmak gerekir.
- İşlemci ve G/Ç bağlı süreçler dengeli bir şekilde yönetilmeli.

Süreçler Arası İletişim Problemleri

- Süreçler bağımsız (*independent*) ya da işbirlikçi (*cooperative*) olabilir.
- İşbirlikçi süreçler birbirleriyle veri paylaşımı yaparlar.
- Süreç çakışmalarıyla nasıl başa çıkılır?
  - aynı koltuk için 2 rezervasyon
- Bağımlılıklar mevcutken doğru sıralama nasıl yapılır?
  - silahı ateşlemeden önce nişan alınması
- İki yöntem var
  - Paylaşımlı bellek (*shared memory*)
  - Mesaj kuyrukları (*message queue*)

Süreçler Arası İletişim



Tekrarlanamaz Yürütme

- *Non-reproducible*

Program 1:

```
repeat
        n = n + 1;
```

Program 2:

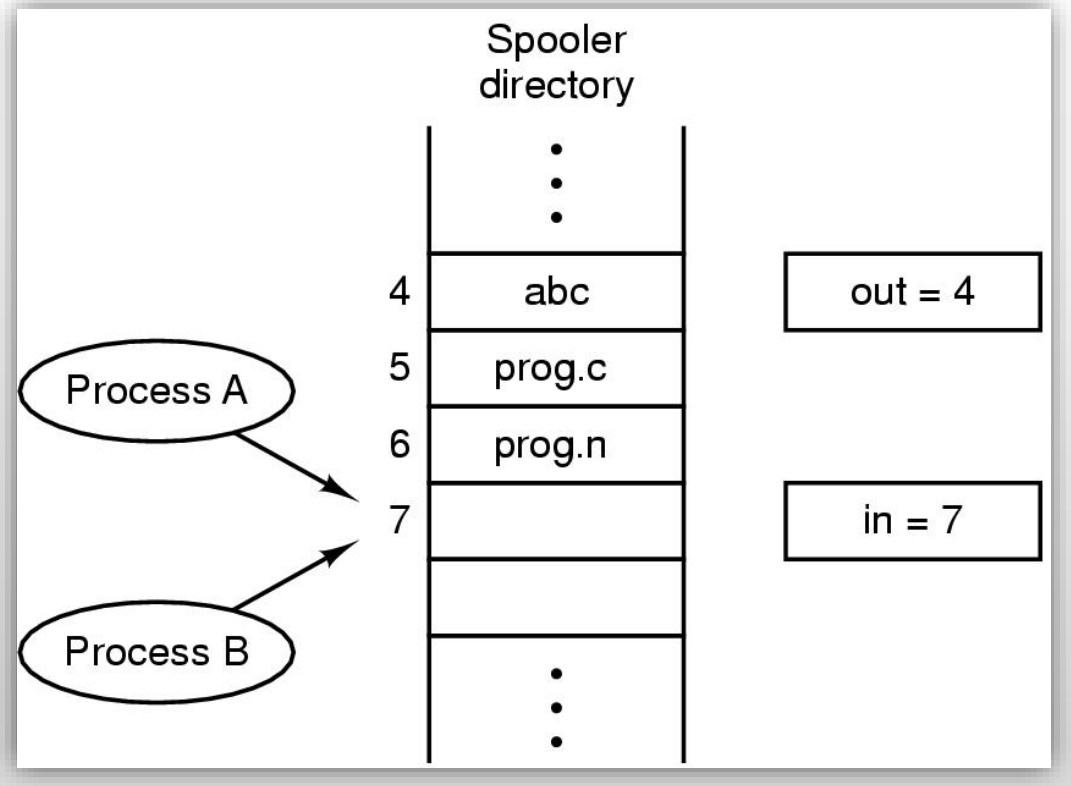
```
repeat
        print(n);
        n = 0;
```

Yürütme sırası farklı olabilir.

- `n = n + 1; print(n); n = 0;`
- `print(n); n = 0; n = n + 1;`
- `print(n); n = n + 1; n = 0;`

Süreçler Arası İletişim

- Yarış durumu: iki süreç aynı bellek alanına aynı anda erişmek isterse.



Yarış Durumu

- İki veya daha fazla süreç,
  - paylaşılan bir veriye erişiyor ise,
  - nihai sonucun hangisinin ne zaman çalıştığına bağlı olmasına denir.
- **Karşılıklı dışlama (*mutual exclusion*)**
  - Birden fazla sürecin aynı anda aynı veriye erişimini engellemek.
- **Kritik bölge (*critical region*)**
  - Programın *paylaşılan alana* erişim yaptığı kod bölümü.

Karşılıklı Dışlama

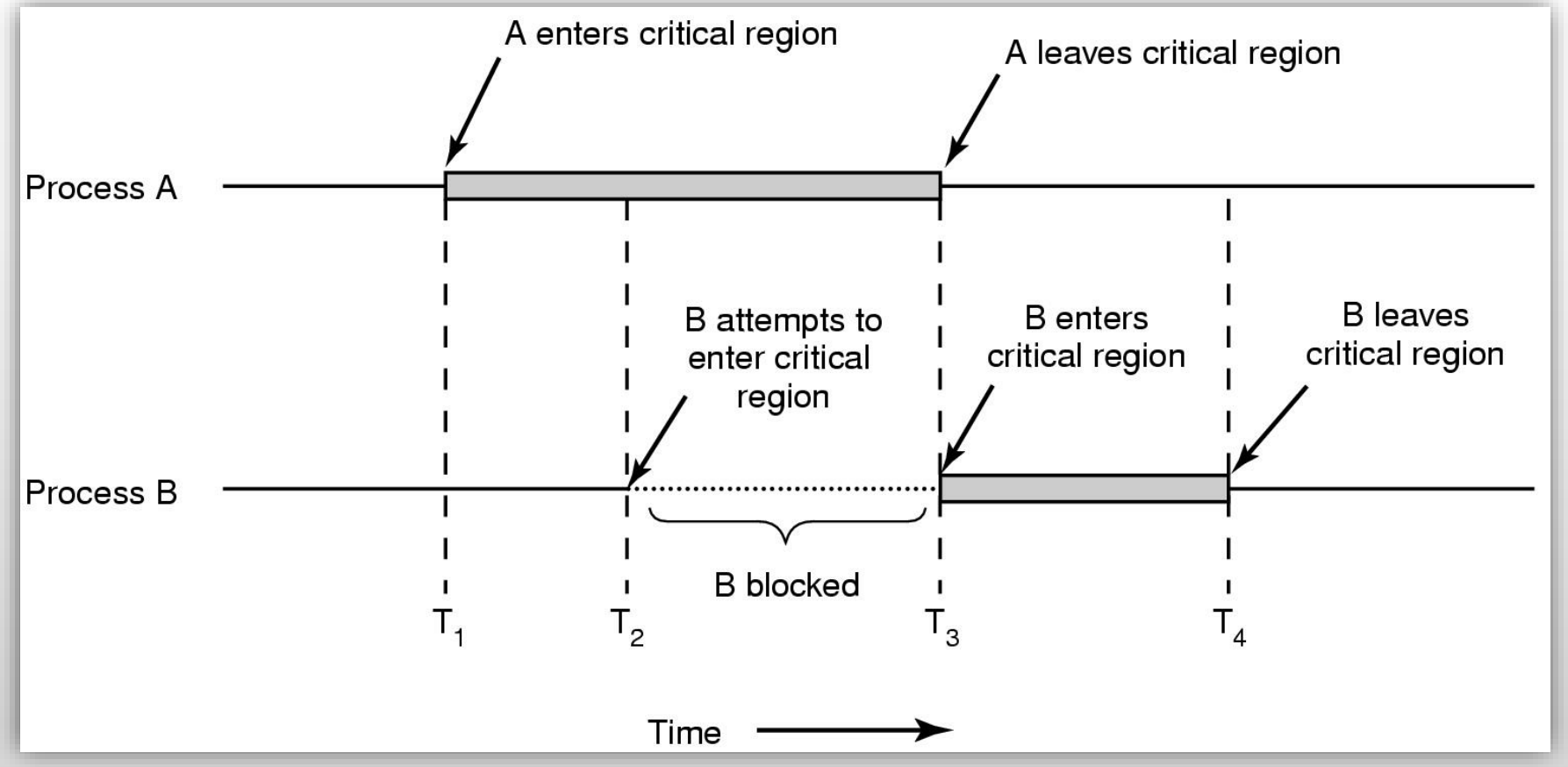
Karşılıklı dışlamayı sağlamak için dört koşul;

- İki süreç *aynı anda* kritik bölgede olmamalı.
- İşlemci hızı ve sayısı hakkında *varsayım* yapılmamalı.
- Kritik bölgenin dışında çalışan süreç, başka bir süreci *engellememeli*.
- Hiçbir süreç kritik bölgeye girmek için *sonsuza* kadar beklememeli.

Kritik Bölge

```
do {

    entry section

        critical section

    exit section

            remainder section

} while (TRUE);
```

Kritik Bölge Kullanarak Karşılıklı Dışlama



Peterson’un Çözümü

- Hangi sürecin kritik bölgeye gireceğini belirtmek için "*turn*" ve "*flag[2]*" olmak üzere iki değişken kullanır.
  - *flag*, süreç tarafından kritik bölgeye girme niyetini belirtir.
  - *turn*, kritik bölgeye girecek bir sonraki süreci belirtir.
- Her iki sürecin de kritik bölgeye aynı anda girmesini önlemek için bir meşgul bekleme (*busy wait*) döngüsü ve bir dizi koşul kullanır.
- Karşılıklı dışlamayı sağlar.
- Süreçlerin sonsuz bir bekleme döngüsüne girmesini engeller.
- Meşgul bekleme döngüsü yüksek miktarda CPU zamanı tüketebilir!

Peterson'un Çözümü

```
int N = 2; // Number of threads
boolean[] flag = new boolean[N];
int turn = 0;
int counter;
void incrementCounter() {
     int i = (int) (Thread.currentThread().getId() % N);
     int j = (i + 1) % N;
     flag[i] = true;      turn = j;
     while (flag[j] && turn == j) {} // Spin loop
     counter++; // Critical region
     System.out.println("Counter:" + counter + "i:" + i + "j:" + j);
     flag[i] = false;
}
```

Uyuma ve Uyanma

- Meşgul beklemenin dezavantajları;
  - Düşük öncelikli süreç kritik bölgede iken, yüksek öncelikli süreç gelirse, düşük öncelikli sürecin çalışmasını engeller.
  - Kilit'ten (*lock*) dolayı meşgul beklemede CPU'yu boşa harcar.
  - Daha düşük öncelikli süreç kritik bölge dışına çıkamaz.
  - Ölümcül kilitlenme (*deadlock*) oluşur.
- Meşgul beklemek yerine bloke etmek
  - Önce uyandır, sonra uyut (*wake up, sleep*)

Üretici Tüketici Problemi

- İki süreç, sabit boyutlu bir arabelleği paylaşıyor olsun.
  - Üretici arabelleğe veri yazar.
  - Tüketici arabellekten veri okur.
- **Sınırsız** tampon bellek var ise;
  - Üretici beklemez, tüketici tampon boş ise bekler.
- **Sınırlı** tampon bellek var ise;
  - Tampon dolu ise üretici bekler.
  - Tampon boş ise tüketici bekler.

Ölümcül Yarış Durumu - Producer

```
int N = 100; /* number of slots in the buffer */
int count = 0; /* number of items in the buffer */
void producer() {
  while (true) { /* repeat forever */
    item = produce_item(); /* generate next item */
    if (count == N) sleep(); /* if buffer is full, go to sleep */
    insert_item(item); /* put item in buffer */
    count = count + 1; /* increment count of items in buffer */
    if (count == 1) wakeup(consumer); /* was buffer empty? */
  }
}
```

Ölümcül Yarış Durumu - Consumer

```
void consumer() {
  while (true) { /* repeat forever */
    if (count == 0) sleep(); /* if buffer empty, sleep */
    item = remove_item(); /* take item out of buffer */
    count = count - 1; /* decrement count of items in buffer */
    if (count == N-1) wakeup (producer); /*was buffer full?*/
    consume_item(item); /* print item */
  }
}
```

Mesaj Gönderme/Alma

- İki süreç haberleşmek için iletişim bağı kurmalı.
- **Fiziksel**
  - Paylaşımlı bellek.
  - Donanım veriyolu.
  - Ağ.
- **Mantıksal**
  - Doğrudan, dolaylı.
  - Senkron, asenkron.
  - Otomatik, tamponlayarak.

Doğrudan İletişim

- Süreçler açık olarak adlarını belirtirler.
  - `send (P, message)` P sürecine mesaj gönder.
  - `receive (Q, message)` Q sürecinden mesaj al.
- Bağlantı otomatik kurulur.
- Bir bağlantı bir çift (*pair*) süreçle ilişkilidir.
- Her bir çift arasında sadece bir bağlantı vardır.
- Bağlantı *çift* yönlüdür.

Doğrudan İletişim

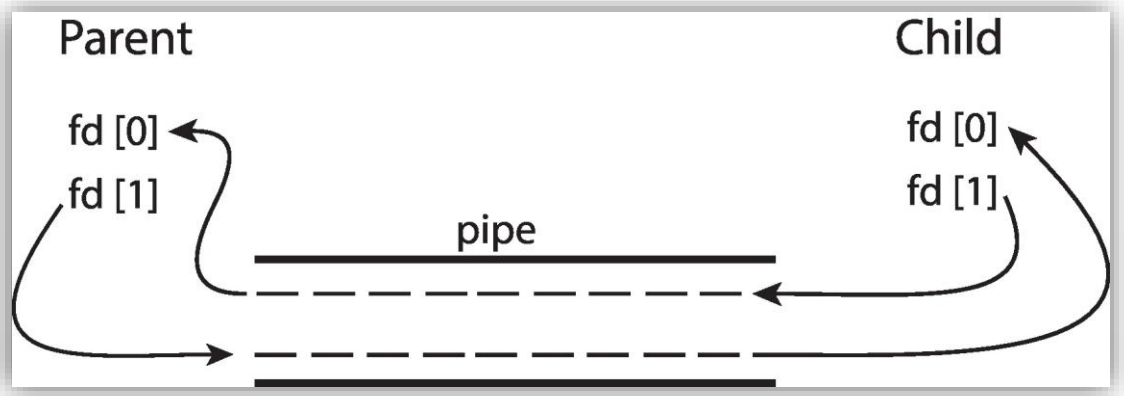
- Posta kutusu aracılığıyla iletişim sağlanır.
  - Her posta kutusu tekil tanımlayıcıya sahiptir.
  - Süreçler aynı posta kutusunu paylaşıyorsa haberleşebilir.
  - `send (A, message)` A posta kutusuna mesaj gönder.
  - `receive (A, message)` A posta kutusundan mesaj al.
- Bir bağlantı bir çok süreç ile ilişkilendirilebilir.
- Her bir süreç çifti bir çok bağlantı paylaşabilir.
- Bağlantılar tek yönlü ve çift yönlü olabilir.

Boru Hattı (pipe)

- İki sürecin iletişim kurmasını sağlayan bir kanal.
- Sorunlar:
  - İletişim tek yönlü mü, çift yönlü mü? (*simplex, duplex*)
  - Çift yönlü iletişim durumunda, yarı mı, tam mı? (*half, full*)
  - Süreçler arasında bir ilişki (ata-çocuk) var mı?
  - Bir ağ üzerinden kullanılabilir mi?
- Türleri;
  - **Sıradan**, Süreç dışından erişilemez. Ata süreç tarafından çocuk süreç ile haberleşmek için oluşturulur.
  - **Adlandırılmış**, Tüm süreçler tarafından erişilebilir.

Boru Hattı (pipe)

- **Sıradan boru hattı**
  - Üretici bir uca yazar.
  - Tüketici diğer uçtan okur.
  - Tek yönlü.
  - Ata-çocuk ilişkisi.
  - Adsız boru hattı olarak da geçer.
- **Adlandırılmış boru hattı**
  - Çift yönlü iletişim.
  - Ata-çocuk ilişkisi gerektirmez.
  - Windows, UNIX destekler.



Pipe Örneği

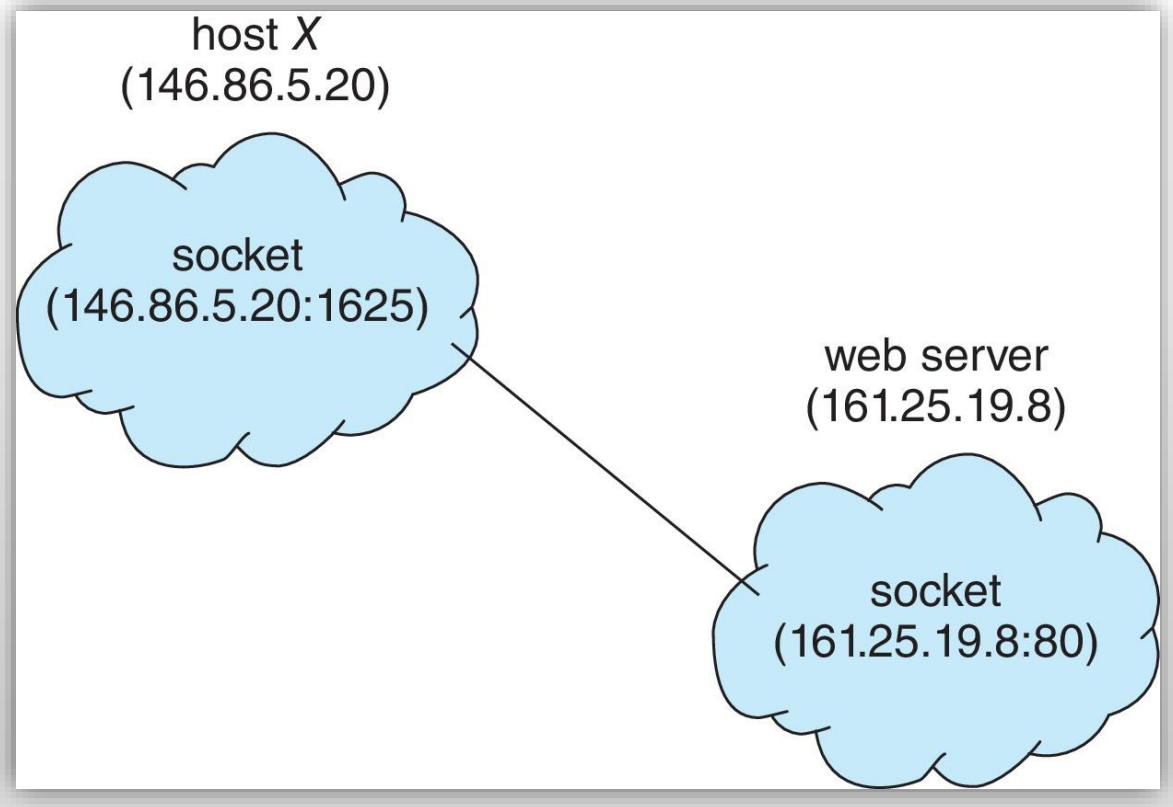
```
int pipefd[2];    // Pipe için file descriptor'lar
if (pipe(pipefd) == -1) // Pipe oluştur
  exit(EXIT_FAILURE);
pid_t pid = fork(); // Yeni bir süreç oluştur
if (pid == -1)  exit(EXIT_FAILURE);
if (pid == 0) {  // Çocuk süreç
  read(pipefd[0], buffer, sizeof(buffer));
  close(pipefd[0]);  // Okuma tarafını kapat
} else {          // Ata süreç
  write(pipefd[1], message, sizeof(message));
  close(pipefd[1]);  // Yazma tarafını kapat
}
```

Soketler

- Soket, iletişim için uç nokta (*end point*) olarak tanımlanır.
- IP adresi ve bağlantı noktasının birleştirilmesi (*IP* + *port*).
- **Port**: ağ hizmetlerini ayırt etmek için kullanılır.
- *161.25.19.8:1625*, **IP**:161.25.19.8 **port**:1625
- İletişim bir çift soket arasında oluşur.
- 1024'ün altındaki tüm bağlantı noktaları standart hizmetler için kullanılır.
- 127.0.0.1 geri döngü (*loopback*) adresi.

Soketler

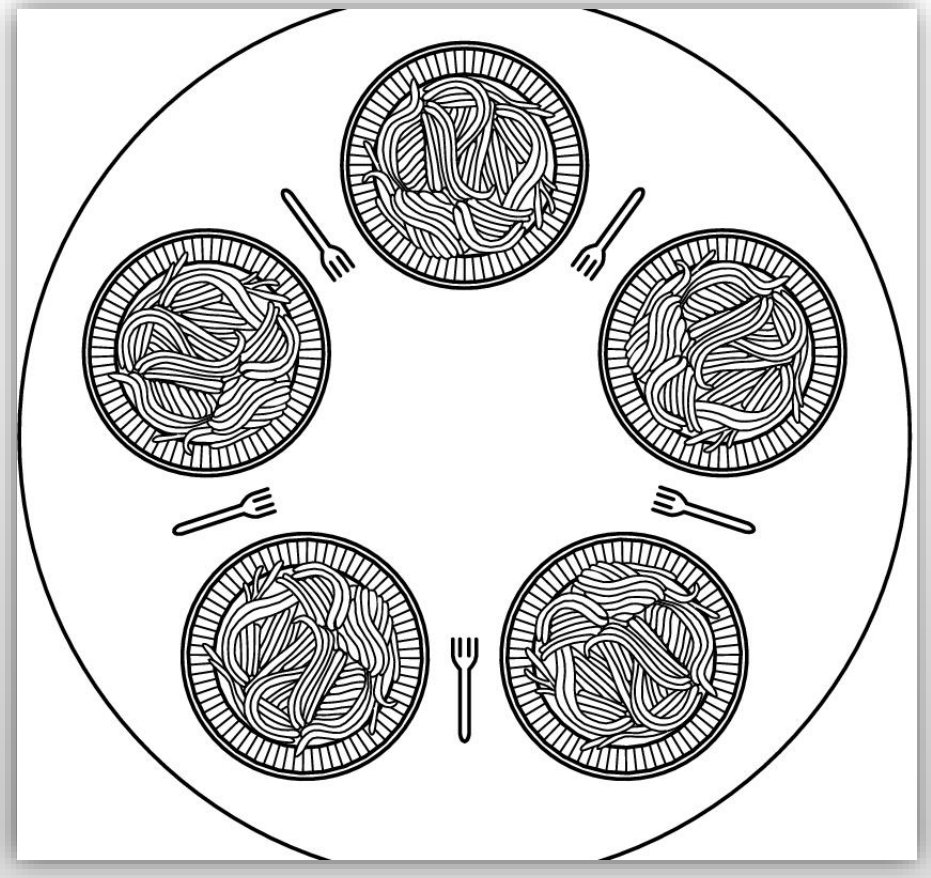
- .



Süreçler Arası İletişim Problemleri

- **Dining Philosopher**
  - Bir filozof ya yer ya da düşünür.
  - Aç kalırsa, iki çatal alıp yemeye çalışır.
- **Okur-Yazar**
  - Veritabanına erişimi modeller.

Dining Philosophers Problemi



Dining Philosophers

```
while(true) {
  think(); // Initially, thinking
  // Take a break from thinking, hungry now
  pick_up_left_fork();
  pick_up_right_fork();
  eat();
  put_down_right_fork();
  put_down_left_fork();
  // Not hungry anymore. Back to thinking!
}
```

Dining Philosophers - loop

```
#define LEFT (i+N-1) % N /* i's left neighbor */
#define RIGHT (i+1) % N  /* i's right neighbor */

void philosopher(int i) {
    while (TRUE) { /* repeat forever */
        think(); /* philosopher is thinking */
        take_forks(i); /* acquire two forks or block */
        eat(); /* philosopher is eating */
        put_forks(i); /* put both forks back on table */
    }
}
```

Dining Philosophers – take forks

```
void take_forks(int i) {
  down(&mutex); /* enter critical region */
  state[i] = HUNGRY; /* philosopher i is hungry */
  test(i); /* try to acquire 2 forks */
  up(&mutex); /* exit critical region */
  down(&s[i]); /* block if forks were not acquired */
}
```

Dining Philosophers – put forks

```
void put_forks (i) {
  down(&mutex); /* enter critical region */
  state[i]= THINKING; /* philosopher has finished eating */
  test(LEFT); /* see if left neighbor can now eat */
  test(RIGHT); /* see if right neighbor can now eat */
  up(&mutex); /* exit critical region */
}
```

Dining Philosophers – test state

```
void test(i) {
  if (state[i] == HUNGRY &&
      state[LEFT] != EATING &&
      state[RIGHT] != EATING) {
    state[i]= EATING;
    up(&s[i]);
    }
}
```

Okur-Yazar Problemi - writer

```
semaphore mutex = 1; /* controls access to 'rc' */
semaphore db = 1; /* controls access to the database */
int rc = 0; /* # of processes reading or wanting to */
void writer(void) {
    while (TRUE) { /* repeat forever */
      think_up_data(); /* noncritical region */
      down(&db); /* get exclusive access */
      write_data_base(); /* update the data */
      up(&db); /* release exclusive access */
  }
}
```

Okur-Yazar Problemi - reader

```
void reader(void) {
  while (TRUE) { /* repeat forever */
    down(&mutex); /* get exclusive access to 'rc' */
    rc = rc + 1; /* one reader more now */
    if (rc == 1) down(&db); /* if this is the first reader ... */
    up(&mutex); /* release exclusive access to 'rc' */
    read_data_base(); /* access the data */
    down(&mutex); /* get exclusive access to 'rc' */
    rc = rc - 1; /* one reader fewer now */
    if (rc == 0) up(&db); /* if this is the last reader ... */
    up(&mutex); /* release exclusive access to 'rc' */
    use_data_read(); /* noncritical region */
  }
}
```

Okur-Yazar Problemi

- Çözümün dezavantajı nedir?
  - Yazar süreci açlık (*starvation*) tehlikesiyle karşı karşıya.

Mobil Sistemlerde Çoklu Görev

- İlk versiyonlarda tek süreç çalışabiliyordu.
- Kullanıcı arayüzü kısıtlarından dolayı
  - Ön planda tek bir süreç çalışabilir.
  - Arka planda bir çok süreç çalışabilir.
  - Ses oynatma gibi özelliklerde kısıtlamalar olabilir.
- Android süreçlerin kullanımı için hizmet (*service*) arayüzüne sahiptir.
  - Hizmet, süreç askıda (*suspend*) olsa bile çalışmaya devam edebilir.

Android Süreç Öncelikleri

- Yüksekten düşüğe göre;
  - Ön planda çalışan süreç (*foreground*),
  - Görünür süreç (*visible*),
  - Hizmet süreci (*service*),
  - Arka planda çalışan süreçler (*background*),
  - Boş süreçler (*empty*).

SON