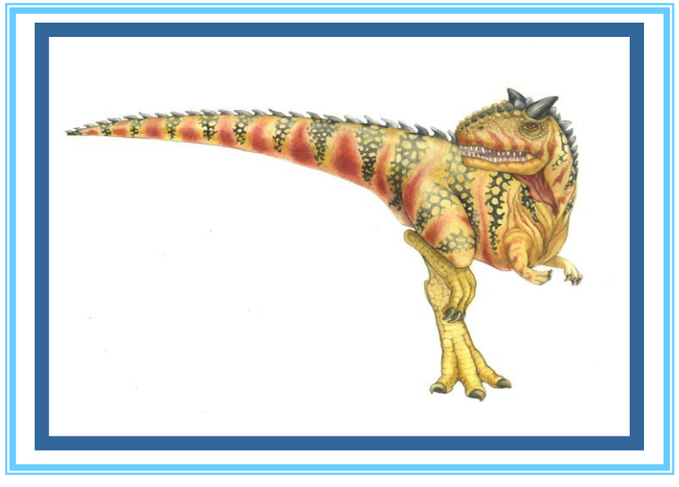
Bölüm 3: Prosesler



Bölüm 3: Prosesler

- Proses Kavramı
- Proses Sıralama (Scheduling)
- Proses Üzerinde İşlemler
- Prosesler Arası İletişim
- Mesaj İletimi ve Paylaşımlı Bellek Sistemlerinde IPC
- IPC (Prosesler Arası İletişim) Örnekleri
- İstemci-Sunucu Sistemleri Arasındaki İletişim

Hedefler

- Bir prosesin ayrı bileşenlerini tanımlamak ve bunların bir işletim sisteminde nasıl temsil edildiğini ve sıralandığını görmek
- Uygun sistem çağrıları kullanarak programlar geliştirmek de dahil olmak üzere, bir işletim sisteminde proseslerin nasıl oluşturulduğunu ve sonlandırıldığını tanımlamak
- Paylaşılan bellek ve mesaj iletimini kullanarak prosesler arası iletişimi tanımlamak ve karşılaştırmak
- Prosesler arası iletişimi gerçekleştirmek için boruhatları (pipes) ve POSIX paylaşılan belleği kullanan programlar tasarlamak
- Soketler ve uzak yordam çağrıları kullanarak istemci-sunucu iletişimini tanımlamak
- Linux işletim sistemiyle etkileşime giren çekirdek modülleri tasarlamak

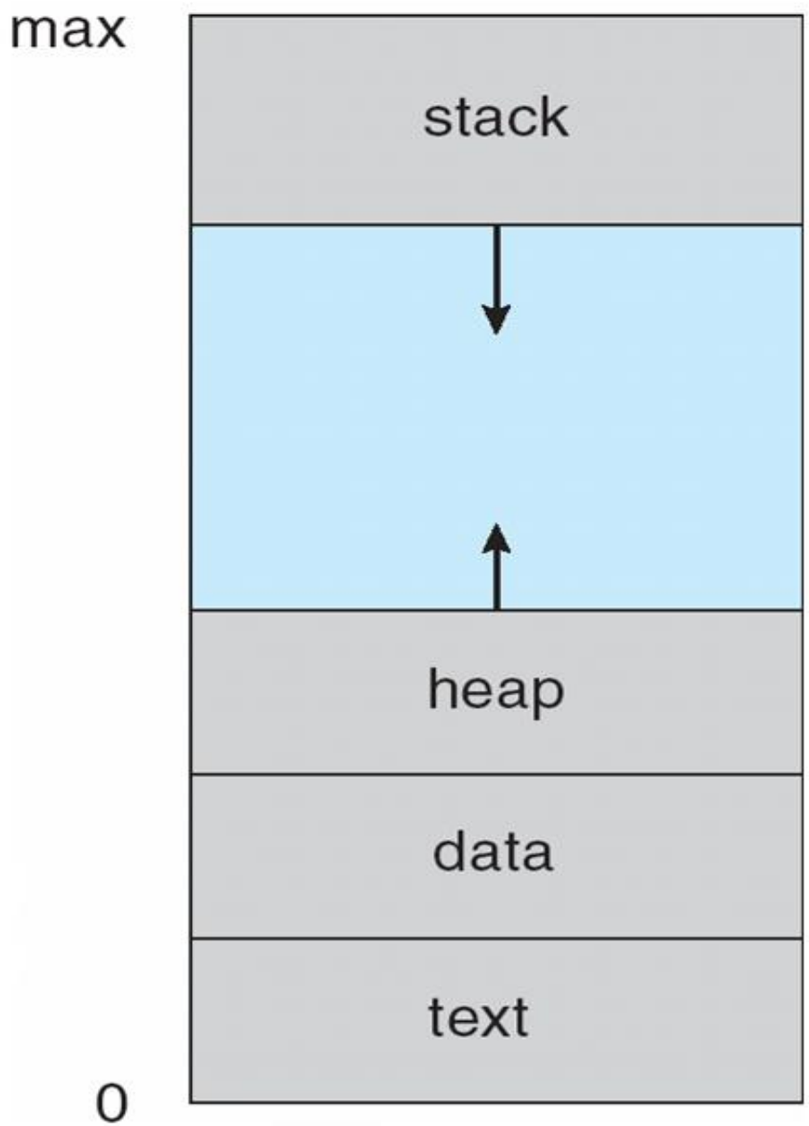
Proses Kavramı

- Bir işletim sistemi programları proses halinde yürütür:
- Ders kitaplarında **görev / işlem / süreç / iş (job) / proses** terimleri birbirlerinin yerine kullanılır.
- **Proses**, yürütülen bir programdır ve prosesler sıralı bir biçimde yürütülmelidir.
- Bir proses aşağıdakileri alanları içerir:
  - Program kodu, metin alanı (**text section**)
  - Mevcut aktiviteyi içeren **program sayacı**, proses kaydedicileri
  - Geçici veriyi tutan yığın (**stack**)
    - Fonksiyon parametreleri, döndürülen adresler, yerel değişkenler
  - Veri bölümü (**data section**) global değişkenleri tutar
  - Çalışma anında dinamik olarak tahsis edilen bellek kısmı yığıt (**heap**)

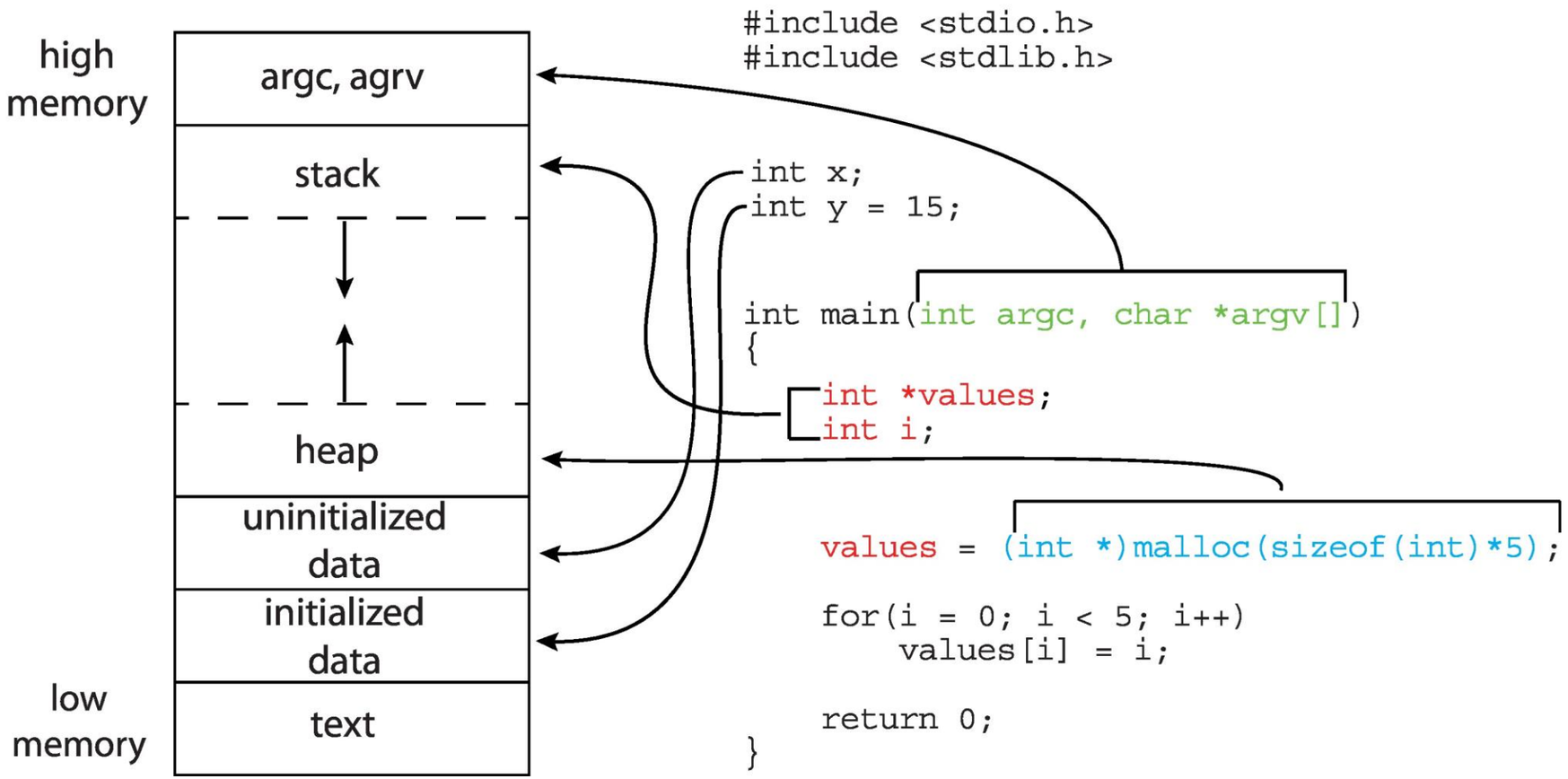
Proses

- Program çalıştırılabilir bir dosya halinde harddiskte tutulan **pasif** bir varlıktır, proses ise **aktiftir**
  - Program çalıştırılabilir halde belleğe yüklendiğinde proses halini alır.
- Programın çalıştırılması komut satırından komutun girilmesi, grafik arayüzde program ikonu üzerine tıklanması vb. ile başlar.
- Bir program birden fazla proses içerebilir.
  - Örneğin birden fazla kullanıcının aynı programı çalıştırması.
    - Derleyici
    - Metin editörü

Bellekteki Bir Proses



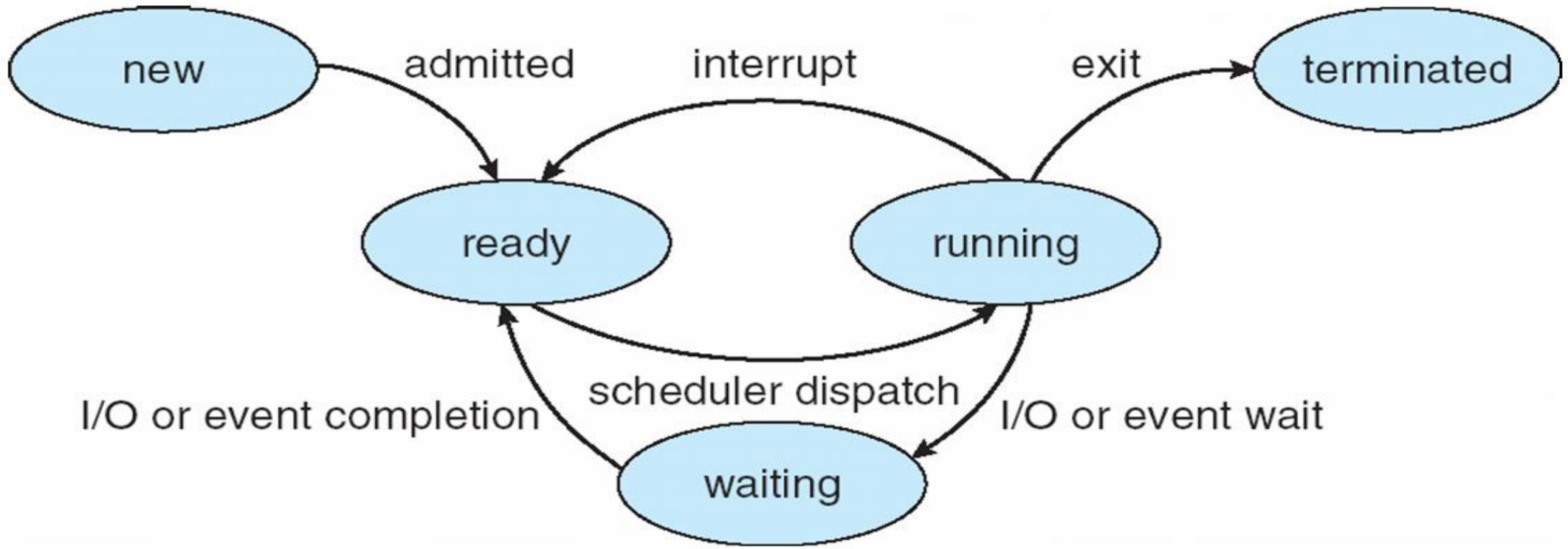
Bir C Programının Bellek Yerleşimi



Proses Durumları

- Çalışma anında proseslerin **durumları** değişir.
  - **Yeni**: Yeni bir proses oluşturuluyor.
  - **Çalışıyor**: İşlemler gerçekleştiriliyor.
  - **Beklemede**: Proses bazı olayların gerçekleşmesini beklemektedir.
  - **Hazır**: Proses, işlemciye aktarılmayı beklemektedir.
  - **Sonlandırılmış**: Prosesin yürütülmesi tamamlandığını belirtir.

Proses Durum Diyagramı

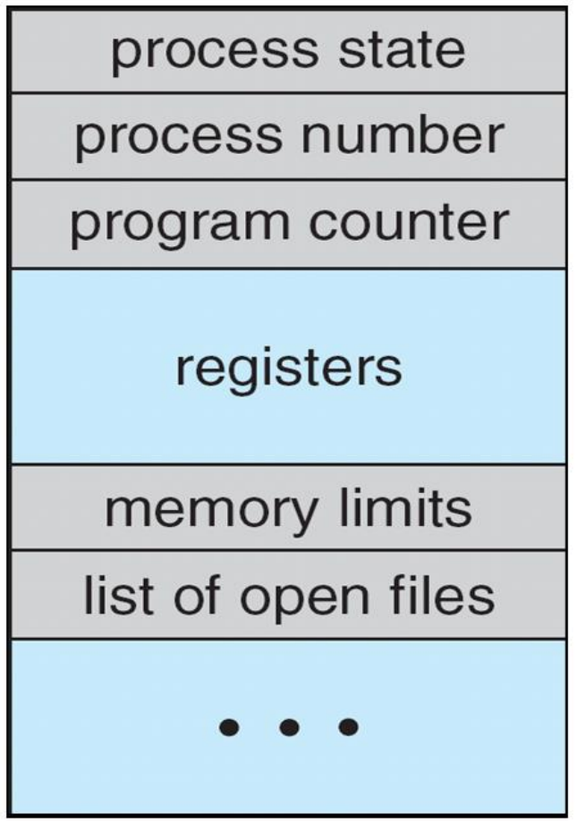


Proses Kontrol Bloğu (PCB)

Herbir proses ile ilişkili bilgi (**görev kontrol bloğu** olarak ta adlandırılır)

Şu bilgileri içerir :

- Proses durumu – çalışıyor, beklemede
- Program Sayacı – bir sonraki çalışacak talimatın konumu
- CPU kaydedicileri – prosese ait tüm kaydedicilerin içerikleri
- CPU sıralama bilgisi – öncelikler, sıralama kuyruk pointerları
- Bellek yönetim bilgisi – prosese tahsis edilen bellek
- Hesap bilgisi – kullanılan CPU, başlangıçtan beri geçen zaman
- I/O durum bilgisi – prosese tahsis edilen I/O aygıtları, açık dosya listesi



İş Parçacıkları (Threads)

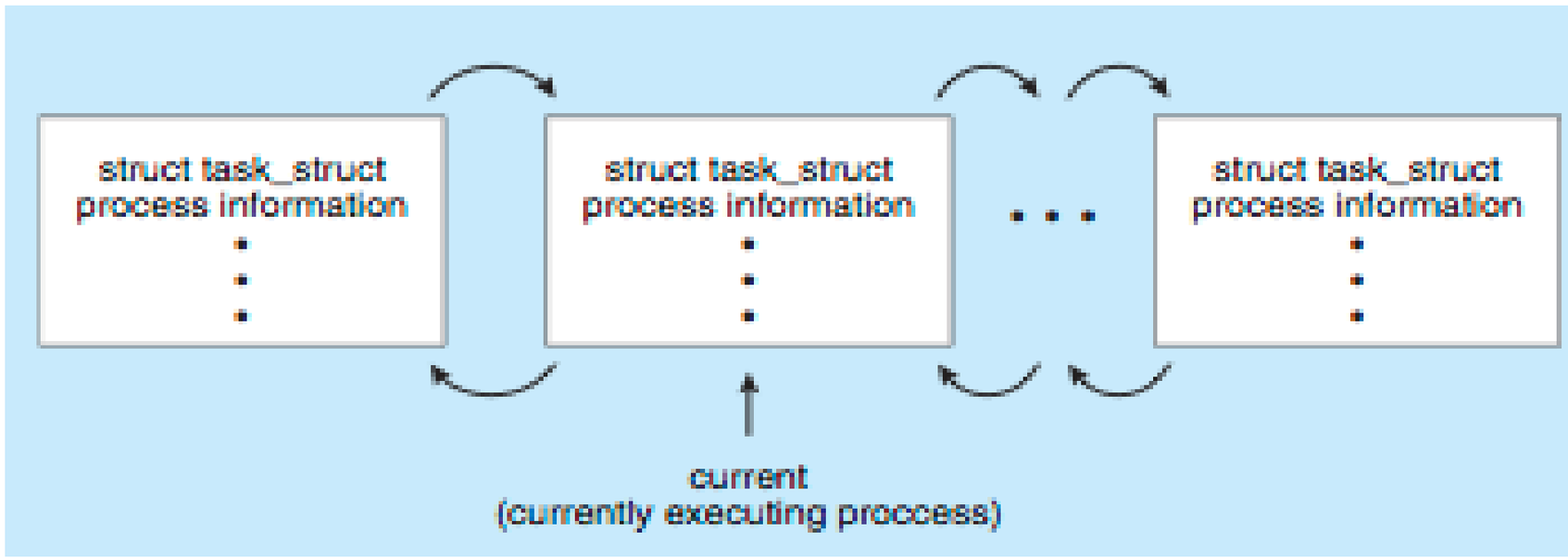
- Şimdiye kadar, proses tek bir yürütülen iş parçacığına sahip
- Proses başına birden çok program sayacına sahip olduğumuzu düşünelim.
  - Birden fazla konum aynı anda yürütülebilir
    - Birden çok kontrol akışı-> **iş parçacıkları**
- Daha sonra iş parçacığı detayları için depolama alanı olmalı, PCB'de çoklu program sayaçları gibi

- 4. Bölümde ayrıntılı olarak ele alınacak

Proseslerin Linux'ta Temsili

- `task_struct` C yapısı içeriği:

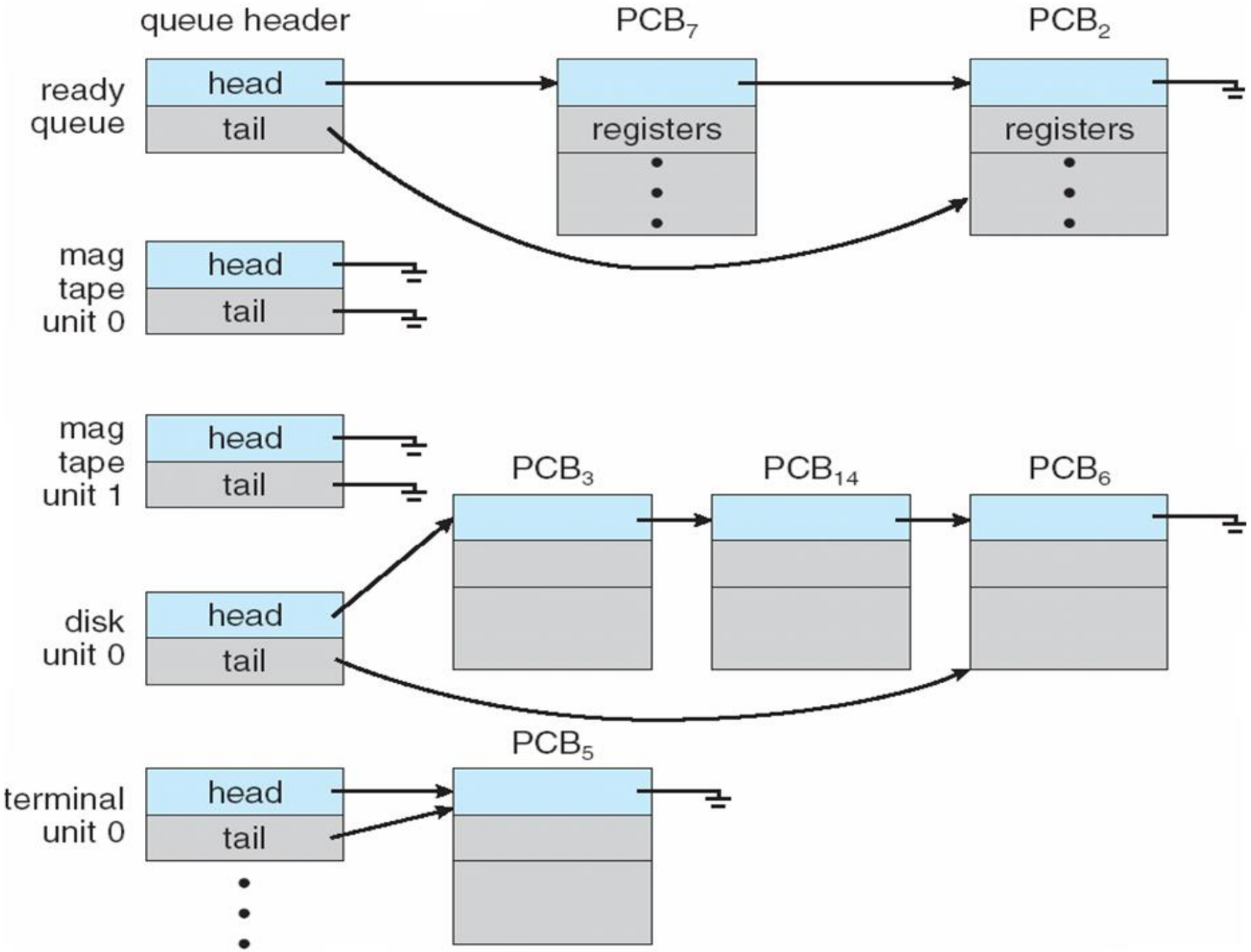
```
pid t pid; /* Proses tanımlayıcı*/
long state; /* Prosesin durumu*/
unsigned int time slice /* sıralama bilgisi*/
struct task struct *parent; /* bu prosesin ebeveyni */
struct list head children; /* bu prosesin çocukları */
struct files struct *files; /* açık dosyaların listesi*/
struct mm struct *mm; /* bu prosesin adres alanı */
```



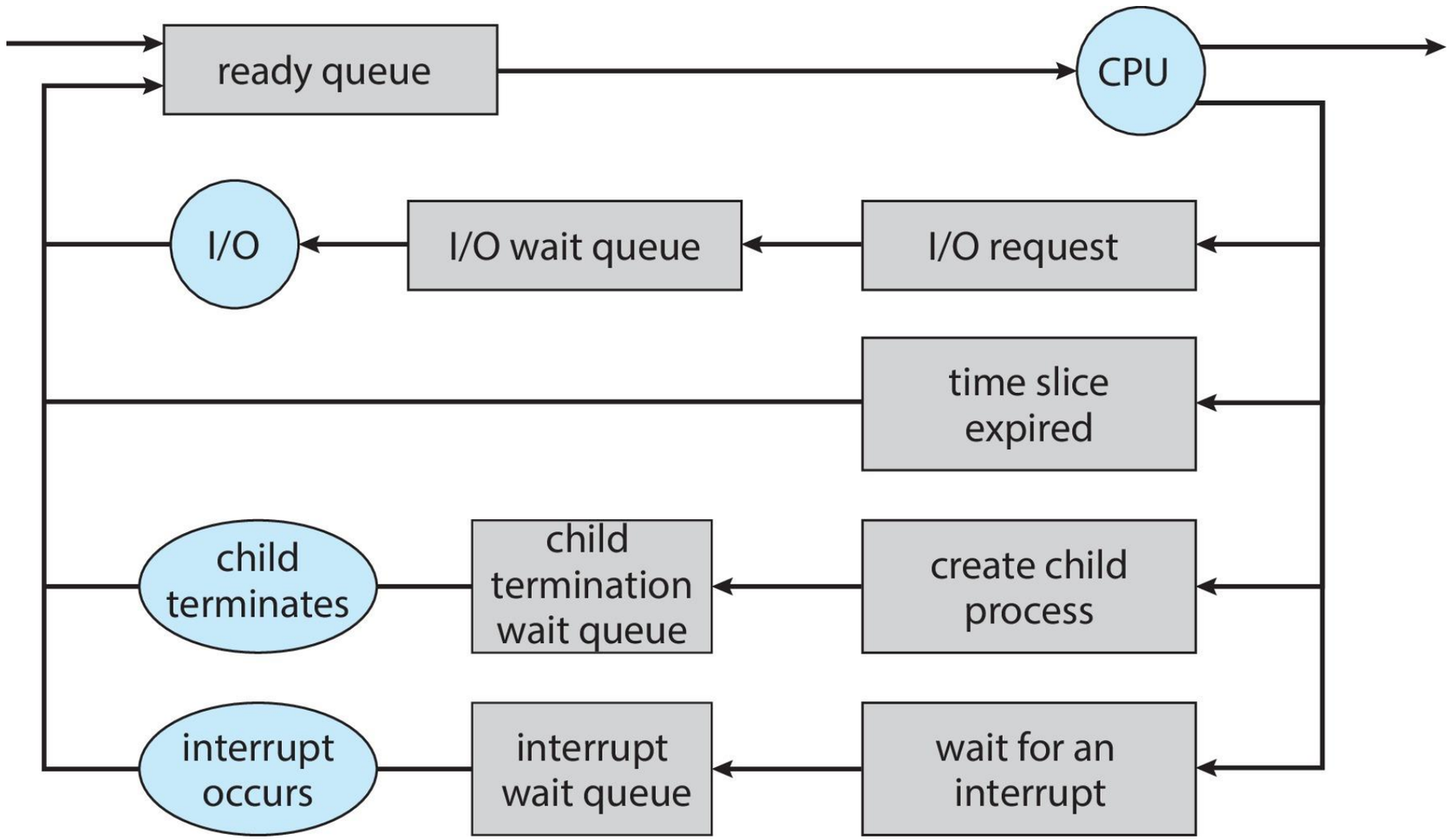
Proses Sıralama

- **Proses sıralayıcı (scheduler),** işleme hazır prosesler arasından CPU çekirdeği üzerinde işlenecek prosesi seçer.
- Hedef - İşlemciyi azami kullan, prosesler arası geçiş ve zaman paylaşımını çok hızla gerçekleştir
- Prosesler aşağıdaki **sıralama kuyruklarında** tutulur.
  - **İş kuyruğu**– sistemdeki tüm prosesler kümesi
  - **Hazır kuyruğu**– ana bellekte bulunan, hazır ve işleme girmeyi bekleyen prosesler kümesi
  - **Bekleme(Aygıt) kuyrukları**– I/O aygıtlarından gelecek mesajı bekleyen prosesler kümesi
  - Prosesler kuyruklar arasında geçiş yapabilir.

Hazır Kuyruğu ve Çeşitli I/O Aygıt Kuyrukları

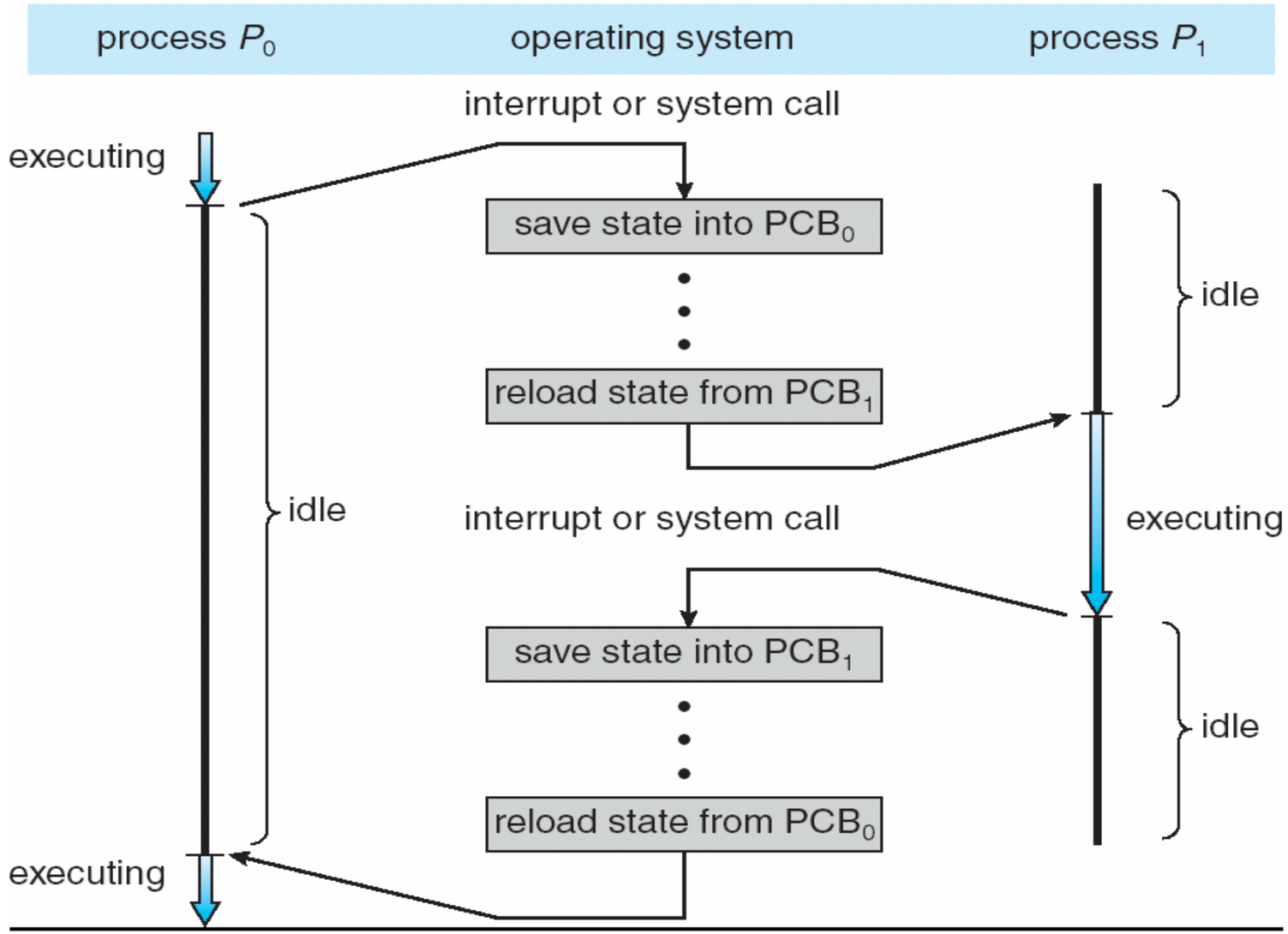


Proses Sıralama Diyagramı



CPU'da Proses Değişimi

CPU bir prosesten diğerine geçtiğinde bir bağlam değişimi **(context switch)** meydana gelir



Bağlam Değişimi (Context Switch)

- CPU diğer prosese geçtiği zaman, sistem mutlaka eski prosesin **durumunu kaydetmeli** ve **bağlam değişimi** yoluyla yeni prosesin daha önce **kaydedilmiş durumunu** yüklemeli

- Bir prosesini **bağlamı(context)** PCB ile temsil edilir
- Bağlam değiştirme bir maliyettir; sistem geçişler sırasında kullanım dışıdır
  - Daha karmaşık OS ve PCB -> daha uzun bağlam değişimi

- Donanım desteği zamana bağlıdır.
  - Bazı donanımlar CPU başına birden fazla kaydedici sağlar -> birden fazla bağlam bir kerede yüklenir

Mobil Sistemlerde Çoklugörev

- Bazı mobil sistemler (örneğin, iOS'un erken sürümü) yalnızca bir işlemin çalışmasına izin verir, diğerleri askıya alınır
- Ekran alanının darlığı nedeniyle, iOS'un sağladığı kullanıcı arabirimi sınırları
  - Kullanıcı arayüzü üzerinden kontrol edilen tek **ön plan** prosesi
  - Çoklu **arka plan** prosesleri– bellekte, çalışıyor ancak ekranda değil ve limitli
  - Sınırlar, tek, kısa görev, sadece olayların bildirimini alma, ses çalma gibi uzun süredir devam eden belirli görevleri içerir
- Android, daha az sınırla ön ve arka planda çalışır
  - Arka plan prosesi, görevleri gerçekleştirmek için bir **servis** kullanır
  - Arka plan prosesi askıya alınsa bile servis çalışmaya devam edebilir
  - Servisin kullanıcı arayüzü yok, küçük bellek kullanımı

Proses İşlemleri

- Sistem aşağıdaki işlermler için mekanizmalar sağlamalıdır:
  - Proseslerin oluşturulması
  - Proseslerin sonlandırılması

Proses Oluşturulması

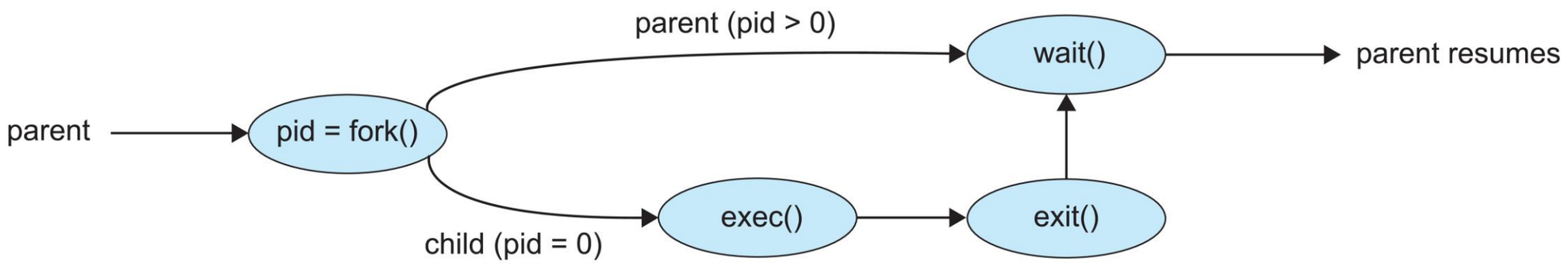
- **Ebeveyn** Proses, **çocuk** prosesleri oluşturur. Bu şekilde ağaç yapısı meydana gelir.
- Genelde prosesler**, bir proses kimlik numarası** (Proses identifier - **pid**) ile tanımlanır ve yönetilir.

- Kaynak Paylaşımı türleri:
  - Ebeveyn ve çocuk prosesler tüm kaynakları paylaşır.
  - Çocuk prosesler ebeveyn prosesin kaynaklarını kullanır.
  - Ebeveyn ve çocuk hiçbir kaynağı paylaşmaz.

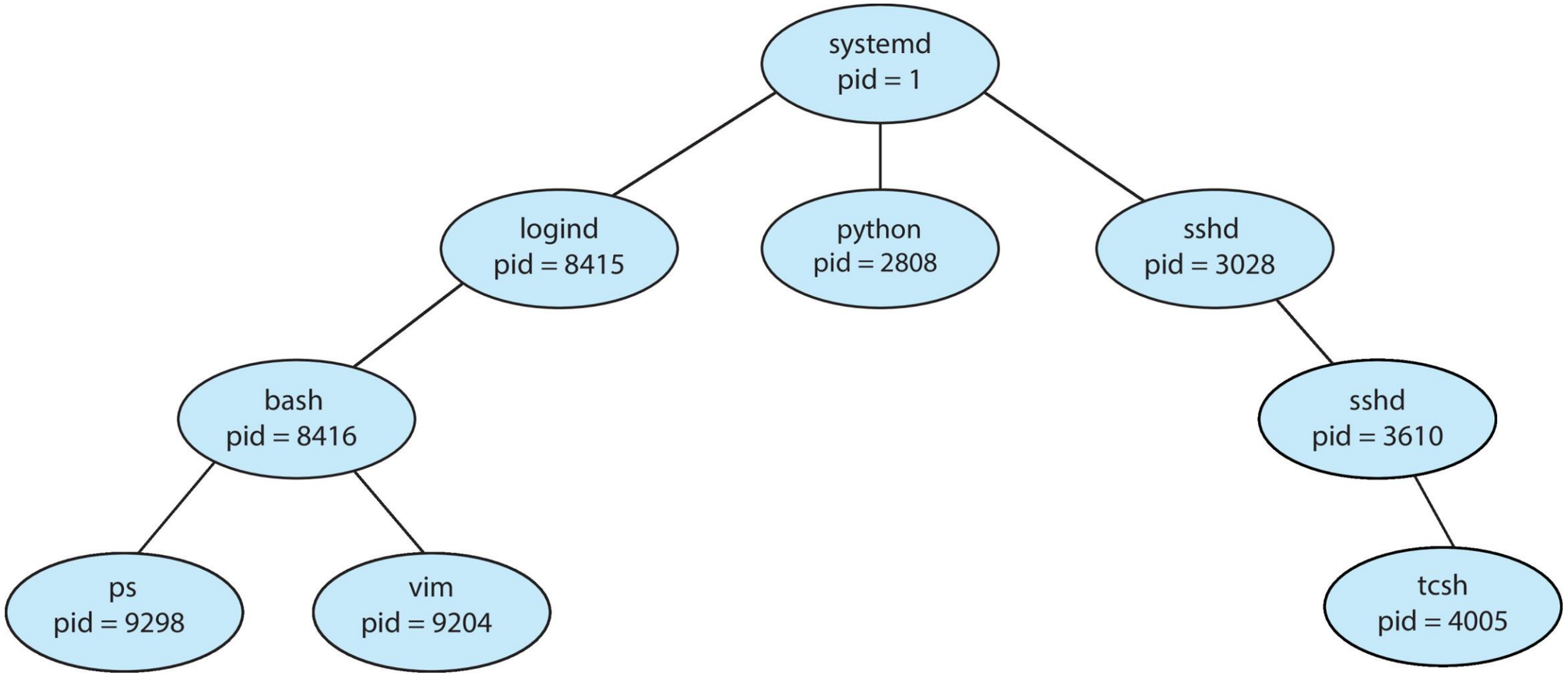
- Uygulama:
  - Ebeveyn ve çocuk proses eşzamanlı çalışır
  - Ebeveyn proses, çocuk proses sonlanana kadar bekler

Proses Oluşturulması (Devam)

- Adres alanı
  - Çocuk proses, ebeveyn prosesin alanını kopyalar .
  - Çocuk prosese bir program yüklenmiş olur.
- UNIX örnekleri :
  - **`fork()`** sistem çağrısı yeni bir proses oluşturur
  - **`exec()`** sistem çağrısı prosesin bellek alanını yeni bir program ile değiştirmek için bir **`fork()`** sonrası çağrılır.
  - Ebeveyn proses çocuğun sonlanmasını beklemek için **`wait ()`** çağırır



Linux Proses Ağacı



Fork İşlemi Yapan C Programı

```
#include <sys/types.h>
#include <studio.h>
#include <unistd.h>
int main()
{
pid_t  pid;
      /* fork another Proses */
      pid = fork();
      if (pid < 0) { /* error occurred */
            fprintf(stderr, "Fork Failed");
            return 1;
      }
      else if (pid == 0) { /* child Proses */
            execlp("/bin/ls", "ls", NULL);
      }
      else { /* parent Proses */
            /* parent will wait for the child */
            wait (NULL);
            printf ("Child Complete");
      }
      return 0;
}
```

Windows API Aracılığıyla Ayrı Proses Oluşturma

```
#include <stdio.h>
#include <windows.h>

int main(VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

    /* allocate memory */
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    /* create child process */
    if (!CreateProcess(NULL, /* use command line */
     "C:\\WINDOWS\\system32\\mspaint.exe", /* command */
     NULL, /* don't inherit process handle */
     NULL, /* don't inherit thread handle */
     FALSE, /* disable handle inheritance */
     0, /* no creation flags */
     NULL, /* use parent's environment block */
     NULL, /* use parent's existing directory */
     &si,
     &pi))
    {
      fprintf(stderr, "Create Process Failed");
      return -1;
    }
    /* parent will wait for the child to complete */
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Child Complete");

    /* close handles */
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```

Proses Sonlanması

- Proses son kod ifadesinin çalıştırır ve işletim sistemine **exit ()** sistem çağrısını kullanarak silinmesini talep eder
  - Durum verisi çocuktan ebeveyn prosese döner (**wait () aracılığıyla**)
  - Prosese ayırılan alan işletim sistemi tarafından geri alınır

- Ebeveyn **abort** () sistem çağrısını kullanarak çocuk proseslerin çalışmasını sonlandırabilir. Bunun sebepleri:
  - Çocuk tahsis edilen kaynakların dışına çıkmıştır.
  - Çocuk prosese tayin edilen iş artık gerekli değildir
  - Ebeveyn prosesten çıkılır, ve işletim sistemi ebeveyn proses sonlandırıldıktan sonra çocuk prosesin çalışmasına izin vermez

Proses Sonlanması

- Bazı işletim sistemleri, ebeveyni sonlandırıldıysa çocuk prosesin var olmasına izin vermez. Bir proses sona ererse, tüm çocukları da sonlandırılmalıdır.
  - **basamaklı sonlandırma**. Tüm çocukların, torunların vs. çalışmasına son verilir.
  - Sonlandırma işletim sistemi tarafından başlatılır.
  - Ebeveyn proses **`wait()`** sistem çağrısını kullanarak bir çocuk prosesin sonlandırılmasını bekleyebilir. Çağrı durum bilgilerini ve sonlandırılan prosesin pid'sini döndürür

```
pid = wait(&status);
```

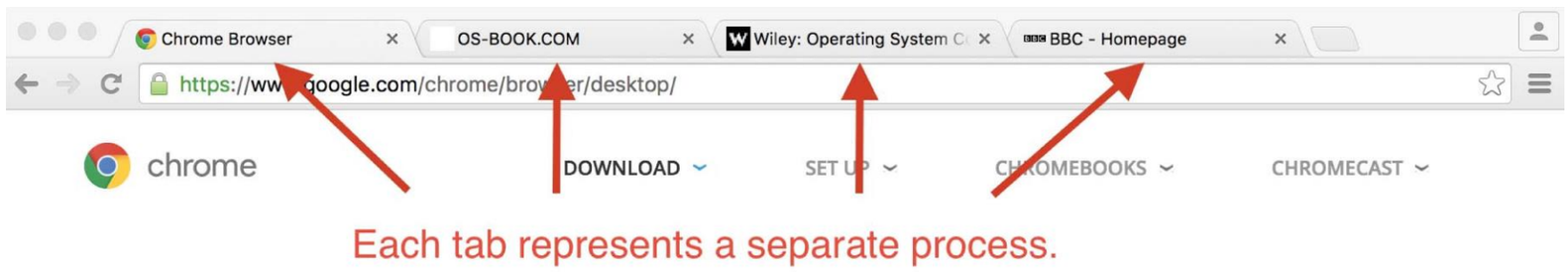
- Bekleyen hiçbir ebeveyn yoksa (wait() çağrılmadıysa) proses bir **zombidir.**
- Ebeveyn wait () çağrılmadan sonlandırıldıysa, proses bir **artıkdır(orphan)**

Android Proses Önem Hiyerarşisi

- Mobil işletim sistemleri genellikle bellek gibi sistem kaynaklarını geri kazanmak için prosesleri sonlandırmak zorundadır. En önemlisinden en az önemlisine:
  - Ön plan prosesi
  - Görünür proses
  - Servis prosesi
  - Arka plan prosesi
  - Boş proses
- Android, en az önemli olan prosesleri sonlandırmaya başlayacaktır.

Çoklu Proses Mimarisi – Chrome Tarayıcısı

- Birçok web tarayıcısı tek proses olarak çalıştırılır (bazıları hala öyle)
  - Bir web sitesi soruna neden olursa, tüm tarayıcı askıda kalabilir veya çökebilir
- Google Chrome Tarayıcı 3 farklı proses türüyle çoklu bir proses yapısındadır:
  - **Tarayıcı** prosesi kullanıcı arayüzünü yönetir, disk ve ağ G/Ç
  - **İşleyici (Renderer)** prosesi web sayfalarını işler, HTML, Javascript ile ilgilenir. Her web sitesi için oluşturulan yeni bir işleyici bulunur
    - Disk ve ağ G/Ç'yi kısıtlayan, güvenlik açıklarının etkisini en aza indiren korumalı (**sandbox**) bir yapıda çalışır
  - Her eklenti türü için **Plug-in** prosesi



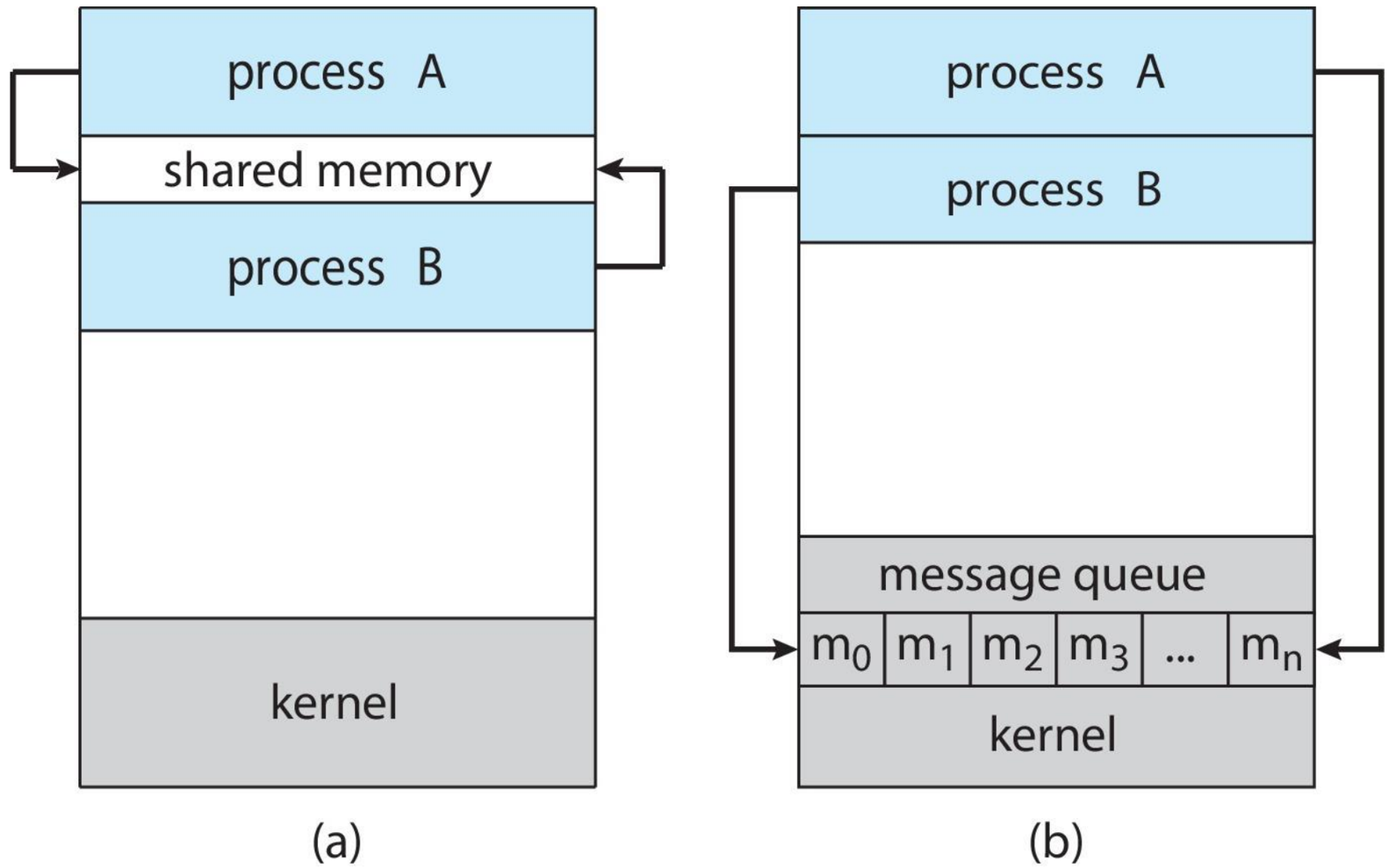
Prosesler Arası İletişim

- Prosesler işletim sistemi içerisinde **bağımsız** ya da **işbirliği halinde** çalışabilirler.
- İşbirliği içerisindeki prosesler veri paylaşımı da dahil olmak üzere diğer prosesleri etkileyebilir ya da diğer proseslerden etkilenebilirler.
- Proseslerin işbirliği yapma nedenleri:
  - Bilgi paylaşımı
  - Daha hızlı hesaplama
  - Modülerlik
  - Rahatlık
- İşbirliği içindeki prosesler **prosesler arası haberleşmeye (Interproses communication - IPC)** ihtiyaç duyarlar.
- 2 temel IPC modeli mevcuttur:
  - **Paylaşımlı bellek** (kullanıcıların kontrolünde)
  - **Mesajlaş iletimi** (OS nin kontrolünde)

Haberleşme Modelleri

(a) Shared memory.

(b) Message passing.



Üretici-Tüketici Problemi

- İşbirliği içindeki proseslere ilişkin bir paradigma:
  - **üretici** proses **tüketici** proses tarafından kullanılmak üzere bilgi üretir.
- İki versiyonu vardır:
  - **Sınırlandırılmamış tampon**: tampon için limit konulmamıştır
    - Üretici asla beklemez
    - Tüketici tamponda ürün yoksa bekler
  - **Sınırlı tampon:** sabit bir tampon boyutu mevcuttur.
    - Üretici eğer tampon dolu ise beklemez zorunda
    - Tüketici tamponda ürün yoksa bekler

Paylaşımlı Bellek Çözümü

- İletişim kurmak isteyen prosesler arasında paylaşılan bir bellek alanı
- İletişim, işletim sisteminin değil, kullanıcı proseslerinin kontrolü altındadır.
- Önemli sorun, kullanıcı proseslerinin paylaşılan belleğe eriştiklerinde eylemlerini senkronize etmesine izin verecek bir mekanizma sağlamaktır.
- Senkronizasyon, Bölüm 6 ve 7'de ayrıntılı olarak ele alınacaktır

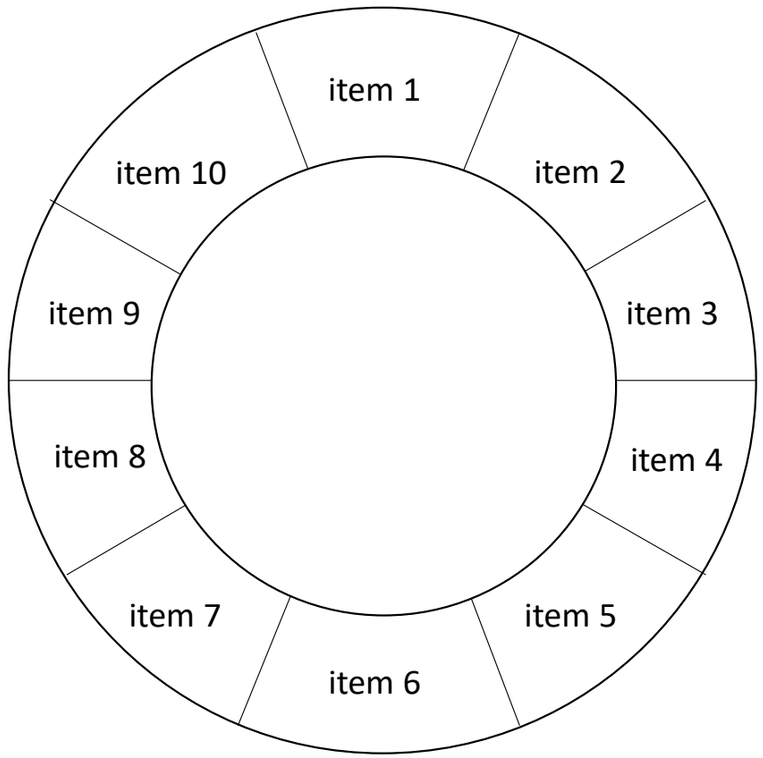
Sınırlı-Tamponlu–Paylaşımlı-Bellek Çözümü

- Paylaşılan veri

```
#define BUFFER_SIZE 10

typedef struct {

  . . .

} item;

item buffer[BUFFER_SIZE];

int in = 0;

int out = 0;
```

- Çözüm doğru, ancak sadece BUFFER_SIZE-1 eleman kullanılabilir.

Sınırlı-Tampon (devamı)



Sınırlı-Tampon – Üretici

```
item next_produced;

while (true) {
   /* produce an item in next produced */
   while (((in + 1) % BUFFER_SIZE) == out)
      ; /* do nothing */
   buffer[in] = next_produced;
   in = (in + 1) % BUFFER_SIZE;
}
```

Sınırlı Tampon – Tüketici

```
item next_consumed;

while (true) {
    while (in == out)
        ; /* do nothing */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    /* consume the item in next
consumed */
}
```

Tamponlar dolduğunda ne olur?

- **Tüm** tamponları dolduran tüketici-üretici sorununa bir çözüm sunmak istediğimizi varsayalım.
- Bunu, tam tampon sayısını izleyen bir tamsayı **`sayacı`** ile yapabiliriz.
- Başlangıçta **`sayaç`** 0 olarak ayarlanır.
- Tamsayı **`sayacı`**, yeni bir tampon elemanı ürettikten sonra üretici tarafından artırılır.
- Tamsayı **`sayacı`**, tampon elemanını tükettikten sonra tüketici tarafından azaltılır ve azaltılır.

Üretici

```
while (true) {
        /* produce an item in next produced */


        while (counter == BUFFER_SIZE)
                ; /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

Tüketici

```
while (true) {
        while (counter == 0)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
      counter--;
        /* consume the item in next consumed */
}
```

Yarış Durumu

- `counter++` aşağıdaki gibi çalışır

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- `counter--` aşağıdaki gibi çalışır

```
register2 = counter
register2 = register2 - 1
counter = register2
```

- Başlangıçta "count = 5" ile bu yürütme sırasını ele alalım:

S0: producer execute `register1 = counter` {register1 = 5}
S1: producer execute `register1 = register1 + 1` {register1 = 6}
S2: consumer execute `register2 = counter` {register2 = 5}
S3: consumer execute `register2 = register2 - 1` {register2 = 4}
S4: producer execute `counter = register1` {counter = 6 }
S5: consumer execute `counter = register2` {counter = 4}

Yarış Durumu (devam)

- Soru – neden ilk çözümde (en fazla N – 1) tamponların doldurulabileceği bir yarış durumu yoktu?
- Bölüm 6'da daha fazlası.

IPC - Prosesler Arası İletişim - Mesajlaşma

- Proseslerin iletişim mekanizması diğer proseslerin işlemleriyle senkronizedir.
- Mesaj sistemi – prosesler birbiri ile, paylaşılan değişkenleri kullanmadan iletişim kurar.
- IPC iki işlemi destekler :
  - **send**(*message*)
  - **receive**(*message*)
- Gönderilecek mesaj boyutu, sabit ya da değişken olabilir.

Mesajlaşma (devam)

- Eğer *P* ve *Q* prosesleri iletişim kurmak istiyorsa, şu işlemleri yapmaları gerekir :
  - Aralarında **iletişim bağlantısı** var olmalıdır.
  - send /receive yardımı ile mesaj alışverişi gerçekleştirmelidirler.
- Uygulama sorunları:
  - Bağlantılar nasıl kurulur?
  - Bir bağlantı ikiden fazla prosesle ilişkilendirilebilir mi?
  - Her iletişim proses çifti arasında kaç bağlantı olabilir?
  - Bağlantının kapasitesi nedir?
  - Bağlantının sabit veya değişken bir iletinin boyutu mu?
  - Bağlantı tek yönlü mü yoksa çift yönlü mü?

Bir İletişim Bağlantısının Uygulanması

- Fiziksel:
  - Paylaşımlı bellek
  - Donanım veri yolu
  - Ağ
- Mantıksal:
  - Doğrudan veya dolaylı
  - Senkron veya asenkron
  - Otomatik veya açık tamponlu

Doğrudan İletişim

- Proseslerin her biri gönderici ve alıcı olarak isimlendirilmelidir :
  - **send** (*P, message*) – P prosesine mesaj gönder
  - **receive**(*Q, message*) – Q prosesinden mesaj al

- İletişim bağlantısının özellikleri :
  - Bağlantılar otomatik olarak kurulur.
  - Her bir proses çifti arasında tam olarak bir bağlantı vardır.
  - Bir link 2 proses ile ilişkilendirilebilir.
  - Bağlantı tek yönlü olabilir, ancak genellikle iki yönlüdür.

Dolaylı İletişim

- Mesajlar port veya posta kutularından alınır veya buralara gönderilir.
  - Her posta kutusu tek bir tanımlayıcıya sahiptir
  - Prosesler paylaşılmış bir posta kutusuna sahipse iletişim kurabilirler.

- İletişim bağlantısı özellikleri şunlardır :
  - Bağlantı, prosesler arası paylaşılmış bir posta kutusu var ise kurulur.
  - Bir bağlantı ikiden fazla proses ile ilişkilendirilebilir.
  - Her bir proses çifti birden fazla bağlantıya sahip olabilir.
  - Bağlantı tek yönlü ya da çift yönlü olabilir.

Dolaylı İletişim

- İşlemler
  - Yeni bir posta kutusu oluştur,
  - Posta kutusu aracılığıyla mesaj gönder ve al.
  - posta kutusunu yok et.

- İletişim basitçe şu şekilde gerçekleşir:

  send(*A, message*) – A'nın posta kutusuna bir mesaj gönder

  receive(*A, message*) – A'nın posta kutusundan bir mesaj al.

Dolaylı İletişim

- Posta kutusu paylaşımı
  - P_1, P_2 ve P_3 prosesleri A posta kutusunu paylaşıyor.
  - P_1 mesaj gönderiyor; P_2 ve P_3 mesajı alıyor.
  - Mesajı hangisi almıştır?
- Çözüm:
  - Bir bağlantının en fazla iki proses ile ilişkilendirilmesine izin ver
  - Bir seferde yalnızca bir proses yürütmesine izin ver
  - Sistemin rastgele bir alıcı seçimine izin ver. Gönderici, alıcının kim olduğunu bildirir.

Senkronizasyon

- Mesaj iletimi engelli ya da engelsiz olabilir.
- **Engelli**, **senkron** iletim olarak düşünülebilir.
  - **Engelli gönderim,** mesaj alınana kadar gönderici engellenir.
  - **Engelli alım,** mesaj hazır olana kadar alıcı engellenir.
- **Engelsiz, asenkron** iletim olarak düşünülebilir.
  - **Engelsiz gönderim**, mesaj yollanır ve devam edilir.
  - **Engelsiz alım**, hazır mesaj varsa alır yoksa boş-null değer alır.
- Farklı kombinasyonlar mümkün
  - Hem gönderme hem de alma engelliyorsa, canlı yayın (**rendezvous)**

Üretici-Tüketici: Mesaşlaşma

- Üretici

```
message next_produced;
 while (true) {
/* produce an item in next_produced */

 send(next_produced);
}
```

- Tüketici

```
message next_consumed;
while (true) {
 receive(next_consumed)

/* consume the item in next_consumed */
}
```

Tamponlama

- Bağlantıyla ilişkilendirilmiş mesaj kuyruğu,
- 3 şekilde uygulanır:
  1. Sıfır kapasite – 0 mesaj
     Gönderici, alıcıyı beklemelidir (Buluşma).
  2. Sınırlı kapasite –n adet mesajın sonlu bir uzunluğa sahip olması
     Gönderici, bağlantı dolu ise beklemelidir.
  3. Sınırsız kapasite – sonsuz uzunluk
     Gönderici hiçbir zaman beklemez.

IPC Sistem Örnekleri - POSIX

- POSIX Paylaşılmış bellek
  - Proses öncelikle paylaşılmış bellek alanı oluşturur.

```
shm_fd = shm_open(name, O CREAT | O RDWR, 0666);
```

  - Aynı zamanda mevcut bir segment i açmak için kullanılır
  - Nesnenin boyutu belirlenir

```
ftruncate(shm_fd, 4096);
```

  - Bir dosya işaretçisini paylaşılan bellek nesnesine eşlemek için **`mmap()`** kullanın
  - Okuma ve paylaşılan belleğe yazma **`mmap()`** tarafından döndürülen işaretçi kullanılarak yapılır.

IPC POSIX Üreteci

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* strings written to shared memory */
const char *message_0 = "Hello";
const char *message_1 = "World!";

/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

   /* create the shared memory object */
   shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

   /* configure the size of the shared memory object */
   ftruncate(shm_fd, SIZE);

   /* memory map the shared memory object */
   ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);

   /* write to the shared memory object */
   sprintf(ptr,"%s",message_0);
   ptr += strlen(message_0);
   sprintf(ptr,"%s",message_1);
   ptr += strlen(message_1);

   return 0;
}
```

IPC POSIX Tüketici

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

   /* open the shared memory object */
   shm_fd = shm_open(name, O_RDONLY, 0666);

   /* memory map the shared memory object */
   ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

   /* read from the shared memory object */
   printf("%s",(char *)ptr);

   /* remove the shared memory object */
   shm_unlink(name);

   return 0;
}
```

IPC Sistem Örnekleri - Mach

- Mach iletişimi mesaj tabanlıdır.
  - Hatta sistem çağrıları dahi birer mesajdır.
  - Her görev iki posta kutusu oluşturur – Kernel ve Notify
  - Mesajlar **`mach_msg()`** fonksiyonu tarafından gönderilir ve alınır
  - Portlar **`mach_port_allocate()`** oluşturularak haberleşir
  - Gönderme ve alma esnektir, mesela eğer posta kutusu dolu ise dört seçenek vardır:
    - Sonsuza kadar bekle
    - Milisaniyeler kadar bekle
    - Derhal döndür
    - Mesajı geçici kaydet

Mach Mesajları

```
#include<mach/mach.h>

struct message {
      mach_msg_header_t header;
      int data;
};

mach port t client;
mach port t server;
```

Mach Mesaj İletimi - Client

```
/* Client Code */

struct message message;

// construct the header
message.header.msgh_size = sizeof(message);
message.header.msgh_remote_port = server;
message.header.msgh_local_port = client;

// send the message
mach_msg(&message.header, // message header
   MACH_SEND_MSG, // sending a message
   sizeof(message), // size of message sent
   0, // maximum size of received message - unnecessary
   MACH_PORT_NULL, // name of receive port - unnecessary
   MACH_MSG_TIMEOUT_NONE, // no time outs
   MACH_PORT_NULL // no notify port
);
```

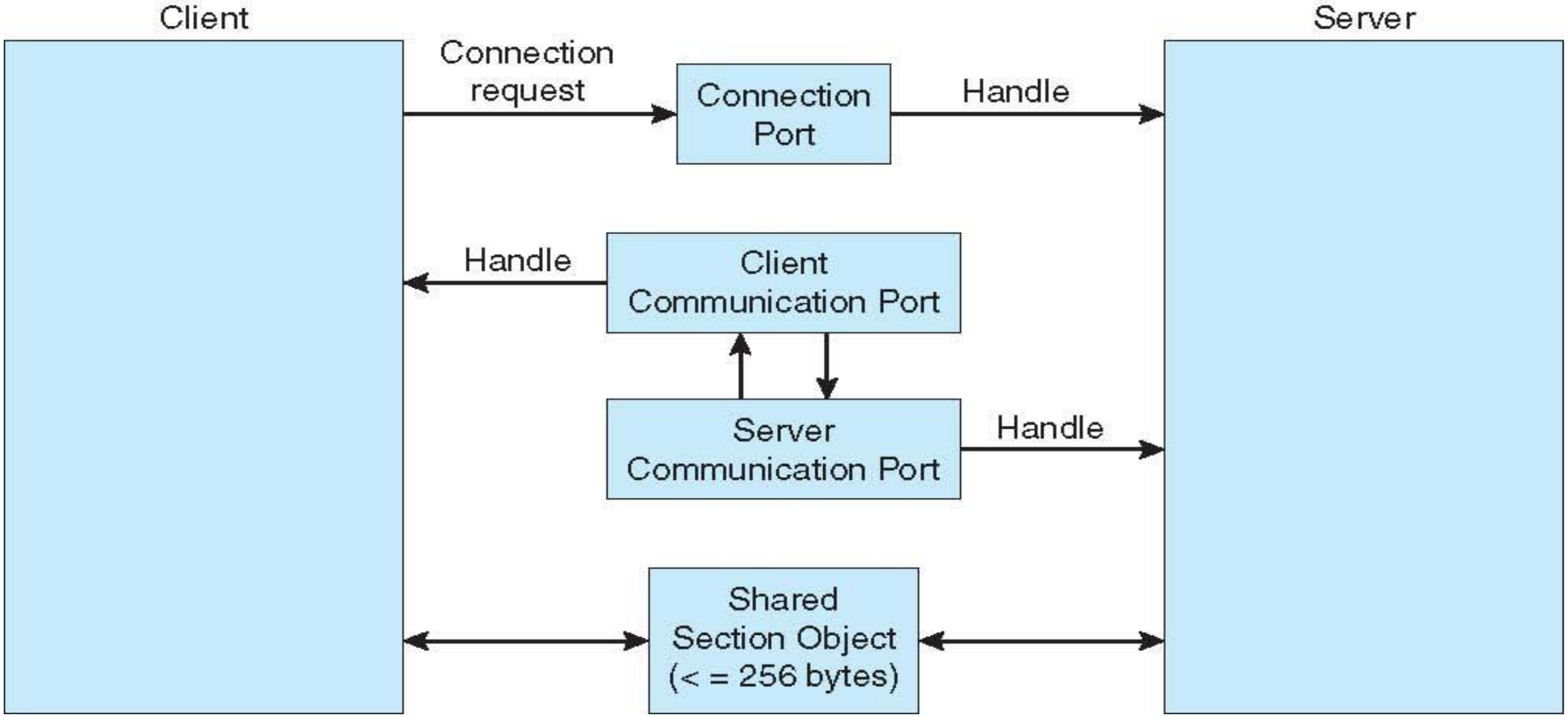
Mach Mesaj İletimi - Server

```
        /* Server Code */

struct message message;

// receive the message
mach_msg(&message.header, // message header
   MACH_RCV_MSG, // sending a message
   0, // size of message sent
   sizeof(message), // maximum size of received message
   server, // name of receive port
   MACH_MSG_TIMEOUT_NONE, // no time outs
   MACH_PORT_NULL // no notify port
);
```

IPC Sistem Örnekleri – Windows XP

- Mesaj iletimi gelişmiş yerel prosedür çağrı **(advanced local procedure call - LPC)** birimi aracılıyla yönetilir
  - Yalnızca prosesler ve benzeri sistemler arasında çalışır.
  - İletişim kanalları kurmak ve sürdürmek için portları (posta kutuları gibi) kullanır.
  - Haberleşme aşağıdaki gibi çalışır:
    - İstemci altsistemi bir **bağlantı port** nesnesi açar.
    - İstemci bağlantı isteği gönderir.
    - Sunucu iki özel **iletişim portu** oluşturur ve bunlardan birini istemciye gönderir.
    - İstemci ve sunucu mesajları göndermek, almak ve cevapları dinlemek amacıyla karşılıklı portlar kullanır.

Windows XP'de Yerel İşlem Çağrısı

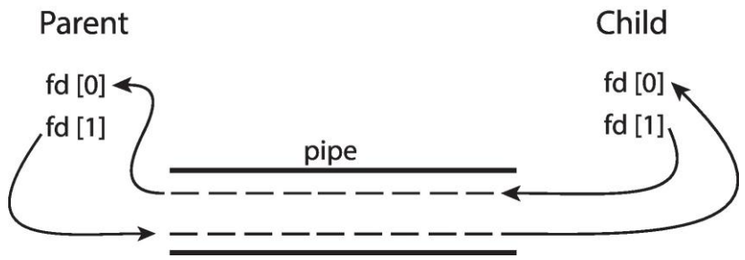


Veri Kanalları- Pipes

- İki proses arasında iletişime izin veren yapıdır.
- Sorunlar:
  - İletişim tek yönü mü, çift yönlü müdür?
  - İletişim iki yönlü ise yarı dubleks mi çalışır, yoksa tam dubleks mi çalışır?
  - İletişim halindeki prosesler arasında bir ilişki (ebeveyn-çocuk) olmalı mıdır?
  - Kanallar ağ üzerinden kullanılabilir mi?
- **Sıradan kanallar** –oluşturan prosesin dışından erişilemez. Genellikle, bir ebeveyn proses bir kanal oluşturur ve çocuk prosesi ile iletişim kurmak için kullanır.
- **İsimli kanallar** – üst-alt ilişkisi olmadan erişilebilir.

Sıradan Kanallar

- **Sıradan kanallar,** standart üretici-tüketici tipi iletişime izin verir.
- Üretici bir uçtan yazar (kanalın yazma ucu)
- Tüketici diğer ucundan okur (kanalın okuma ucu)
- Sıradan kanallar bu nedenle tek yönlü iletişim sağlar.
- Haberleşen prosesler arasında ebeveyn-çocuk ilişkisi gerekir.



- Windows bunları **anonim kanallar** olarak adlandırır

İsimli Kanallar

- İsimli Kanallar, sıradan olanlardan daha güçlüdür.
- İletişim çift yönlüdür.
- Haberleşen prosesler arasında ebeveyn-çocuk ilişkisi gerekli değildir.
- Birden fazla proses kullanabilir.
- UNIX ve Windows işletim sistemlerince desteklenir.

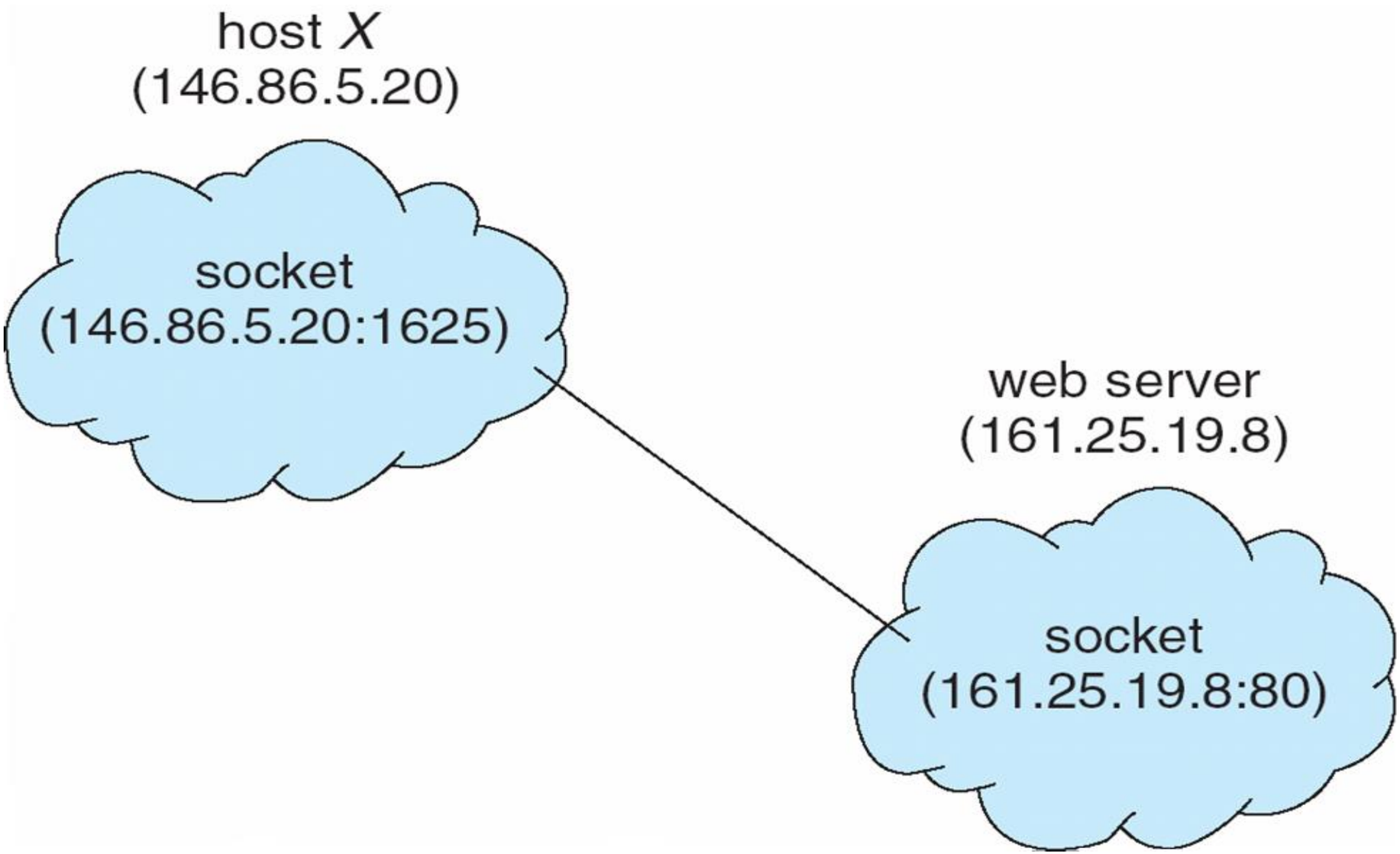
İstemci – Sunucu Sistemlerinde İletişim

- Soketler
- Uzaktan Prosedür Çağrıları
- Uzaktan Metot Çağrıları (RMI - Java)

Soketler

- Bir **soket**, bir iletişim uç noktası olarak tanımlanabilir.
- IP adresinin ve **portun** birleşimidir.
  - Port, bir ana bilgisayardaki ağ hizmetlerini ayırt etmek için mesaj paketinin başında bulunan bir sayıdır
- **161.25.19.8:1625** soketi, **1625** portu ve **161.25.19.8** sunucusu demektir.
- İletişim, bir çift soket arasında meydana gelir.
- 1024'ün altındaki tüm port numaralrı iyi bilinmektedir, standart hizmetler için kullanılır
- Prosesin çalıştığı sisteme başvurmak için özel IP adresi 127.0.0.1 (geridöngü-**loopback**)

Soket İletişimi



Java Soketleri

- Üç tip soket
  - **Connection-oriented** (**TCP**)
  - **Connectionless** (**UDP**)
  - **`MulticastSocket`** class– data can be sent to multiple recipients
- Consider this “Date” server in Java:

```
import java.net.*;
import java.io.*;

public class DateServer
{
   public static void main(String[] args) {
      try {
         ServerSocket sock = new ServerSocket(6013);

         /* now listen for connections */
         while (true) {
            Socket client = sock.accept();

            PrintWriter pout = new
             PrintWriter(client.getOutputStream(), true);

            /* write the Date to the socket */
            pout.println(new java.util.Date().toString());

            /* close the socket and resume */
            /* listening for connections */
            client.close();
         }
      }
      catch (IOException ioe) {
         System.err.println(ioe);
      }
   }
}
```

Java Soketleri

The equivalent Date client

```
import java.net.*;
import java.io.*;

public class DateClient
{
   public static void main(String[] args) {
      try {
         /* make connection to server socket */
         Socket sock = new Socket("127.0.0.1",6013);

         InputStream in = sock.getInputStream();
         BufferedReader bin = new
            BufferedReader(new InputStreamReader(in));

         /* read the date from the socket */
         String line;
         while ( (line = bin.readLine()) != null)
            System.out.println(line);

         /* close the socket connection*/
         sock.close();
      }
      catch (IOException ioe) {
         System.err.println(ioe);
      }
   }
}
```

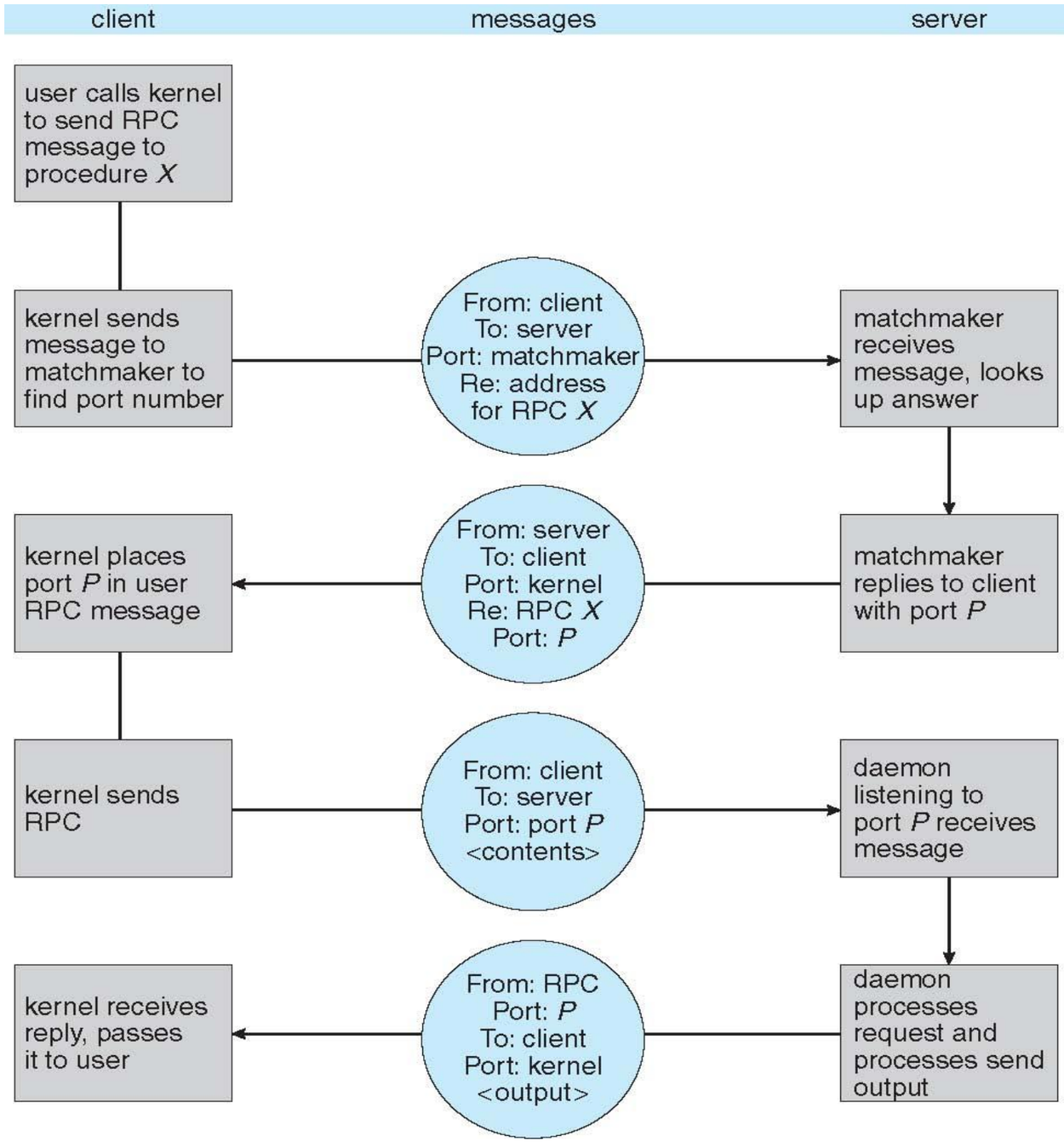
Uzak Yordam Çağrıları

- Uzak prosedür çağrısı (Remote procedure call - RPC), yordam çağrılarını bağlı sistemler üzerindeki işlemlere ayırır.
  - Servisleri ayırt etmek için portları kullanır
- **Stub** – sunucu üzerindeki gerçek prosedür için istemci tarafındaki aracı
- İstemci tarafındaki stub, sunucunun yerini belirler ve parametreleri yönlendirir.
- Sunucu tarafındaki stub, mesajı alır, yönlendirilmiş parametreleri açar ve yordamı sunucu üzerinde uygular.
- Windows' ta stub kodu **Microsoft Interface Definition Language** (**MIDL**) dili kullanılarak yazılır

Uzak Yordam Çağrıları (devam)

- Farklı mimariler için veri temsili **External Data Representation** (**XDL**) formatı ile yönetilir
  - **Big-endian** and **little-endian**
- Uzaktan iletişimde yerelden daha fazla hata senaryosu var
  - İletiler en fazla bir kez değil, tam olarak bir kez ve tam teslim edilir
- İşletim sistemi genellikle istemci ve sunucuyu bağlamak için bir randevu (veya **çöpçatan**) hizmeti sağlar

RPC'nin Çalışma Prensibi



Bölüm 3 - Son