

Bölüm 3: İşlemler (Processes)

Bölüm 3: İşlemler

- İşlem Kavramı
- İşlem Zamanlaması (Process Scheduling)
- İşlemler Üzerindeki Faaliyetler
- İşlemler Arası İletişim (Interprocess Communication)
- IPC Sistemi Örnekleri
- İstemci-Sunucu Sistemlerde İletişim

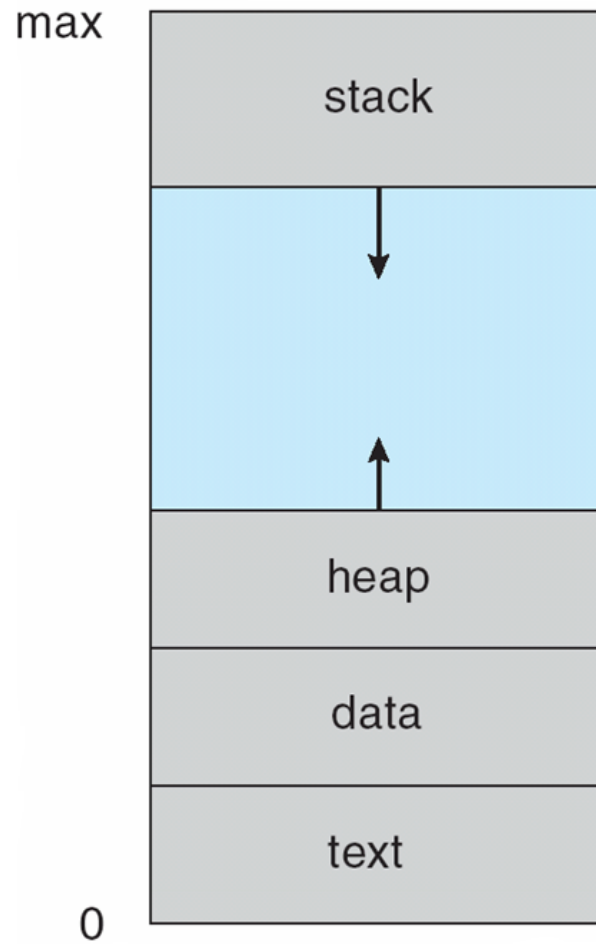
Hedefler

- İşlem kavramını (çalışmakta olan bir program) tanıtmak
- İşlemlerin pek çok özelliklerini tanıtmak: zamanlama, oluşturma, sonlandırma ve iletişim
- İstemci-sunucu sistemlerinde iletişimi açıklamak

İşlem Kavramı

- İşletim sistemleri farklı tipte programlar çalıştırabilir:
 - Toplu iş sistemleri - iş
 - Zaman paylaşımli sistemler (time-shared systems) – kullanıcı programları veya görevleri
- Ders kitabı **iş (job)** ve **işlem (process)** kelimelerini kabaca birbirleri yerine kullanabilmekte
- İşlem – çalışmakta olan bir programdır
- İşlemler aşağıdakileri içermelidir:
 - **program sayacı (program counter)**
 - **yığın (stack)**
 - **veri bölümü (data section)**

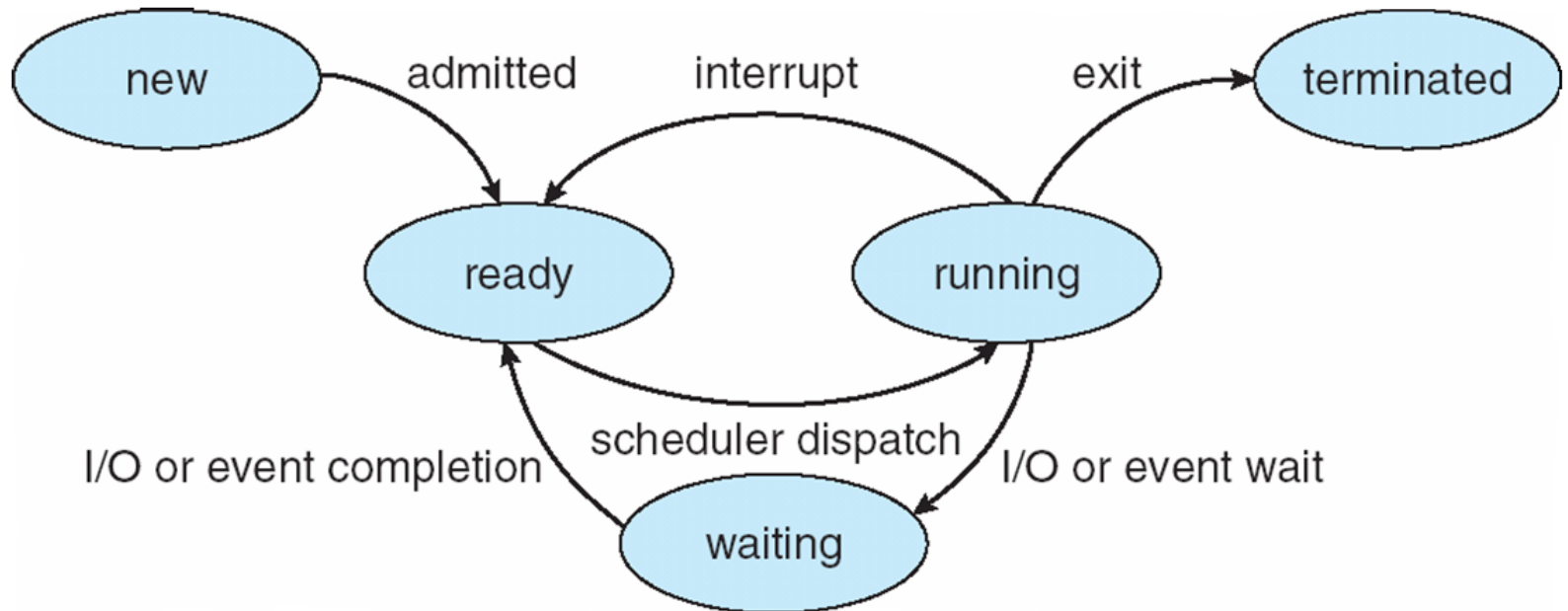
Hafızadaki İşlemler



İşlem Durumu

- Bir işlem çalıştırılırken durumunu (state) değiştirir
 - **yeni (new)**: İşlem oluşturuldu
 - **çalışıyor (running)**: İşlem komutları çalıştırılıyor
 - **bekliyor (waiting)**: İşlem bir olayın gerçekleşmesini bekliyor
 - **hazır (ready)**: İşlem bir işlemciye atanmayı bekliyor
 - **sonlandırılmış (terminated)**: İşlem çalışmayı bitirmiş

İşlem Durumlarını Gösteren Şema



İşlem Kontrol Bloğu (PCB)

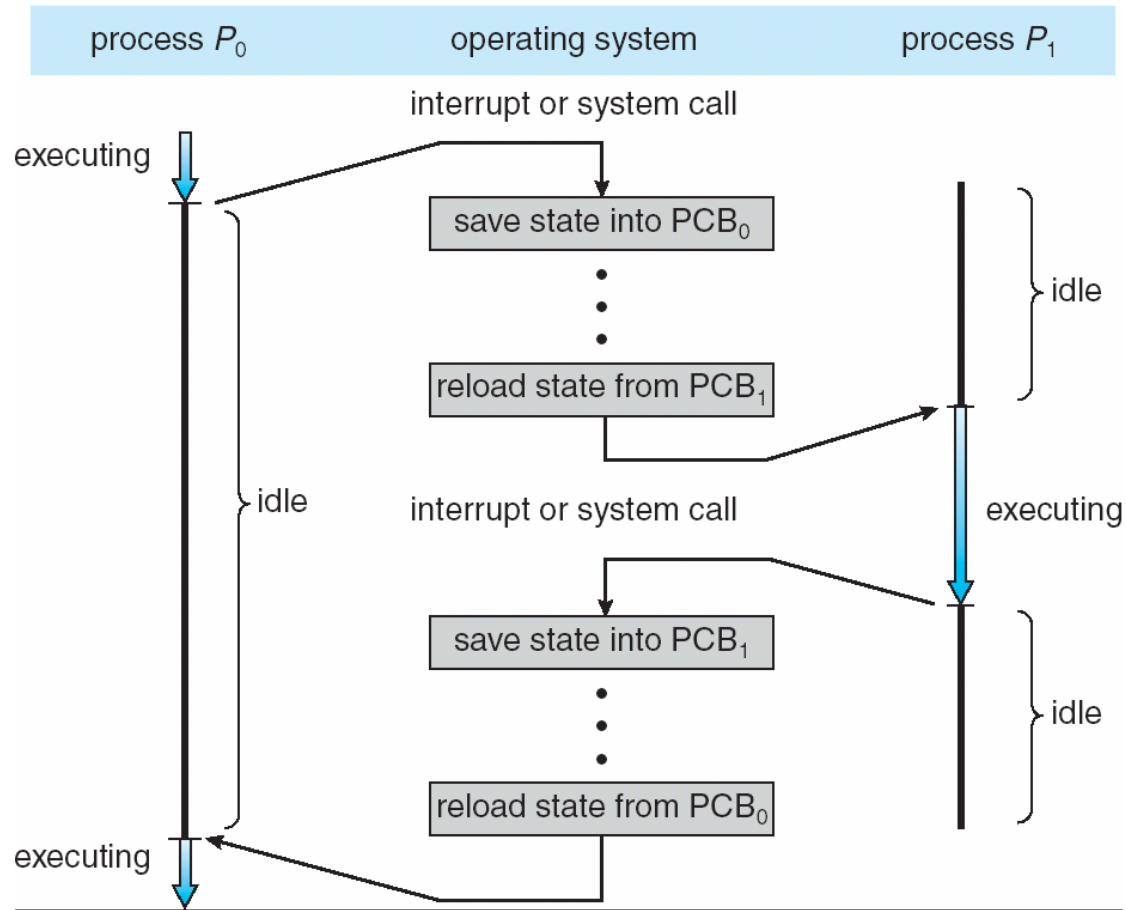
Herhangi bir işlem ile ilişkili bilgiler (**process control block**)

- İşlem durumu
- İşlem sayacı
- CPU yazmaçları (CPU registers)
- CPU zamanlama bilgileri
- Hafıza yönetim bilgileri
- Hesap (accounting) bilgileri
- I/O durum bilgileri

İşlem Kontrol Bloğu (PCB)



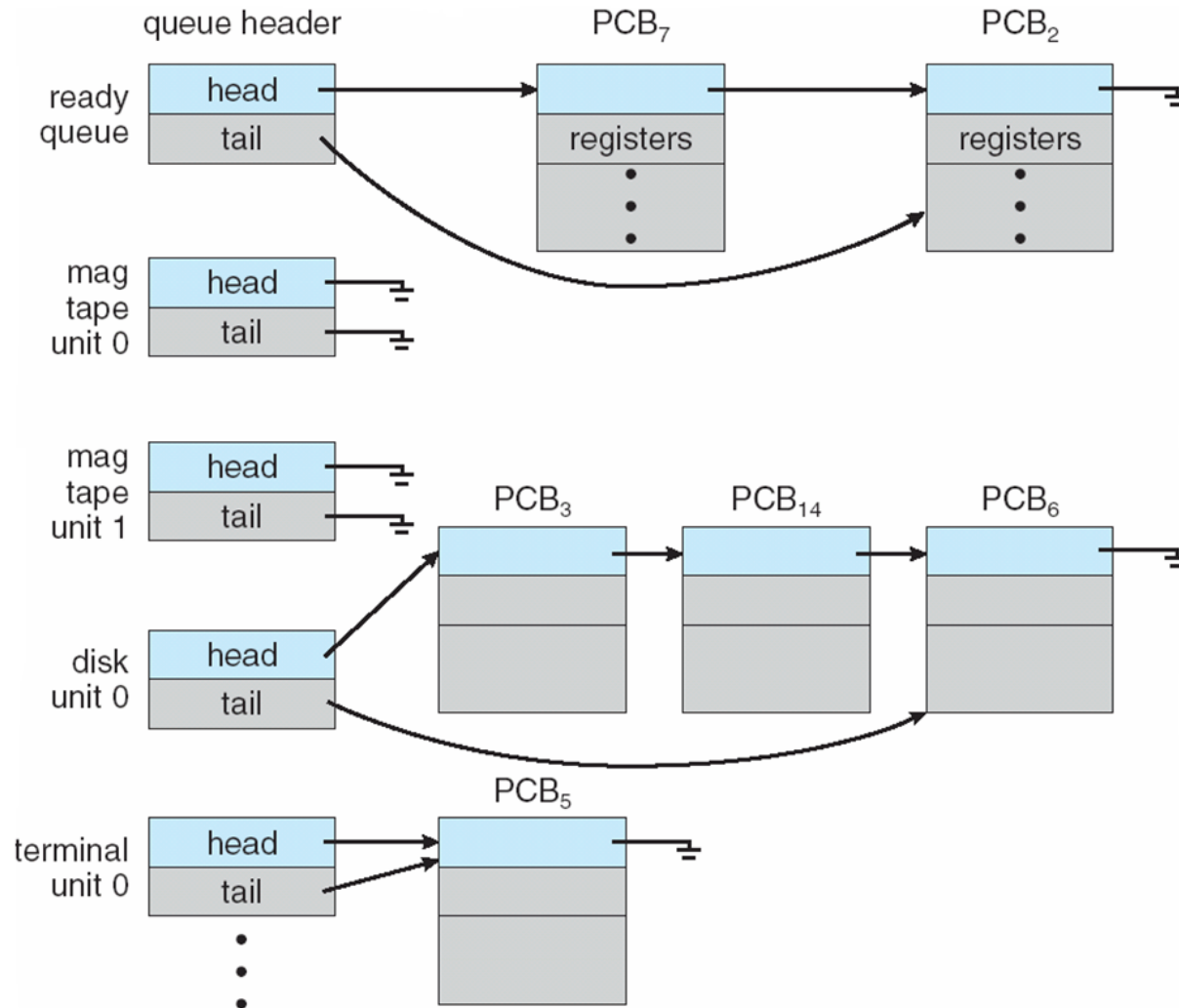
CPU İşlemden İşleme Geçiş Yapar



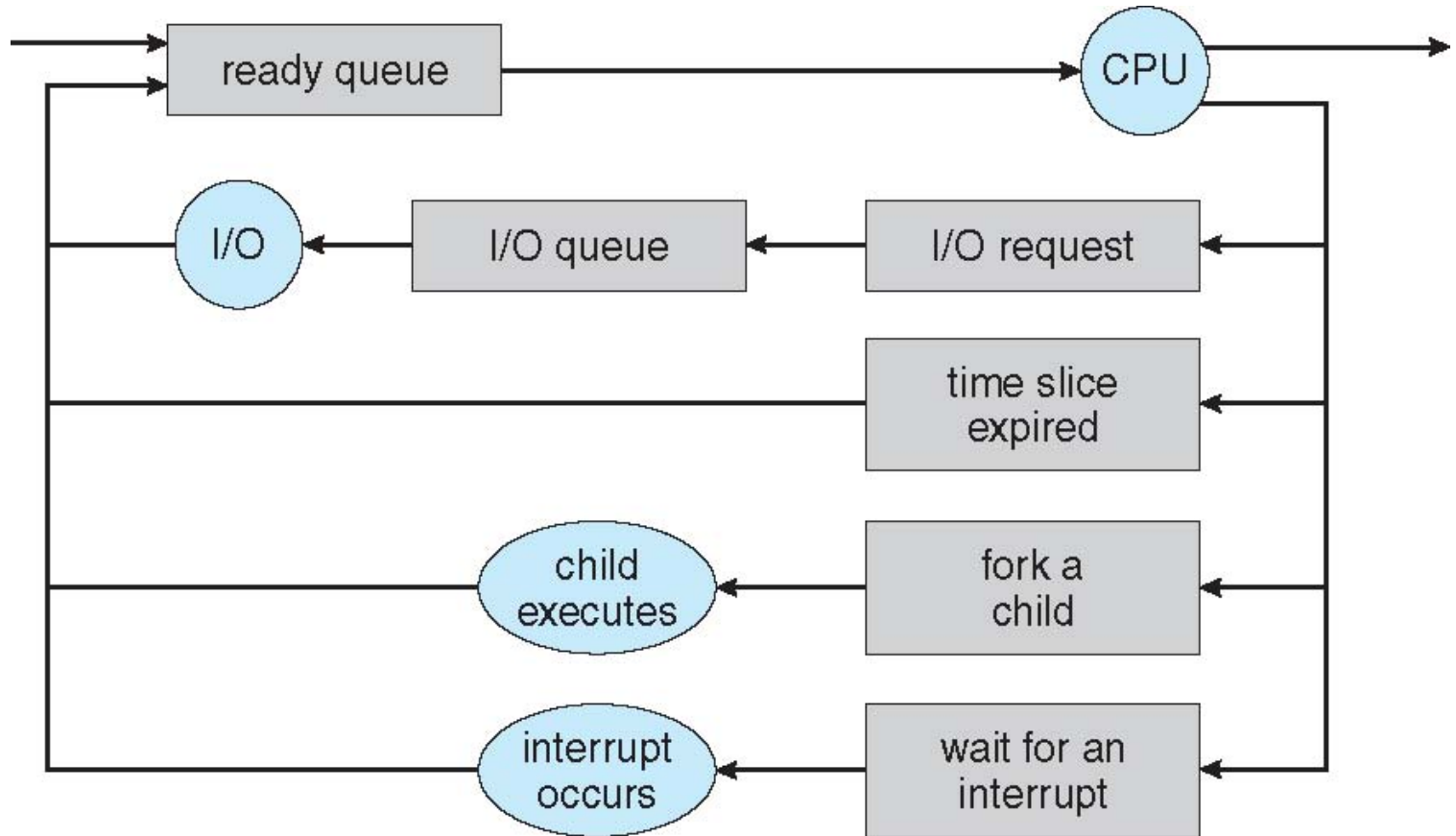
İşlem Zamanlama Kuyrukları

- Process Scheduling Queues
- **İş kuyruğu (job queue)** – sistemdeki tüm işlemlerin kümesi
- **Hazır kuyruğu (ready queue)** – ana hafızadaki, tüm işlemlerin kümesi - çalışmaya hazır ve çalıştırılmayı bekleyen
- **Cihaz kuyrukları (device queues)** – bir I/O cihazını kullanmayı bekleyen işlemler kümesi
- İşlemler değişik kuyruklar arasında geçiş yapar

Hazır Kuyruğu ve Bazı I/O Cihaz Kuyrukları



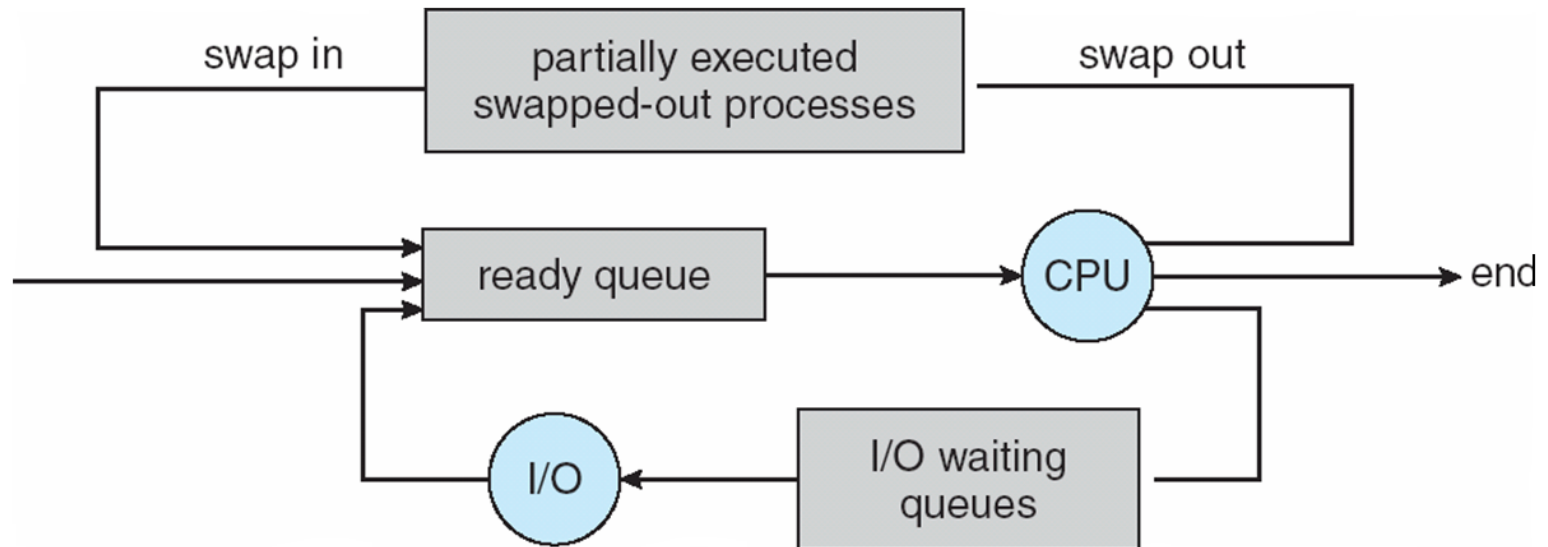
İşlem Zamanlaması Gösterimi



Zamanlayıcılar

- **Uzun vadeli zamanlayıcılar (long-term scheduler)**
 - veya iş zamanlayıcısı (job scheduler)
 - hangi işlemlerin hazır kuruğuna (ready queue) alınması gerektiğine karar verir
- **Kısa vadeli zamanlayıcılar (short-term scheduler)**
 - veya CPU zamanlayıcısı
 - sıradaki hangi işlemin CPU tarafından çalıştırılacağına karar verir

Orta Vadeli Zamanlama



Orta Vadeli Zamanlayıcı (medium-term scheduler)

Zamanlayıcılar- Planlayıcılar- Scheduler (devam)

- Kısa vadeli zamanlayıcı çok sık çalıştırılır (milisaniye) \Rightarrow hızlı çalışmalı
- Uzun vadeli zamanlayıcı çok sık çalıştırılmaz (saniye, dakika) \Rightarrow yavaş olabilir
- Uzun vadeli zamanlayıcı çoklu programlamanın (multiprogramming) seviyesini kontrol eder
- İşlemler aşağıdaki iki kategoriye ayrılabilir:
 - **I/O ağırlıklı işlemler (I/O-bound process)** – CPU üzerinde kısa süreli işlemler yaparlar ve zamanlarının çoğu I/O işleri yaparak geçer
 - **CPU ağırlıklı işlemler (CPU-bound process)** – zamanlarının çoğunu CPU üzerinde uzun işlemler yaparak geçirirler, I/O işleri çok daha azdır

Ortam Değişikliği

- **Context Switch**
- CPU başka bir işleme geçerken, sistem eski işlemin durumunu kaydetmeli ve yeni işlemin kaydedilmiş durumunu yüklemelidir
- Bir işlemin **ortamı (context)** PCB'de tutulur
- Ortam değişikliği için harcanan zaman ek yüküdür. Değişim sırasında işlemci kullanıcının işine direk yarayan bir iş yapmamaktadır
- Harcanan zaman donanım desteğine bağlıdır

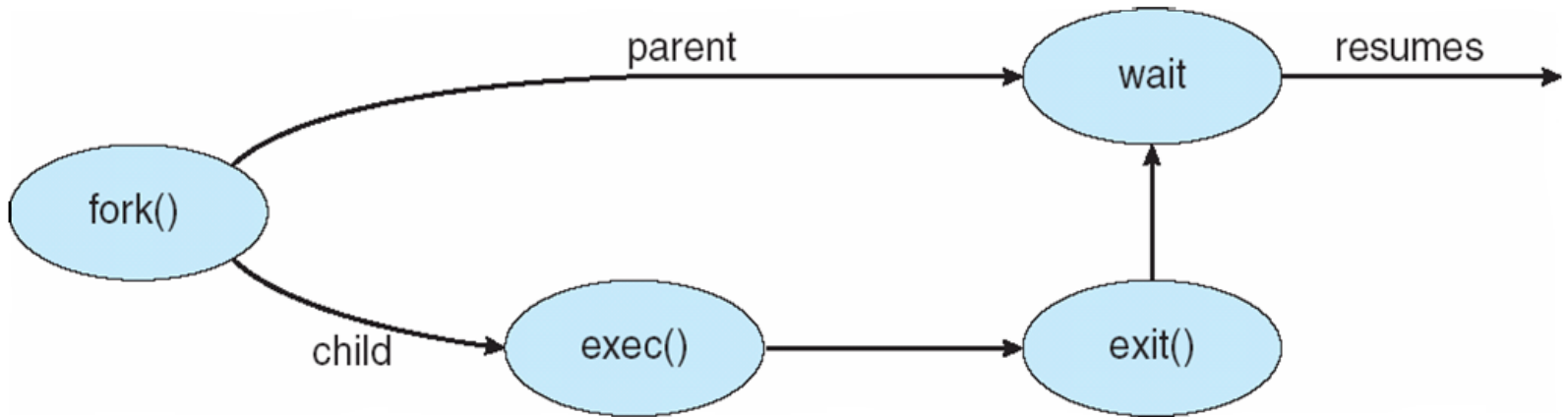
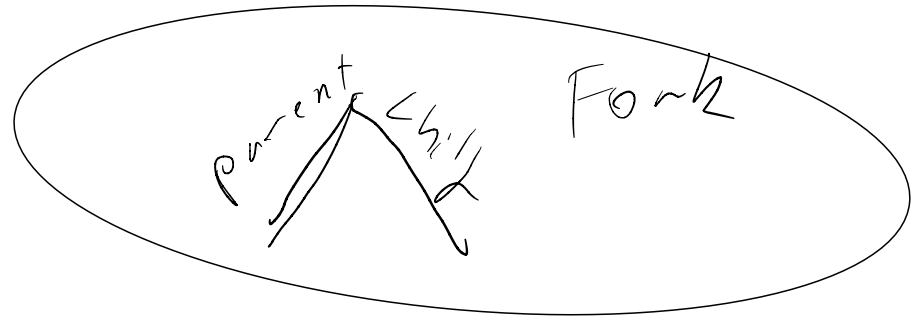
Process Oluşturma

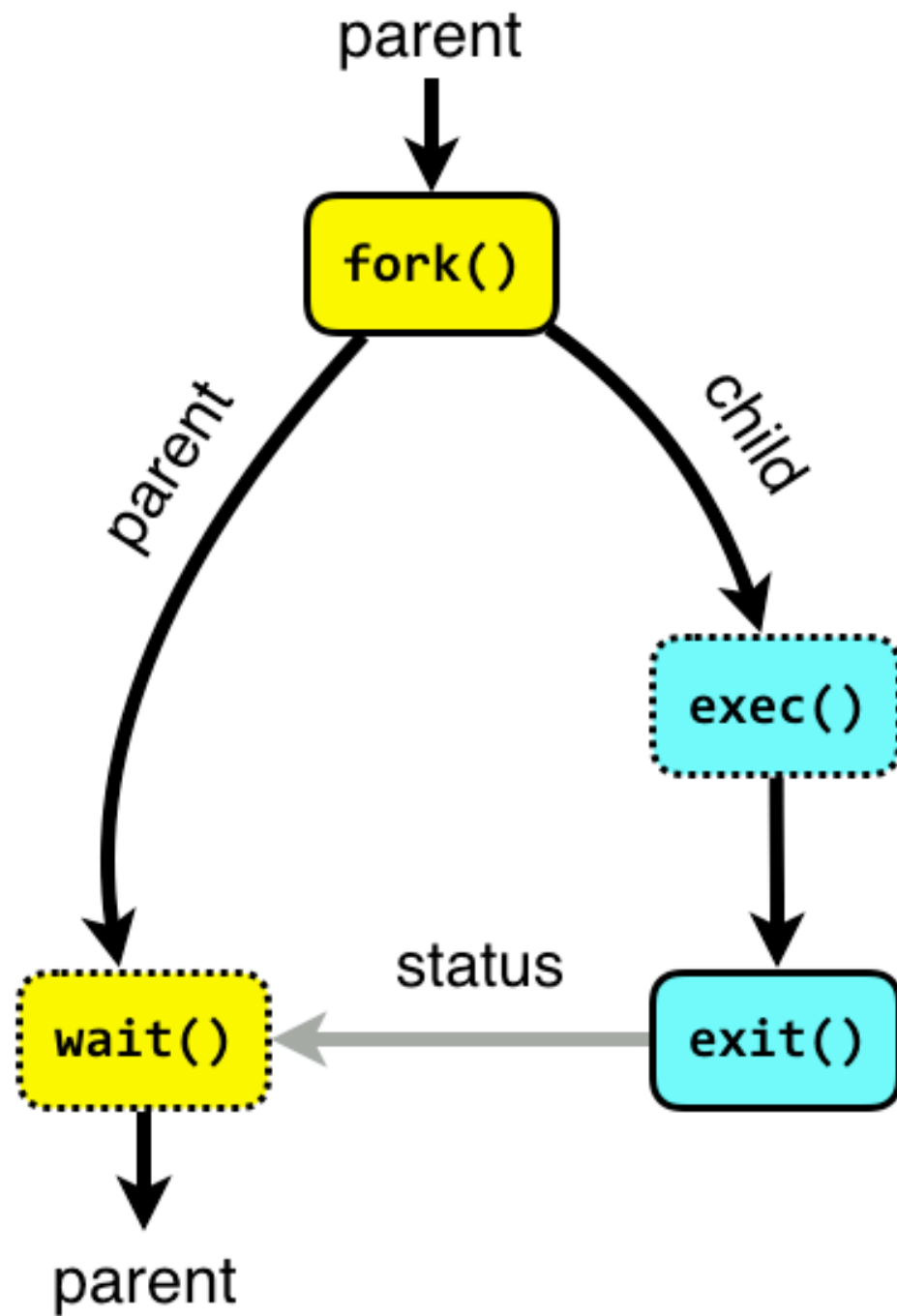
- **Ana işlem** (parent process) **çocuk işlemleri** (children processes) oluşturur
- Çocuk işlemlerde yeni işlem oluşturabileceğinden, sistemde işlemlerin bir ağacı oluşur
- Genellikle işlemler **işlem belirteci (process identifier - pid)** kullanılarak birbirlerinden ayrılırlar ve yönetilirler
- Kaynak paylaşımı
 - Ana işlem ve çocuklar tüm kaynakları paylaşırlar
 - Çocuklar ana işlemin kaynaklarının belli bir alt kümesini paylaşır
 - Ana işlem ve çocuklar kaynak paylaşmazlar
- Çalıştırma
 - Ana işlem ve çocuklar aynı anda çalışır
 - Ana işlem, çocuk işlemler sonlanana kadar bekler

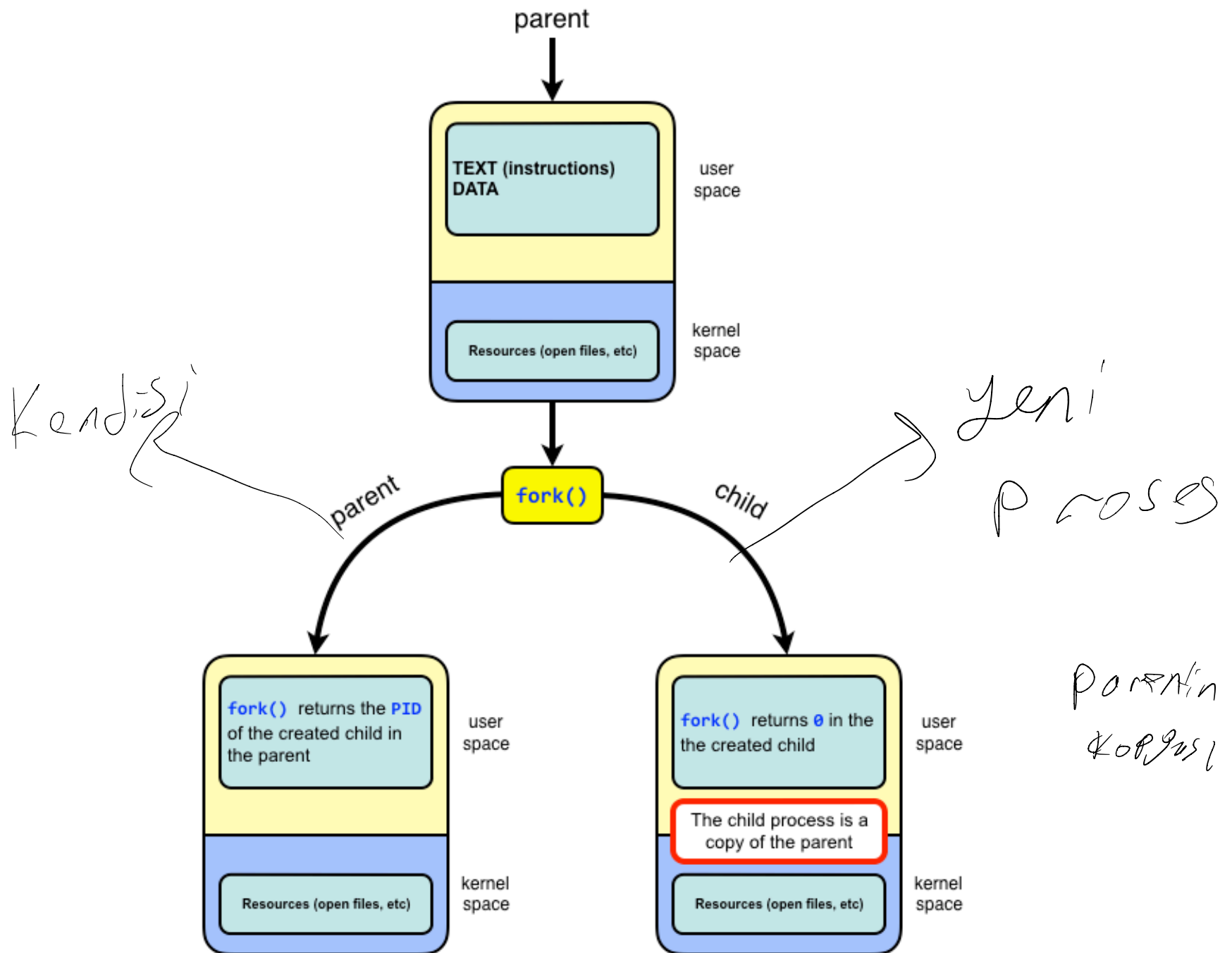
İşlem Oluşturma (devam)

- **Hafıza alanı (address space)**
 - Çocuk ana işlemin kopyasına sahip
 - Çocuk adres alanına yeni bir program yükler
- UNIX örnekleri
 - **fork** sistem çağrısı yeni bir işlem oluşturur
 - **exec** sistem çağrısı, **fork** sistem çağrısı ile yeni işlem oluşturulduktan sonra, işlemin hafıza alanına yeni bir program yüklemek için kullanılır

İşlem Oluşturma Şeması







Yeni İşlem Oluşturan C Programı

```
int main()
{
    pid_t  pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        exit(-1);
    }
    else if (pid == 0) { /* child process */
        execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to
        complete */
        wait (NULL);
        printf ("Child Complete");
        exit(0);
    }
}
```



POSIX'te İşlem Oluşturma

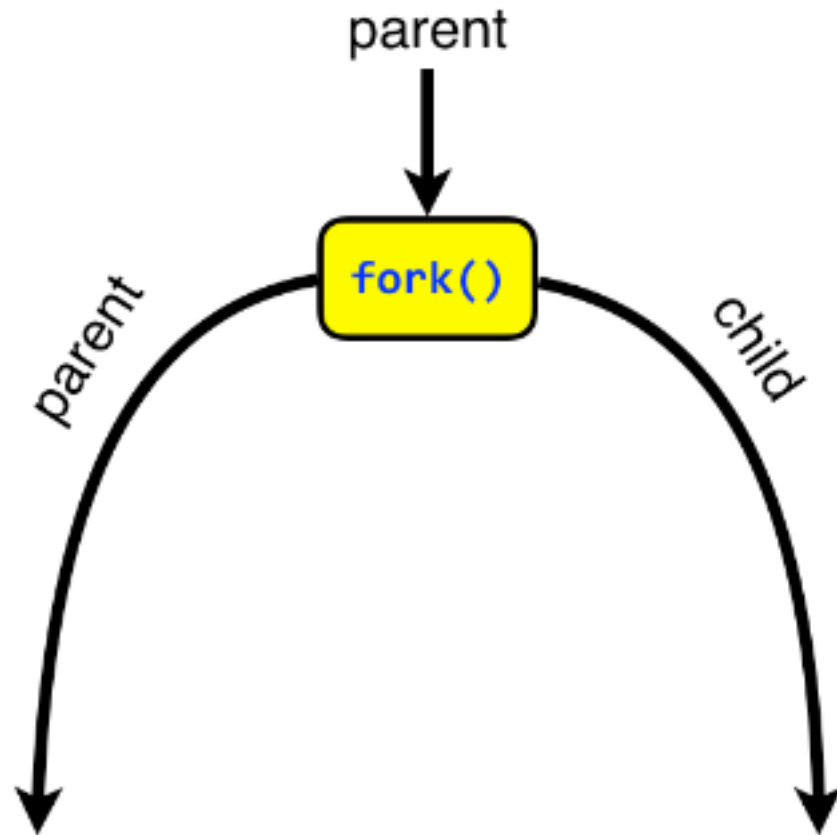
```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
    pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        exit(-1);
    }
    else if (pid == 0) { /* child process */
        execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
        printf("Child Complete");
        exit(0);
    }
}
```


POSIX'te İşlem Oluşturma wait kullanmadan



Win32'de İşlem 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 line
        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);
}
```

Java'da İşlem Oluşturma

Burada Kaldık

```
import java.io.*;

public class OSProcess
{
    public static void main(String[] args) throws IOException {
        if (args.length != 1) {
            System.err.println("Usage: java OSProcess <command>");
            System.exit(0);
        }

        // args[0] is the command
        ProcessBuilder pb = new ProcessBuilder(args[0]);
        Process proc = pb.start();

        // obtain the input stream
        InputStream is = proc.getInputStream();
        InputStreamReader isr = new InputStreamReader(is);
        BufferedReader br = new BufferedReader(isr);

        // read what is returned by the command
        String line;
        while ( (line = br.readLine()) != null)
            System.out.println(line);

        br.close();
    }
}
```

Figure 4-18. Creating and running a process

Zombie processes



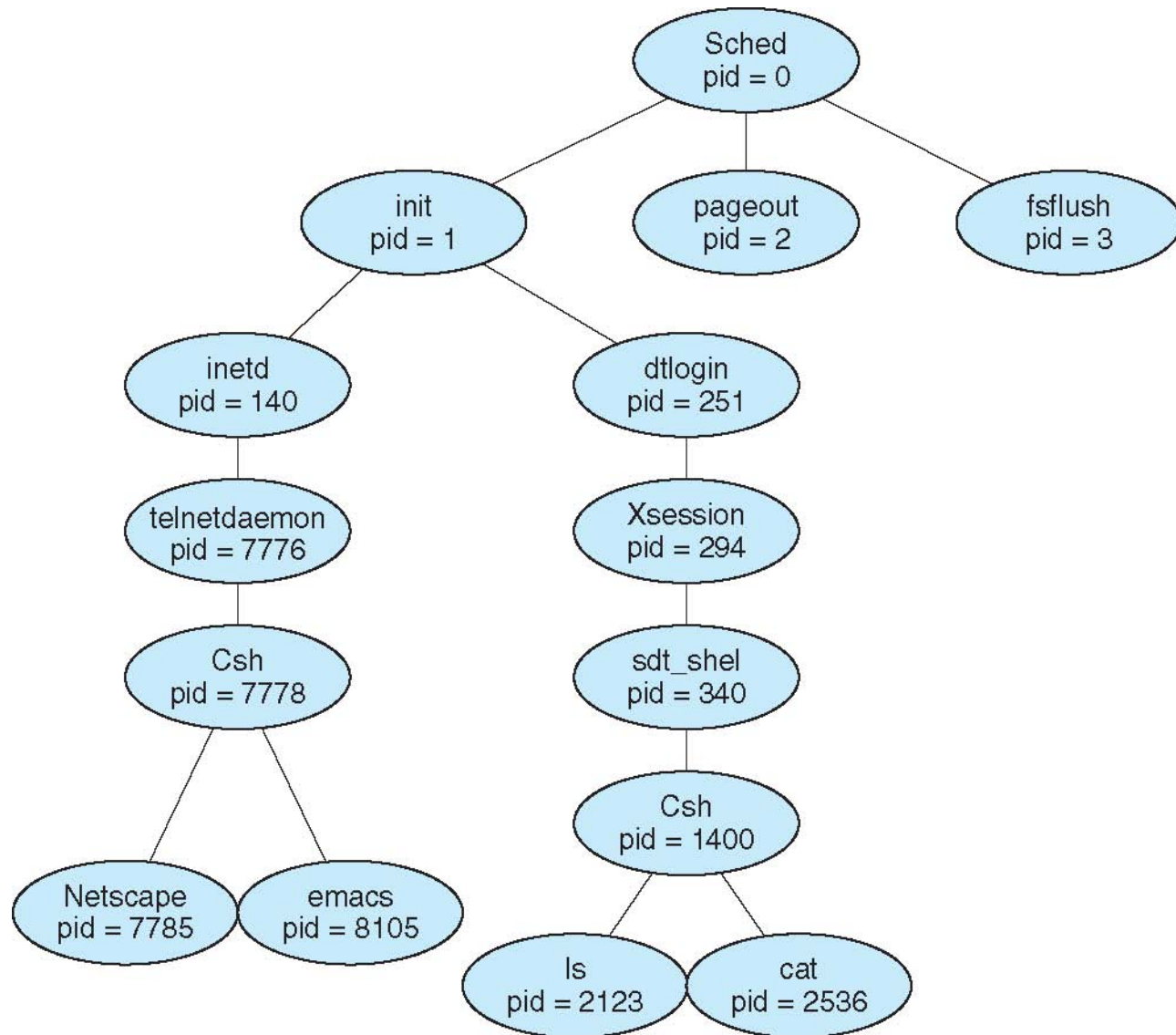
Zombie Process

- İşini bitiren bir process parent'ı waiti call'unu çağırana kadar zombie olarak isimlendirilir.
- Bir işlem sonlandırıldığında tüm belleğini ve onunla ilişkili kaynakları sisteme geri verir böylece kaynaklar diğer süreçler tarafından kullanılır.
- Ancak bu process parent tarafından bekleniyorsa bu processin PCB girdisinin sistemde olması gerekmektedir. Yani bu durumda her kaynağı silemedik!
- Parent wait call'unu gerçekleştirdikten sonra artık zombie processin PCB girdisi de silinerek temelli öldürülür.
- Uzun süre zombie olarak kalan processler hata olarak nitelendirilir.

Orphan Process

- Bir **orphan** process parenti ölüp, kendisi çalışmaya devam eden bir processdir.
- Herhangi bir orphan process başka bir process (*init* process) tarafından evlat edilir.
- Teknik olarak init processinin yeni parent'ı olsa da hala **orphan** olarak isimlendirilir.

Solaris'te Tipik Bir İşlem Ağacı



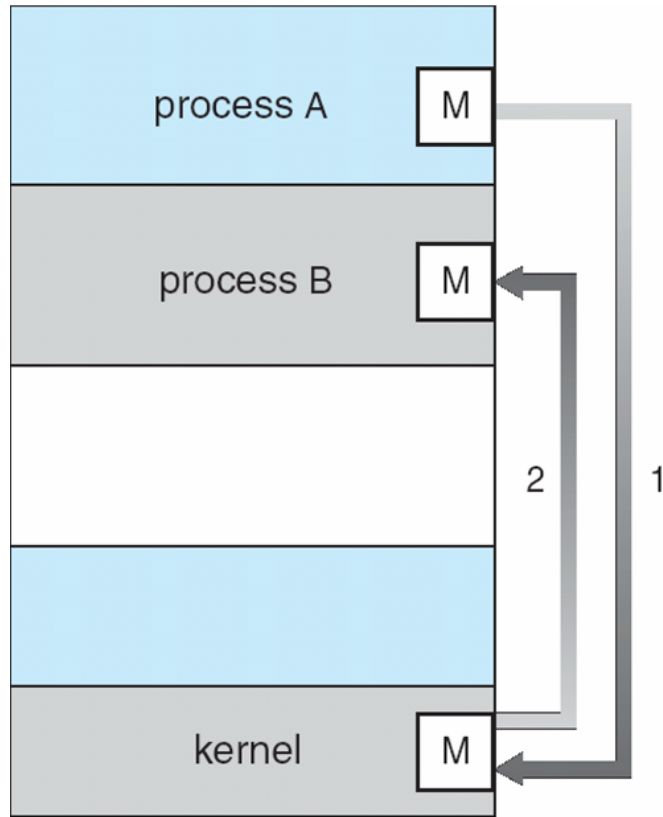
Process Sonlandırma

- **Process Termination**
- İşlemler son komutlarını çalıştırdıktan sonra işletim sistemine kendilerini silmelerini isterler (**exit**)
 - İşlemin dönüş değeri çocuktan ana işleme döndürülür (**wait** ile)
 - İşlemin kaynakları işletim sistemi tarafından geri alınır
- Ana işlem çocuk işlemlerin çalışmasını sonlandırabilir (**abort**)
 - Çocuk kendine ayrılan kaynakları aşmışsa
 - Çocuk işleme atanan göreve artık ihtiyaç yoksa
 - Eğer ana işlem sonlandırılıyorsa
 - Bazı işletim sistemleri ana işlem sonlandırıldıktan sonra çocuklarının çalışmaya devam etmesine izin vermez
 - **Peşpeşe sonlandırma (cascading termination)** ile tüm çocuk işlemler sonlandırılır

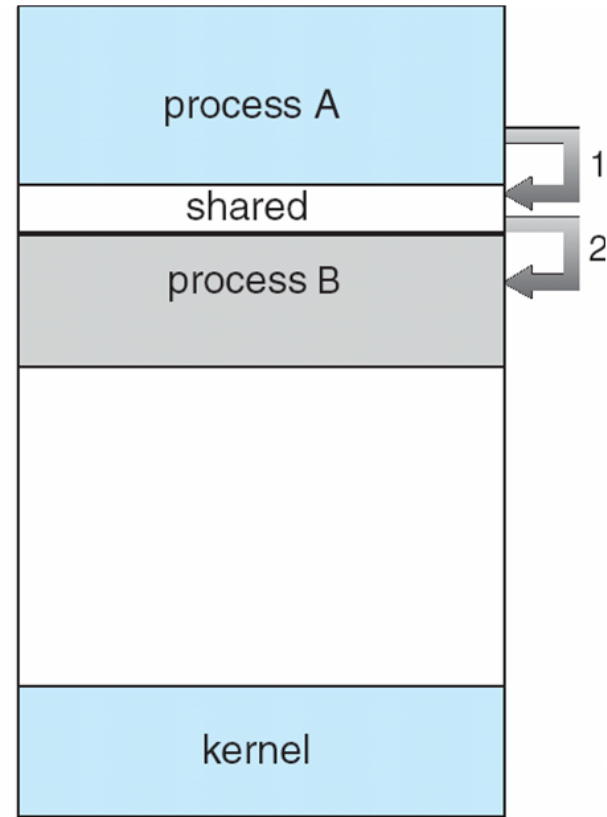
İşlemler Arası İletişim

- Interprocess Communication
- Bir sistemdeki işlemler **bağımsız (independent)** ya da **işbirliği yapıyor (cooperating)** olabilir
- İşbirliği yapan işlemler birbirlerini etkileyebilirler veya veri paylaşabilirler
- Neden işlemler işbirliği yapar?
 - Bilgi paylaşımı
 - İşlem hızını arttırmak
 - Modülerlik sağlamak
- İşbirliği yapan işlemler **işlemler arası iletişime (interprocess communication - IPC)** ihtiyaç duyar
- IPC'nin iki modeli
 - **Paylaşımlı bellek** (shared memory)
 - **Mesaj gönderme** (message passing)

İletişim Modelleri



(a)

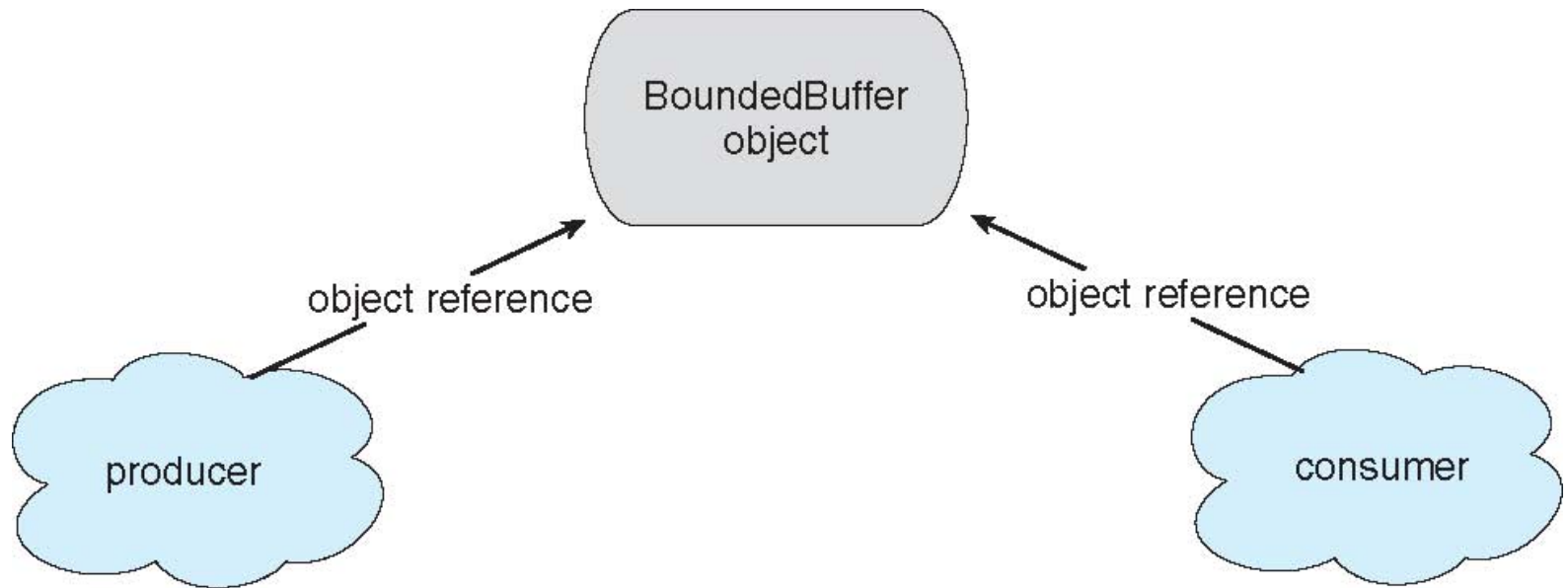


(b)

Üretici-Tüketici Problemi (shared memory için)

- Producer-Consumer Problem
- İşbirliği yapan işlemler için problem örneği: **üretici (producer)** işlem bilgi üretirken, **tüketici (consumer)** işlem üretilen bilgiyi tüketir
 - **sınırsız tampon bellek** (*unbounded-buffer*): tampon bellek için herhangi bir pratik sınır getirmez
 - **sınırlı tampon bellek** (*bounded-buffer*): sınırlı boyutta bir tampon bellek kullanıldığını varsayar

Java'da Paylaşımlı Bellek Simülasyonu



Sınırlı Tampon Bellek Çözümü - Arayüz

```
public interface Buffer <E>
{
    // Producers call this method
    public void insert(E item);

    // Consumers call this method
    public E remove();
}
```

Sınırlı Tampon Bellek – Üretici

```
// Producers call this method
public void insert(E item) {
    while (count == BUFFER_SIZE)
        ; // do nothing -- no free space

    // add an item to the buffer
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
    ++count;
}
```

Sınırlı Tampon Bellek – Tüketici

```
// Consumers call this method
public E remove() {
    E item;

    while (count == 0)
        ; // do nothing -- nothing to consume

    // remove an item from the buffer
    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    --count;

    return item;
}
```

İşlemler Arası İletişim – Mesaj Gönderme

- İşlemlerin iletişim kurması ve eylemlerini senkronize etmelerini sağlayan mekanizma
- Mesaj sistemi – işlemler birbirleriyle paylaşımlı değişkenlere bağlı kalmaksızın haberleşir
- IPC mekanizması iki işlemi sağlar:
 - **send** (*mesaj gönderme*) – mesaj boyutu sabit ya da değişken olabilir
 - **receive** (*mesaj alma*)
- Eğer P ve Q işlemleri iletişim kurmak isterse:
 - birbirleri arasında bir **iletişim bağlantısı (communication link)** sağlamalıdır
 - send/receive kullanarak mesajlaşmalıdır.
- İletişim bağlantısının gerçekleştirimi
 - fiziksel (e.g., paylaşımlı hafıza, donanım hattı)
 - mantıksal (e.g., mantıksal özellikler)

Gerçekleştirim Soruları

- Bağlantılar nasıl sağlanır?
- Bir bağlantı ikiden fazla işlemle ilişkilendirilmeli midir?
- İletişim kuran işlemlerin her bir çifti arasında kaç tane bağlantı olabilir?
- Bağlantının kapasitesi nedir?
- Bağlantının desteklediği mesajların boyutu sabit mi yoksa değişken midir?
- Bağlantı çift yönlü mü (bi-directional) yoksa tek yönlü (unidirectional) müdür?

Direk İletişim

- İşlemler birbirlerini açıkça isimlendirmelidir:
 - **send (P, message)** – P işlemine bir mesaj gönder
 - **receive(Q, message)** – Q işleminden bir mesaj al
- İletişim bağlantısının özellikleri
 - Bağlantılar otomatik olarak sağlanır
 - Bir bağlantı sadece bir çift işlemci arasında oluşturulur
 - Her bir çift için sadece bir bağlantı oluşturulur
 - Bağlantı tek yönlü olabilir, ama genellikle çift yönlüdür

Dolaylı İletişim

- Mesajlar **posta kutusuna (messagebox)** yönlendirilir ve post kutusundan alınır
 - Posta kutusu yerine **port** terimi de kullanılır
 - Her bir posta kutusu özgün bir ada sahiptir (**unique id**)
 - İşlemler sadece bir post kutusunu paylaşıyor ise iletişime geçebilir
- İletişim bağlantısının özellikleri
 - Bağlantı sadece işlemler ortak bir posta kutusunu paylaşıyor ise oluşturulur
 - İki'den fazla işlem tek bir posta kutusu ile ilişkilendirilebilir
 - Bir işlem çifti birden fazla iletişim bağlantısına sahip olabilir
 - Bağlantı tek yönlü veya çift yönlü olabilir

Dolaylı İletişim (devam)

- İşlemler (İşletim sisteminin tek bir posta kutusu varsa)
 - yeni bir posta kutusu oluşturma
 - posta kutusu aracılığıyla mesaj gönderme veya alma
 - posta kutusunun yok edilmesi
- Temel bileşenler:
 - send(A, message)** – A posta kutusuna bir mesaj gönder
 - receive(A, message)** – A posta kutusundan bir mesaj al

Posta Kutusu Paylaşımı

- Posta kutusu paylaşımı
 - P_1 , P_2 , ve P_3 A posta kutusunu paylaşıyor
 - P_1 , mesaj gönderir; P_2 ve P_3 alır
 - Mesajı kim alır?
- Çözümler
 - Bir bağlantının en fazla iki işlem ile ilişkilendirilmesine izin ver
 - Herhangi bir anda sadece bir işlemin mesaj almasına izin ver
 - Sistemin herhangi bir alıcıyı seçmesine izin ver. Gönderici kimin mesajı aldığı konusunda bilgilendirilir

Senkronizasyon

- **Synchronization**
- Mesaj gönderme **bloklanan (blocking)** veya **bloklanmayan (non-blocking)** şekilde gerçekleşebilir
- Bloklanan mesajlaşma senkrondur (synchronous)
 - **Blocking**, alıcı taraf mesajı alana kadar, gönderici taraf bloklanır
 - **Blocking**, gönderici taraf mesajı gönderene kadar, alıcı taraf bloklanır
- Bloklanmayan mesajlaşma asenkrondur (asynchronous)
 - **Non-Blocking**, gönderici taraf mesajı gönderdikten sonra beklemeksizin çalışmaya devam eder
 - **Non-Blocking**, alıcı taraf ya geçerli bir mesaj alır ya da null mesaj olarak çalışmaya devam eder

Tampon Bellek Kullanımı (Buffering)

- Bağlantı ile ilişkilendirilen kuyruk üç farklı şekilde gerçekleştirilebilir
 1. **Sıfır kapasite** – 0 mesaj
Gönderici alıcıyı beklemelidir: randevu (rendezvous)
 2. **Sınırlı kapasite** – n mesajı alacak şekilde sınırlı kapasite
Gönderici bağlantı kapasitesi dolu ise beklemelidir
 3. **Sınırsız kapasite**
Gönderici asla beklemez

IPC Sistem Örnekleri - POSIX

- POSIX Paylaşımlı Hafıza

- İşlem öncelikle paylaşımlı hafıza segmentini oluşturur

```
segment id = shmget(IPC_PRIVATE, size, S_IRUSR | S_IWUSR);
```

- Paylaşımlı belleğe erişmek isteyen işlem, bu paylaşım alanı ile bağlantı kurmalıdır

```
shared memory = (char *) shmat(id, NULL, 0);
```

- Şimdi işlem paylaşımlı hafızaya yazabilir

```
sprintf(shared memory, "Writing to shared memory");
```

- Paylaşım alanı ile yapacak işi kalmayan işlem, bu paylaşım alanı ile bağlantısını sonlandırmalıdır

```
shmdt(shared memory);
```


IPC Sistem Örnekleri - Mach

- Mach iletişimi mesaj tabanlıdır
 - Sistem çağrıları dahi sisteme mesaj olarak gönderilir
 - Her bir işlem oluşturulduğunda iki posta kutusuna sahiptir - **Kernel** and **Notify**
 - Mesaj transferi için sadece üç sistem çağrısı kullanılır
`msg_send()`, `msg_receive()`, `msg_rpc()`
 - İletişim kurmak için gereken posta kutuları aşağıdaki komutla oluşturulabilir
`port_allocate()`

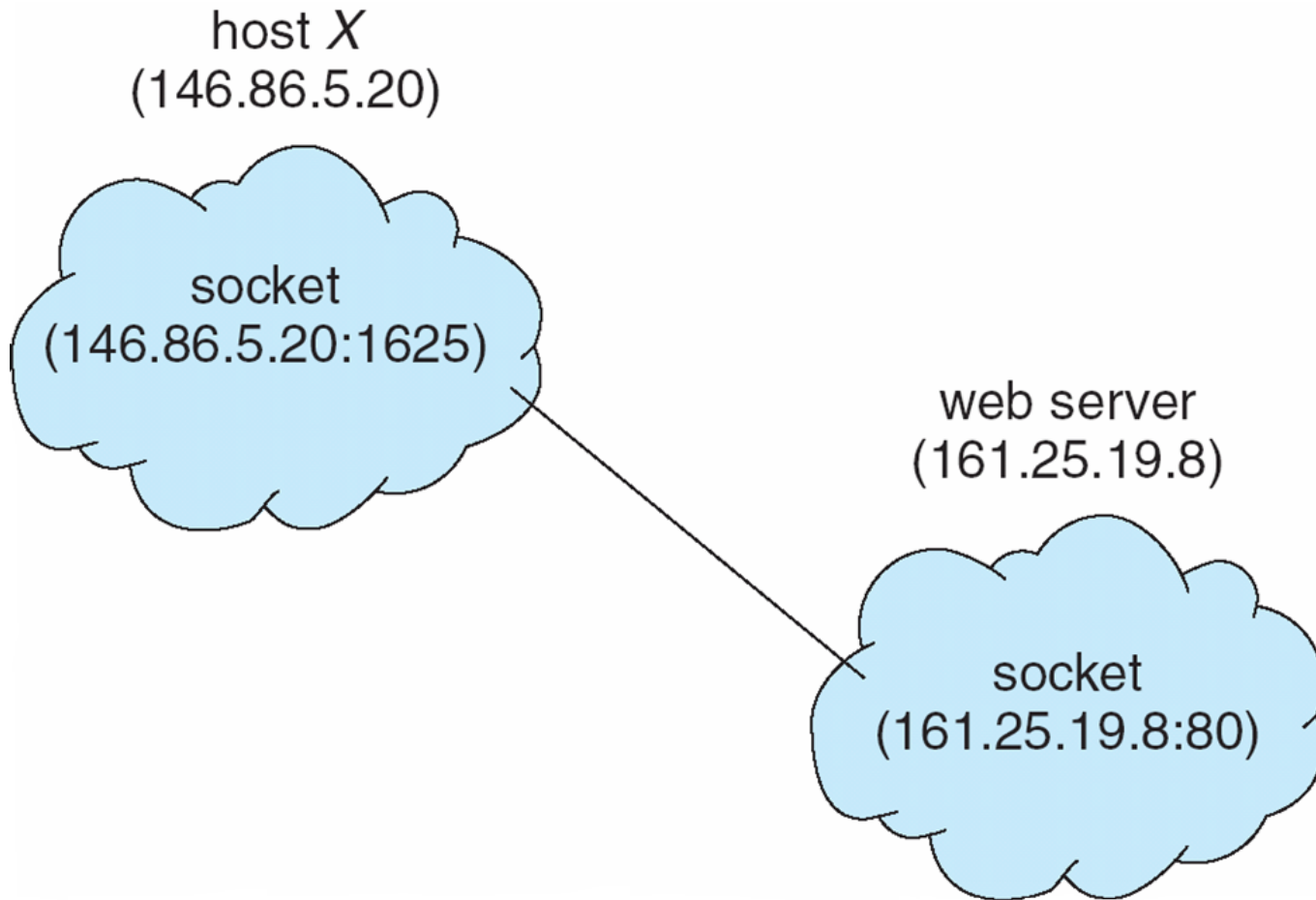
İstemci-Sunucu Sistemlerinde İletişim

- **Soketler** (Sockets)
- **Uzak Prosedür Çağrılar** (Remote Procedure Calls)
- **Uzak Metot Çağırma** (Remote Method Invocation)

Soketler

- Soketler iletişim amaçlı bağlantı noktaları olarak tanımlanır
- IP adresi ve portun birleşimidir
- **161.25.19.8:1625** soketi **161.25.19.8** IP'li makinanın **1625** numaralı portuna arşılık gelmektedir
- İletişim bir çift soket arasında gerçekleşir

Soket İletişimi



Java'da Soket İletişimi - Sunucu

```
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'da Soket İletişimi - İstemci

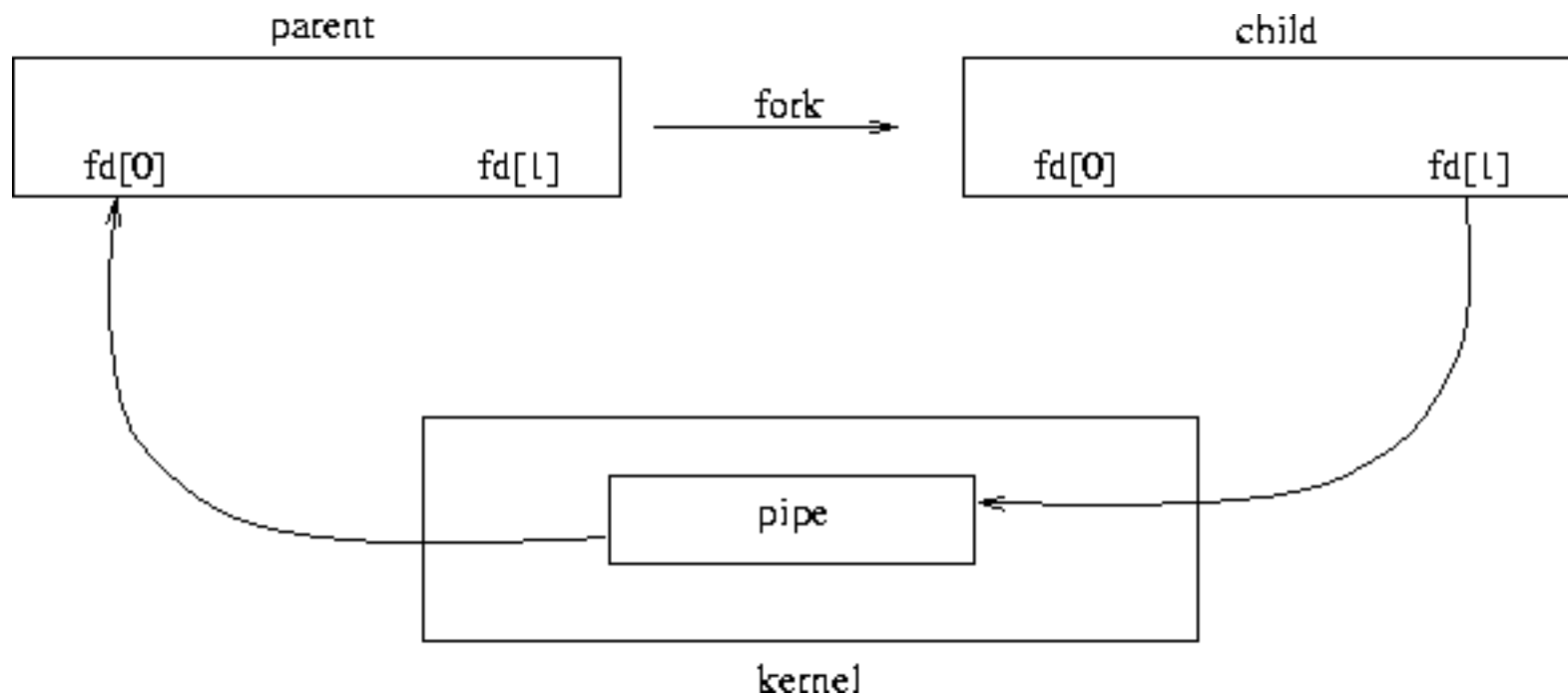
```
public class DateClient
{
    public static void main(String[] args) {
        try {
            //make connection to server socket
            Socket sock = new Socket("127.0.0.1",6013);

            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);
        }
    }
}
```

Pipe



Ordinary Pipe

- Bir **pipe**, processler arası iletişimi sağlamak için çift yönlü ama bir anda tek yönü kullanılabilen bir FIFO'dur.
- **Implementationları** genelde işletim sisteminin I/O sistemleri ile entegredir.
- Tipik olarak parent process pipe oluşturur, fork ile bu pipe çocuğu ile paylaşır.
- Unix sistemlerinde `pipe()` system çağrı ile oluşturulur.

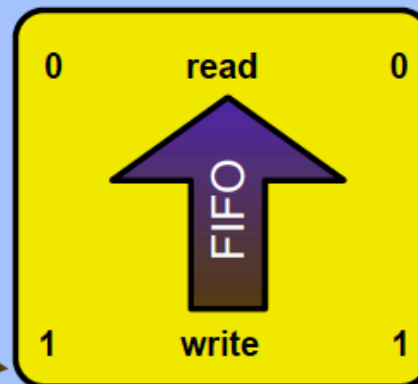
Using pipe()



Parent Process

```
int pfd[2];  
pipe(pfd);  
// pfd[0] = 3  
// pfd[1] = 4
```

Descriptors	
0	stdin
1	stdout
2	stderr
3	pipe read
4	pipe write



(1)

As a result of the **pipe()** system call, a pipe object is created. A pipe is a FIFO buffer that can be used for inter process communication (IPC).

(2)

As a result of the **pipe()** system call, the elements of the array **pfd** are assigned descriptor numbers to the created pipe.

Now a single process can write data to, and later read data from the pipe ...

User Space

Kernel Space

Using pipe() together with fork()

User Space

Parent Process

```
int pfd[2];  
pipe(pfd);  
// pfd[0] = 3  
// pfd[1] = 4  
fork();  
close(pfd[0]);
```

The parent can close the read descriptor to the pipe.

Now, the parent can act as a single producer and the child as a single consumer of data through the pipe using the read() and write() system calls - just as if it was a file.

Child Process

The child can close the write descriptor to the pipe.

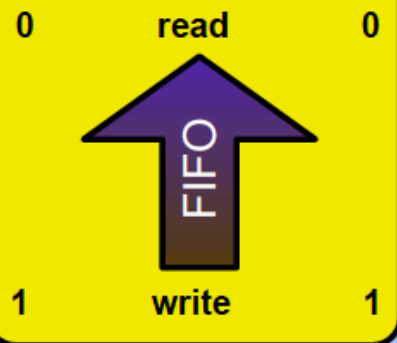
```
close(pfd[1]);
```

Descriptors

	Descriptors
0	stdin
1	stdout
2	stderr
3	pipe read
4	pipe write

Descriptors

	Descriptors
0	stdin
1	stdout
2	stderr
3	pipe read
4	pipe write



Kernel Space

SIGPIPE

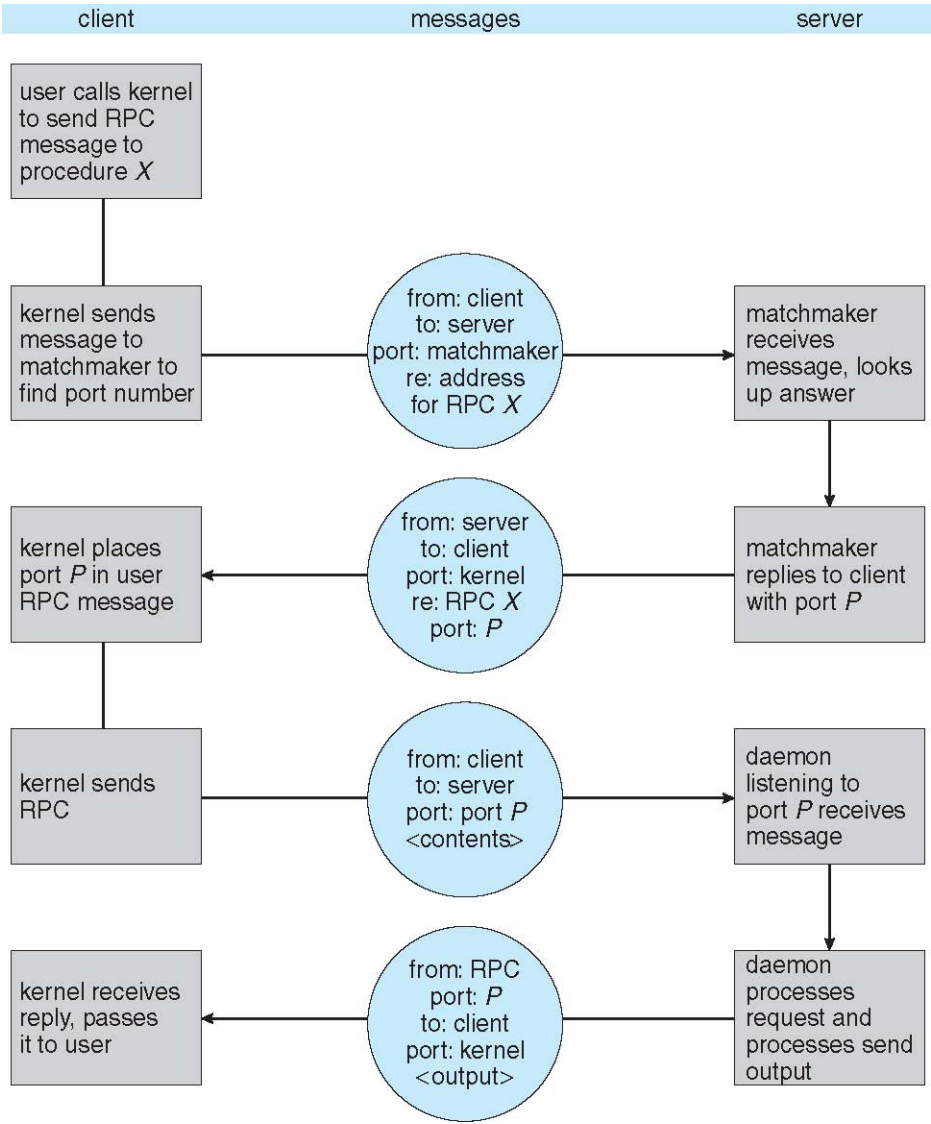
The SIGPIPE signal is used to notify a producer (writer) when there is no longer a consumer (reader) attached to a pipe.

Attempt	Conditions	Result
Read	Empty pipe, writer attached	Reader blocked
Write	Full pipe, reader attached	Writer blocked
Read	Empty pipe, no writer attached	EOF returned
Write	No readers attached	SIGPIPE

Uzak Prosedür Çağrılar (RPC)

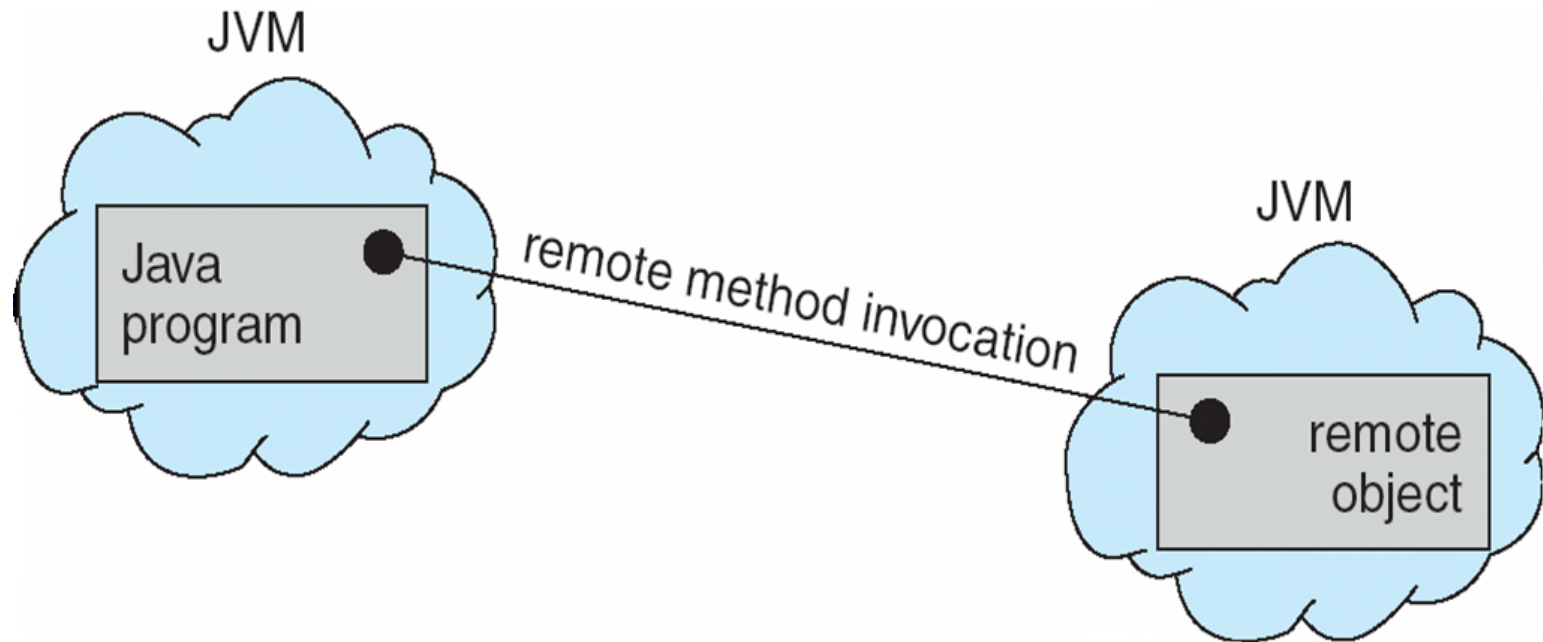
- Remote procedure call (RPC)
- Ağa bağlı sistemlerdeki işlemler arasında metot (prosedür, fonksiyon) çağrımını soyutlar
- **Stub** – sunucudaki asıl metot için istemci tarafında bulunan vekildir
- İstemci tarafı sunucuyu konumlandırır ve metot parametrelerini paketler
- Sunucu tarafı mesajı alır, parametreleri açığa çıkarır ve sunucudaki metodu çalıştırır

RPC'nin Çalıştırılması

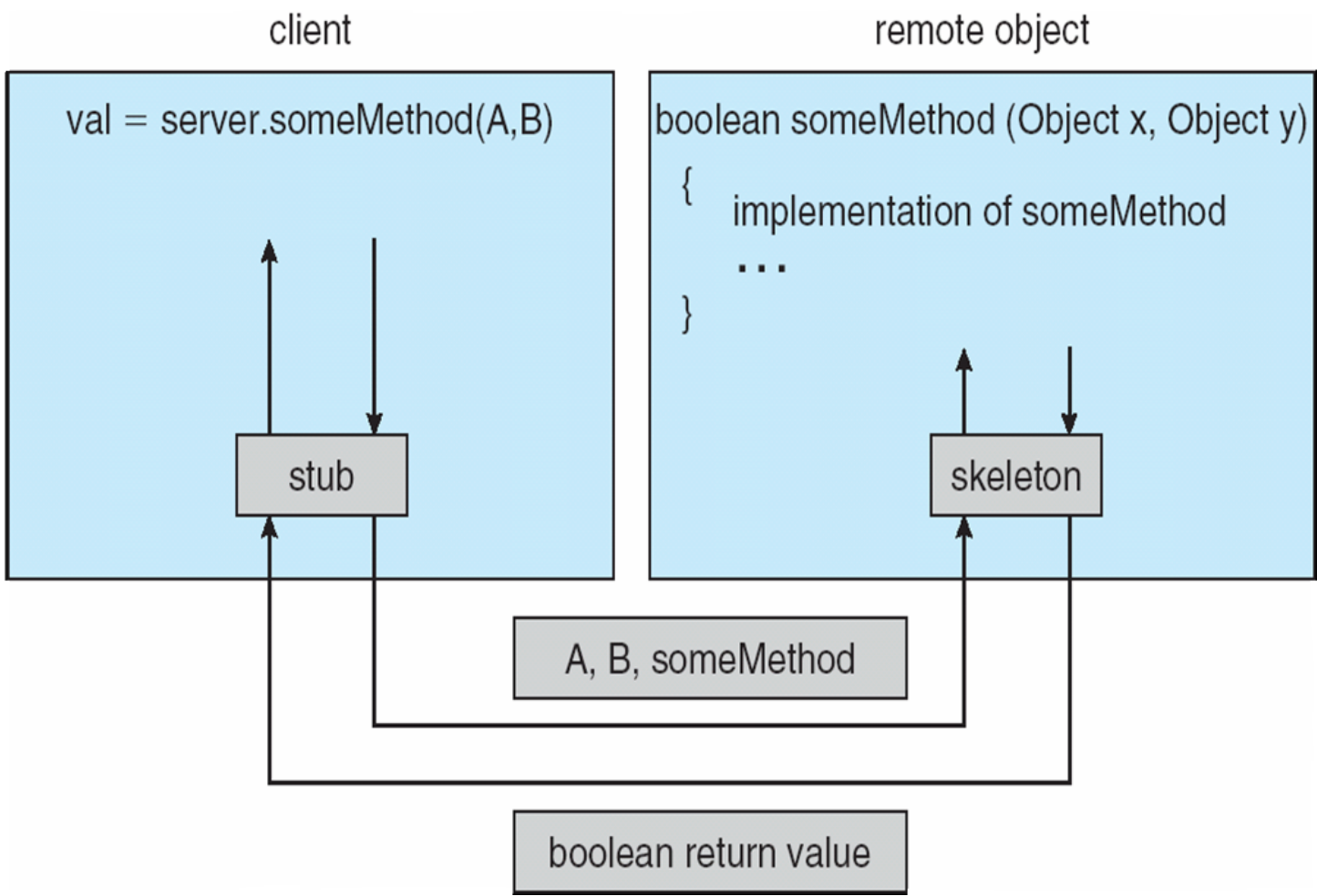


Uzak Metot Çağırma (RMI)

- **Remote Method Invocation (RMI)**
- RPC'ye benzer Java mekanizmasıdır
- RMI, bir Java programının, uzak bir nesnede bulunan bir metodu çağırmasını sağlar



Parametrelerin Paketlenmesi



RMI Örneği - Arayüz

```
public interface RemoteDate extends Remote
{
    public abstract Date getDate() throws RemoteException;
}
```

RMI Örneği – Sunucu Taraf

```
public class RemoteDateImpl extends UnicastRemoteObject
    implements RemoteDate
{
    public RemoteDateImpl() throws RemoteException { }

    public Date getDate() throws RemoteException {
        return new Date();
    }

    public static void main(String[] args) {
        try {
            RemoteDate dateServer = new RemoteDateImpl();

            // Bind this object instance to the name "DateServer"
            Naming.rebind("DateServer", dateServer);
        }
        catch (Exception e) {
            System.err.println(e);
        }
    }
}
```

RMI Örneği – İstemci Taraf

```
public class RMIClient
{
    public static void main(String args[]) {
        try {
            String host = "rmi://127.0.0.1/DateServer";

            RemoteDate dateServer = (RemoteDate)Naming.lookup(host);
            System.out.println(dateServer.getDate());
        }
        catch (Exception e) {
            System.err.println(e);
        }
    }
}
```