



Chapter 4: Processes

- Process Concept
- Process Scheduling
- Operations on Processes
- Cooperating Processes
- Interprocess Communication
- Communication in Client-Server Systems





Process Concept

- An operating system executes a variety of programs:
 - Batch system – jobs
 - Time-shared systems – user programs or tasks
- Textbook uses the terms *job* and *process* almost interchangeably
- Process – a program in execution; process execution must progress in sequential fashion
- A process includes:
 - program counter
 - stack
 - data section





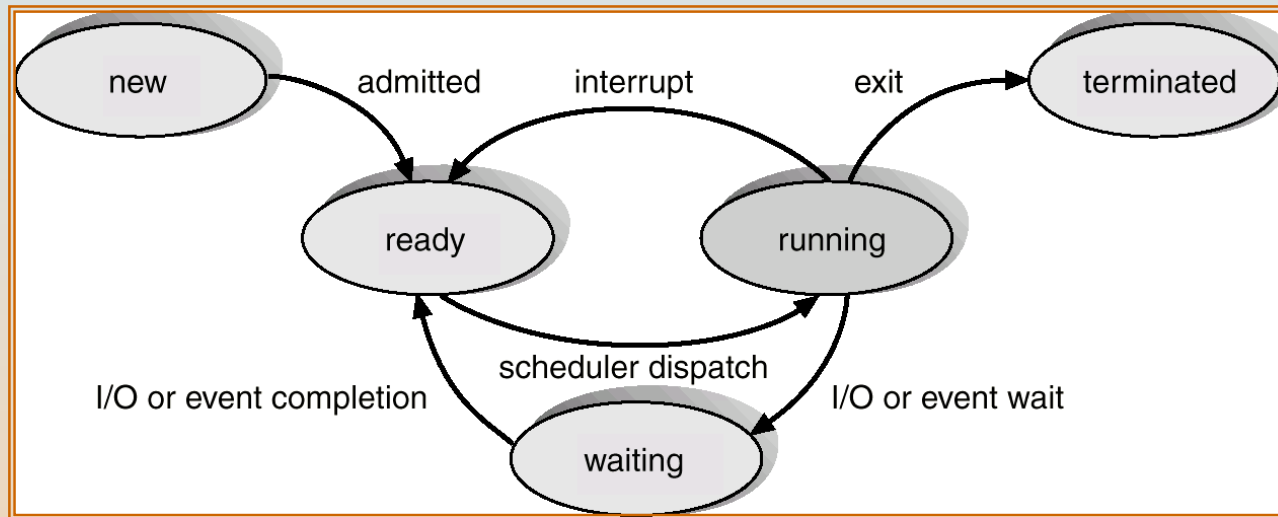
Process State

- As a process executes, it changes *state*
 - **new**: The process is being created
 - **running**: Instructions are being executed
 - **waiting**: The process is waiting for some event to occur
 - **ready**: The process is waiting to be assigned to a process
 - **terminated**: The process has finished execution





Diagram of Process State





Process Control Block (PCB)

Information associated with each process

- Process state
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information





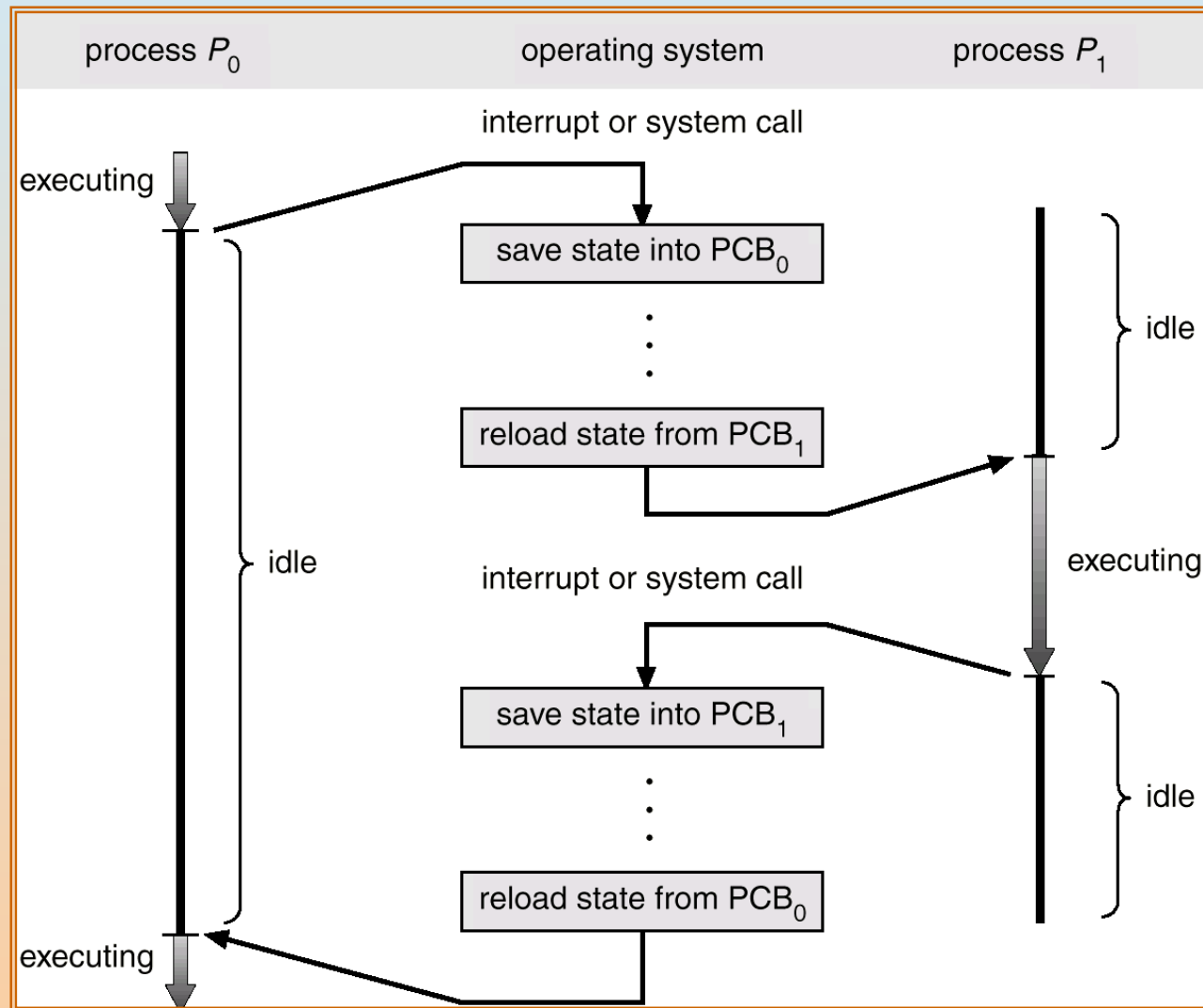
Process Control Block (PCB)

process state
process number
program counter
registers
memory limits
list of open files
...





CPU Switch From Process to Process



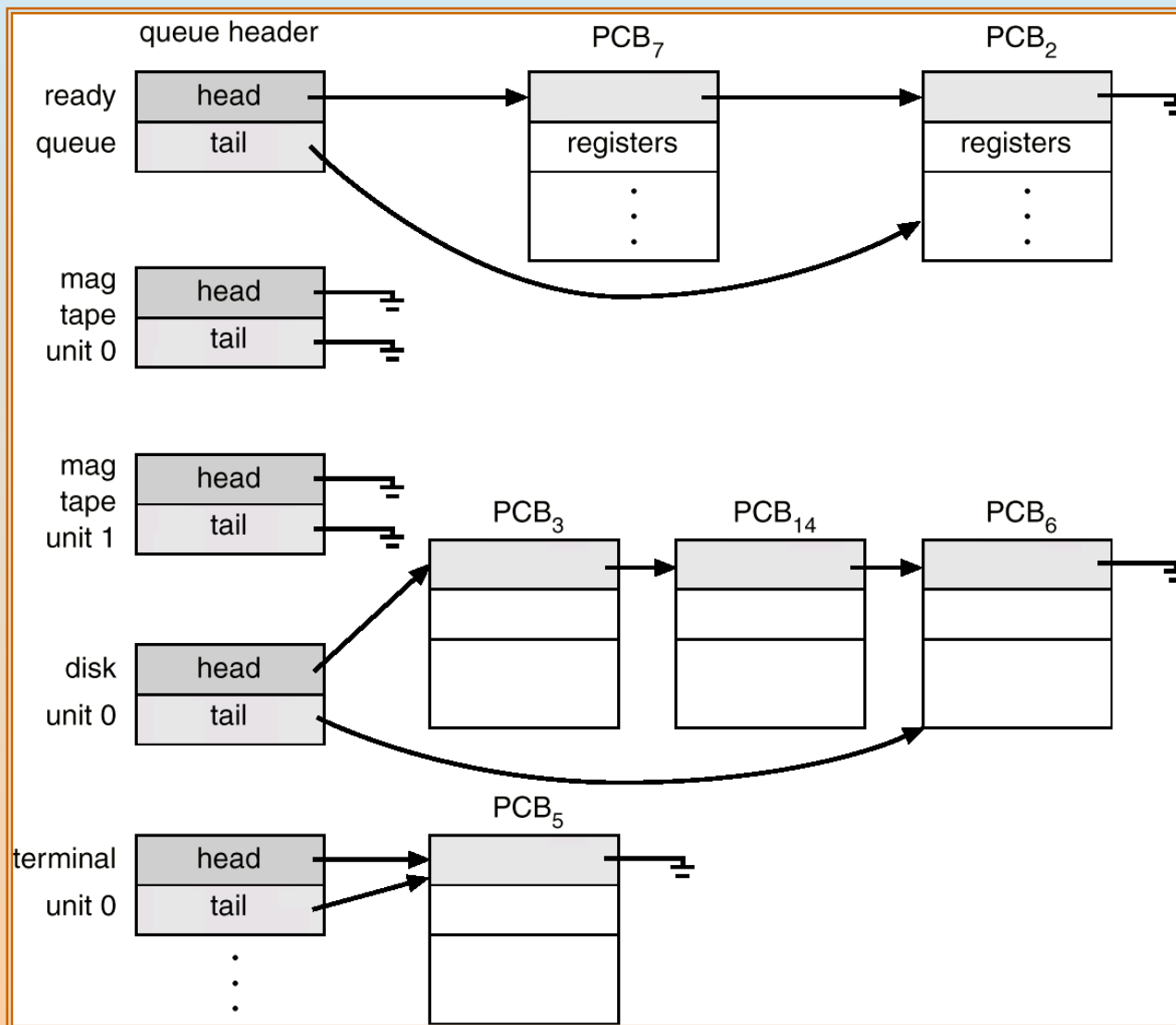


Process Scheduling Queues

- *Job queue* – set of all processes in the system
- *Ready queue* – set of all processes residing in main memory, ready and waiting to execute
- *Device queues* – set of processes waiting for an I/O device
- Process migration between the various queues

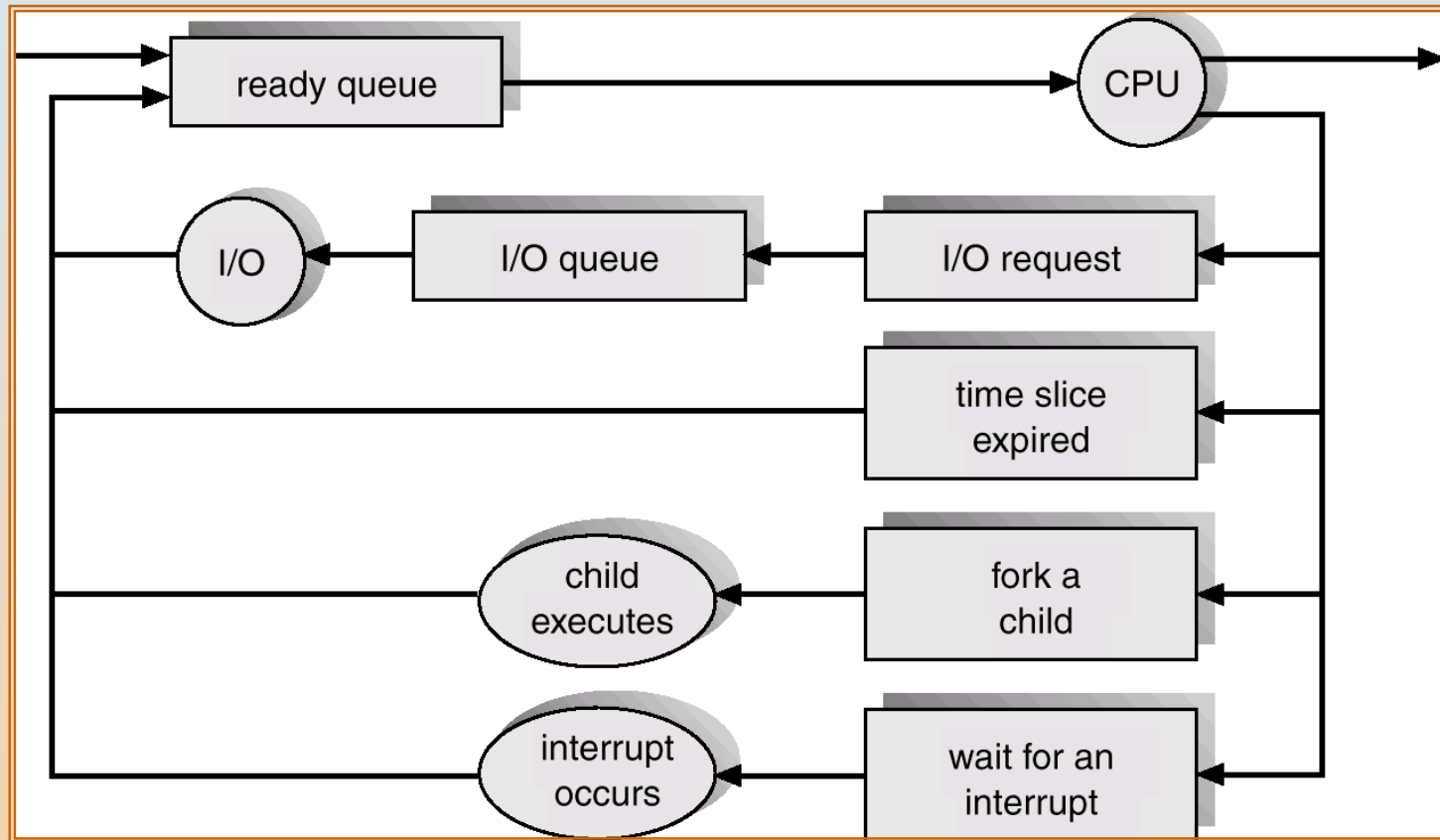


Ready Queue And Various I/O Device Queues





Representation of Process Scheduling





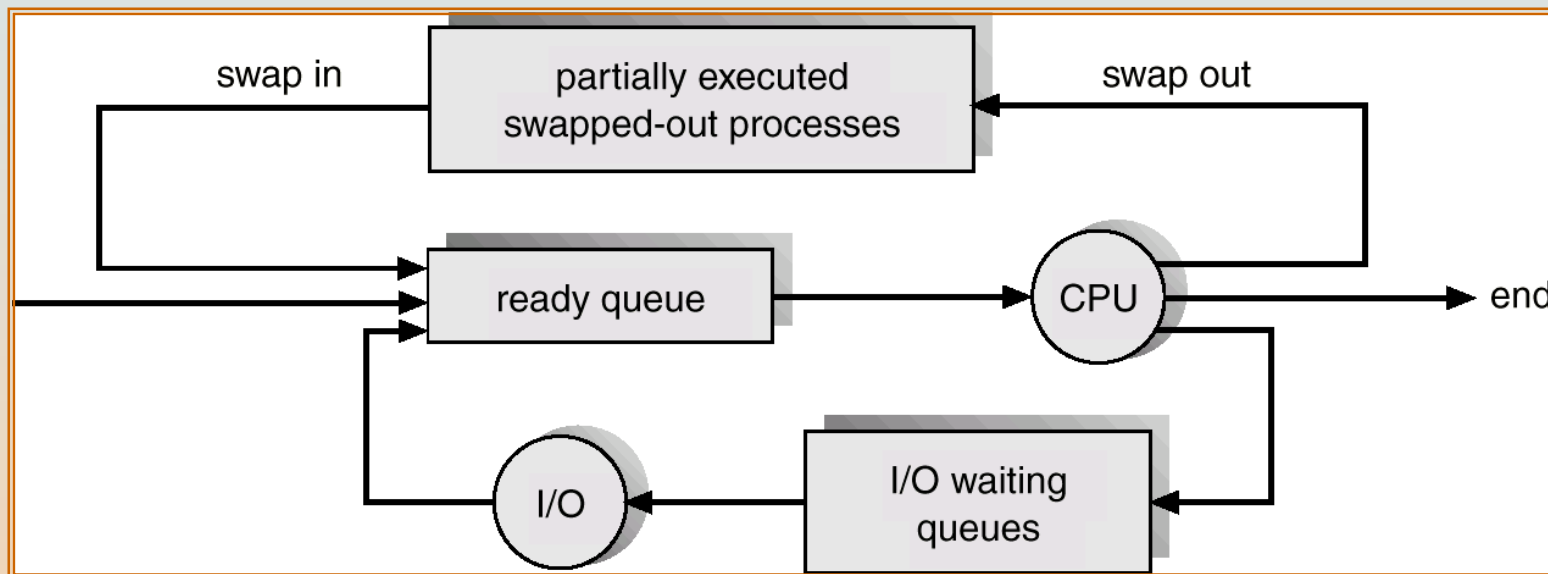
Schedulers

- *Long-term scheduler* (or job scheduler) – selects which processes should be brought into the ready queue
- *Short-term scheduler* (or CPU scheduler) – selects which process should be executed next and allocates CPU





Addition of Medium Term Scheduling





Schedulers (Cont.)

- Short-term scheduler is invoked very frequently (milliseconds) \Rightarrow (must be fast)
- Long-term scheduler is invoked very infrequently (seconds, minutes) \Rightarrow (may be slow)
- The long-term scheduler controls the *degree of multiprogramming*
- Processes can be described as either:
 - *I/O-bound process* – spends more time doing I/O than computations, many short CPU bursts
 - *CPU-bound process* – spends more time doing computations; few very long CPU bursts





Context Switch

- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process
- Context-switch time is overhead; the system does no useful work while switching
- Time dependent on hardware support





Process Creation

- Parent process create children processes, which, in turn create other processes, forming a tree of processes
- Resource sharing
 - Parent and children share all resources
 - Children share subset of parent's resources
 - Parent and child share no resources
- Execution
 - Parent and children execute concurrently
 - Parent waits until children terminate





Process Creation (Cont.)

- Address space
 - Child duplicate of parent
 - Child has a program loaded into it
- UNIX examples
 - **fork** system call creates new process
 - **exec** system call used after a **fork** to replace the process' memory space with a new program





C Program Forking Separate Process

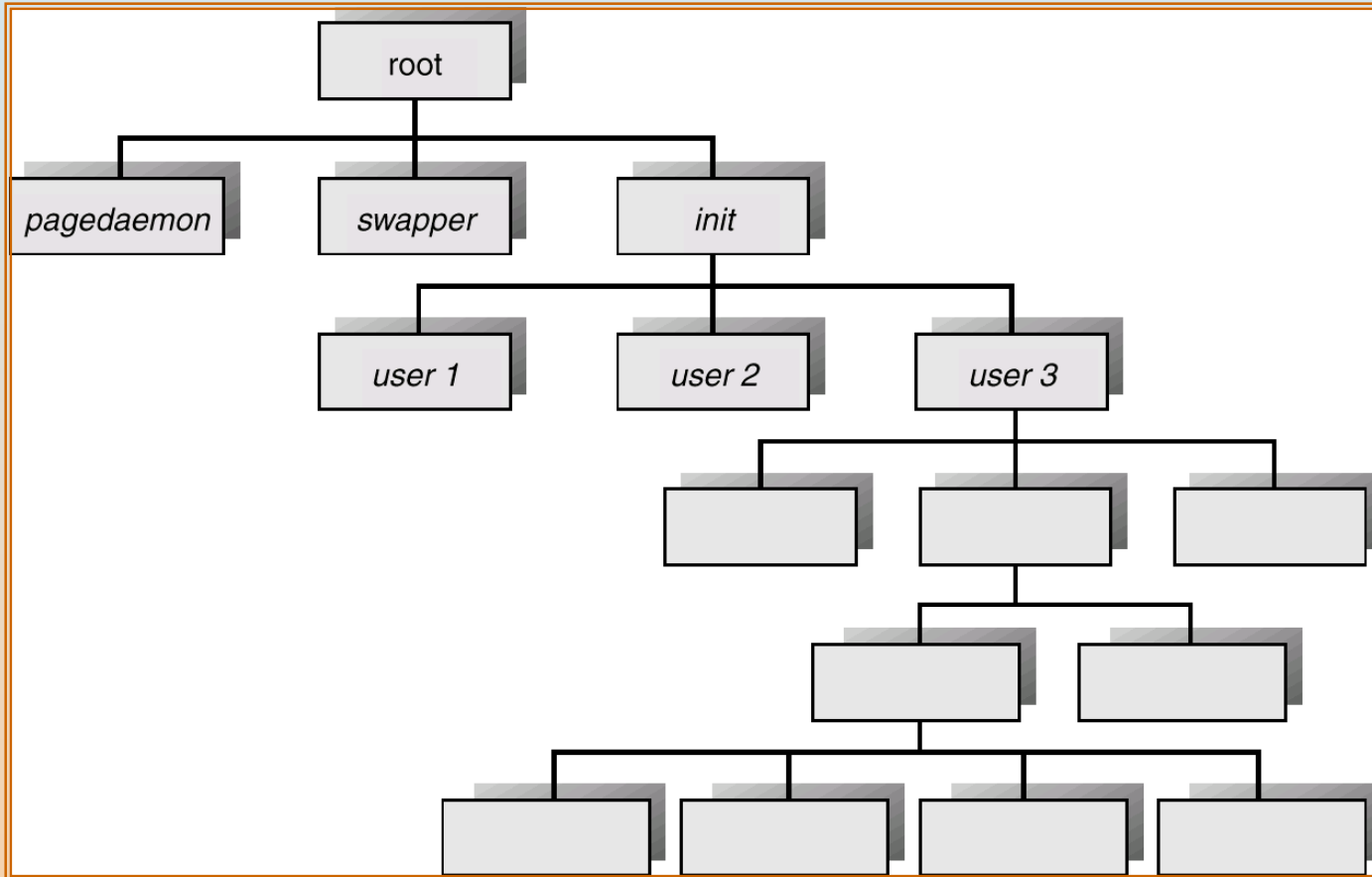
```
#include <stdio.h>
#include <unistd.h>
int main(int argc, char *argv[])
{
    int pid;

    /* 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);
    }
}
```





Processes Tree on a UNIX System





Process Termination

- Process executes last statement and asks the operating system to decide it (**exit**)
 - Output data from child to parent (via **wait**)
 - Process' resources are deallocated by operating system
- Parent may terminate execution of children processes (**abort**)
 - Child has exceeded allocated resources
 - Task assigned to child is no longer required
 - If parent is exiting
 - ▶ Some operating system do not allow child to continue if its parent terminates
 - All children terminated - *cascading termination*





Cooperating Processes

- *Independent* process cannot affect or be affected by the execution of another process
- *Cooperating* process can affect or be affected by the execution of another process
- Advantages of process cooperation
 - Information sharing
 - Computation speed-up
 - Modularity
 - Convenience





Producer-Consumer Problem

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
 - *unbounded-buffer* places no practical limit on the size of the buffer
 - *bounded-buffer* assumes that there is a fixed buffer size





Bounded-Buffer – Shared-Memory Solution

```
public interface Buffer
{
    // producers call this method
    public abstract void insert(Object item);
    // consumers call this method
    public abstract Object remove();
}
```





Bounded-Buffer – Shared Memory Solution

```
import java.util.*;

public class BoundedBuffer implements Buffer
{
    private static final int BUFFER SIZE = 5;
    private int count; // number of items in the buffer
    private int in; // points to the next free position
    private int out; // points to the next full position
    private Object[] buffer;
    public BoundedBuffer() {
        // buffer is initially empty
        count = 0;
        in = 0;
        out = 0;
        buffer = new Object[BUFFER SIZE];
    }
    // producers calls this method
    public void insert(Object item) {
        // Slide 4.24
    }
    // consumers calls this method
    public Object remove() {
        // Figure 4.25
    }
}
```





Bounded-Buffer – Insert() Method

```
public void insert(Object item) {  
    while (count == BUFFER SIZE)  
        ; // do nothing -- no free buffers  
    // add an item to the buffer  
    ++count;  
    buffer[in] = item;  
    in = (in + 1) % BUFFER SIZE;  
}
```





Bounded Buffer – Remove() Method

```
public Object remove() {  
    Object item;  
    while (count == 0)  
        ; // do nothing -- nothing to consume  
    // remove an item from the buffer  
    --count;  
    item = buffer[out];  
    out = (out + 1) % BUFFER SIZE;  
    return item;  
}
```





Interprocess Communication (IPC)

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
 - **send**(*message*) – message size fixed or variable
 - **receive**(*message*)
- If P and Q wish to communicate, they need to:
 - establish a *communication link* between them
 - exchange messages via send/receive
- Implementation of communication link
 - physical (e.g., shared memory, hardware bus)
 - logical (e.g., logical properties)





Implementation Questions

- How are links established?
- Can a link be associated with more than two processes?
- How many links can there be between every pair of communicating processes?
- What is the capacity of a link?
- Is the size of a message that the link can accommodate fixed or variable?
- Is a link unidirectional or bi-directional?





Direct Communication

- Processes must name each other explicitly:
 - **send** (P , *message*) – send a message to process P
 - **receive**(Q , *message*) – receive a message from process Q
- Properties of communication link
 - Links are established automatically
 - A link is associated with exactly one pair of communicating processes
 - Between each pair there exists exactly one link
 - The link may be unidirectional, but is usually bi-directional





Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports)
 - Each mailbox has a unique id
 - Processes can communicate only if they share a mailbox
- Properties of communication link
 - Link established only if processes share a common mailbox
 - A link may be associated with many processes
 - Each pair of processes may share several communication links
 - Link may be unidirectional or bi-directional





Indirect Communication

■ Operations

- create a new mailbox
- send and receive messages through mailbox
- destroy a mailbox

■ Primitives are defined as:

send(*A, message*) – send a message to mailbox *A*

receive(*A, message*) – receive a message from mailbox *A*





Indirect Communication

■ Mailbox sharing

- P_1 , P_2 , and P_3 share mailbox A
- P_1 sends; P_2 and P_3 receive
- Who gets the message?

■ Solutions

- Allow a link to be associated with at most two processes
- Allow only one process at a time to execute a receive operation
- Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.





Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
 - **Blocking send** has the sender block until the message is received
 - **Blocking receive** has the receiver block until a message is available
- **Non-blocking** is considered **asynchronous**
 - **Non-blocking send** has the sender send the message and continue
 - **Non-blocking receive** has the receiver receive a valid message or null





Buffering

- Queue of messages attached to the link; implemented in one of three ways
 1. Zero capacity – 0 messages
Sender must wait for receiver (rendezvous)
 2. Bounded capacity – finite length of n messages
Sender must wait if link full
 3. Unbounded capacity – infinite length
Sender never waits





Client-Server Communication

- Sockets
- Remote Procedure Calls
- Remote Method Invocation (Java)





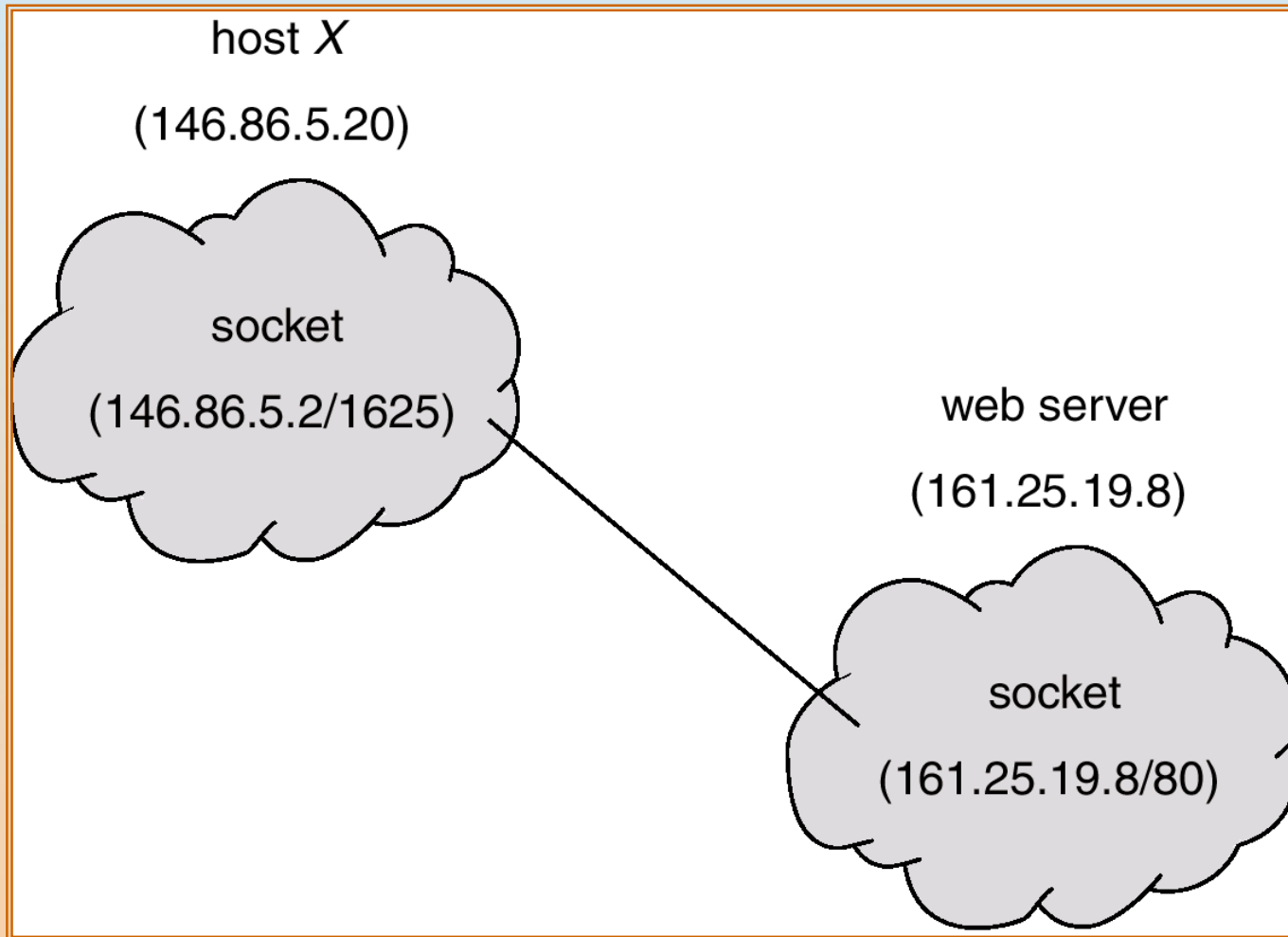
Sockets

- A socket is defined as an *endpoint for communication*
- Concatenation of IP address and port
- The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**
- Communication consists between a pair of sockets





Socket Communication





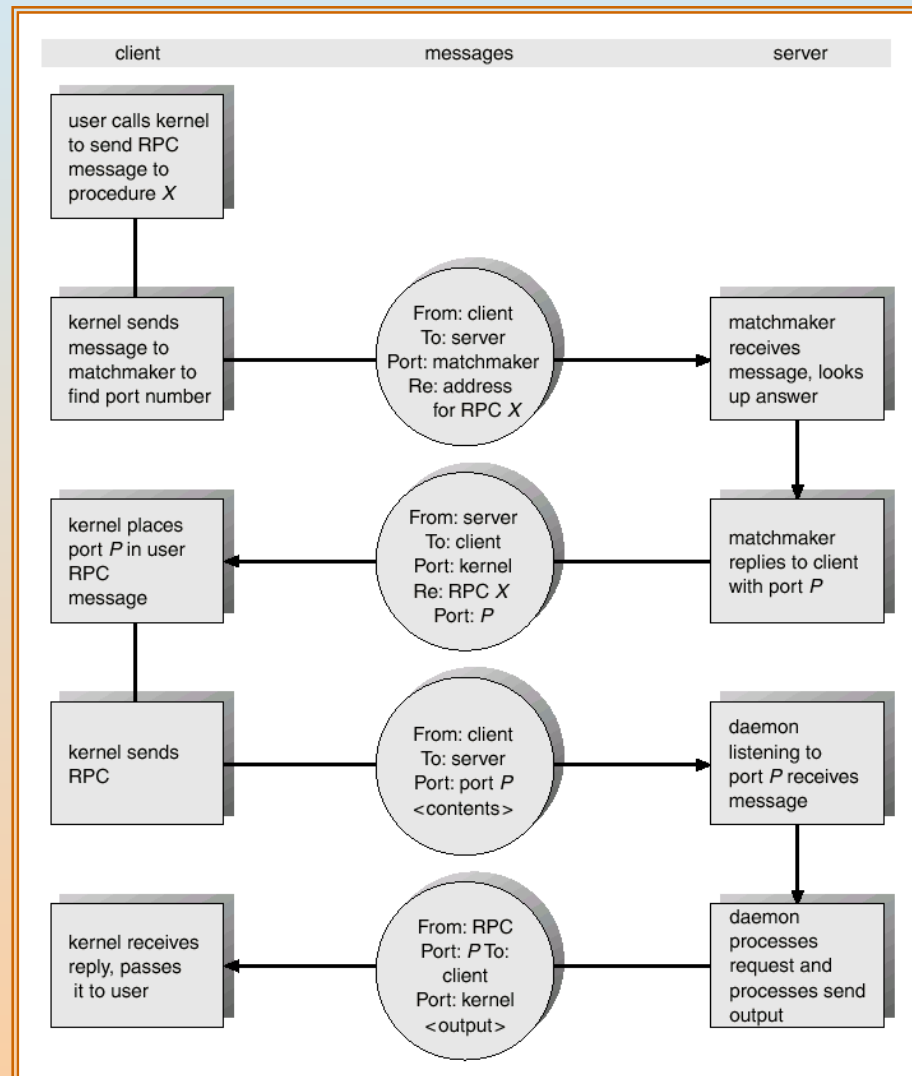
Remote Procedure Calls

- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems.
- **Stubs** – client-side proxy for the actual procedure on the server.
- The client-side stub locates the server and *marshalls* the parameters.
- The server-side stub receives this message, unpacks the marshalled parameters, and performs the procedure on the server.





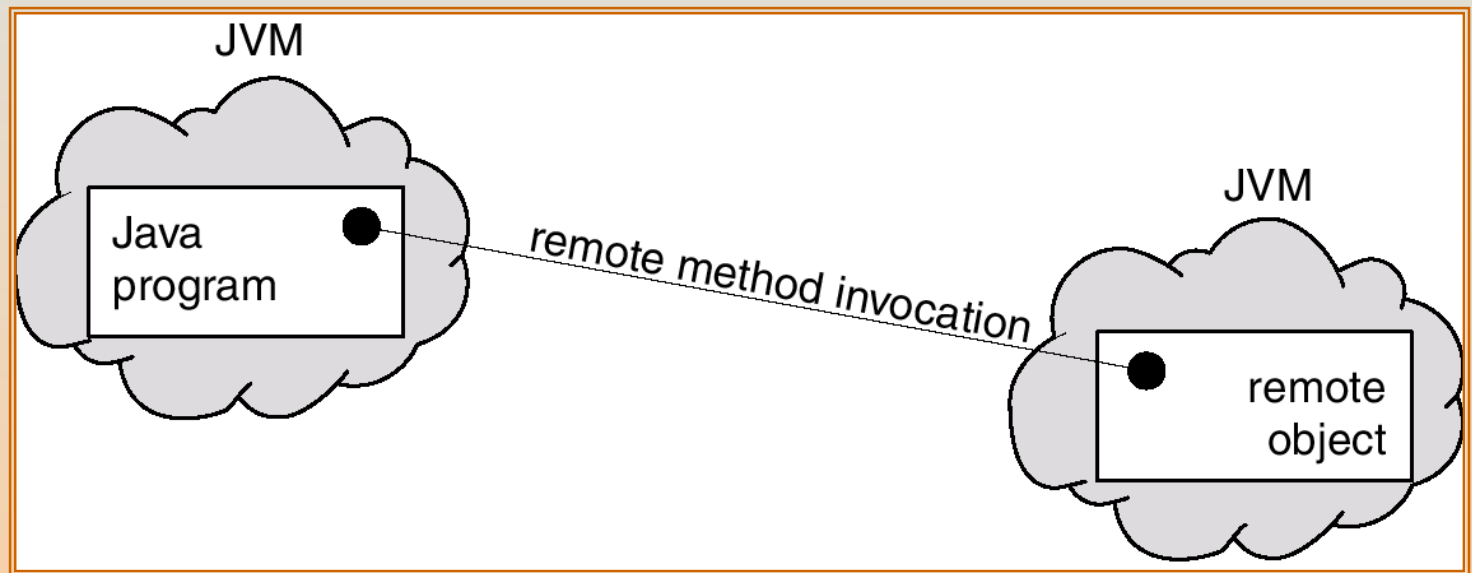
Execution of RPC





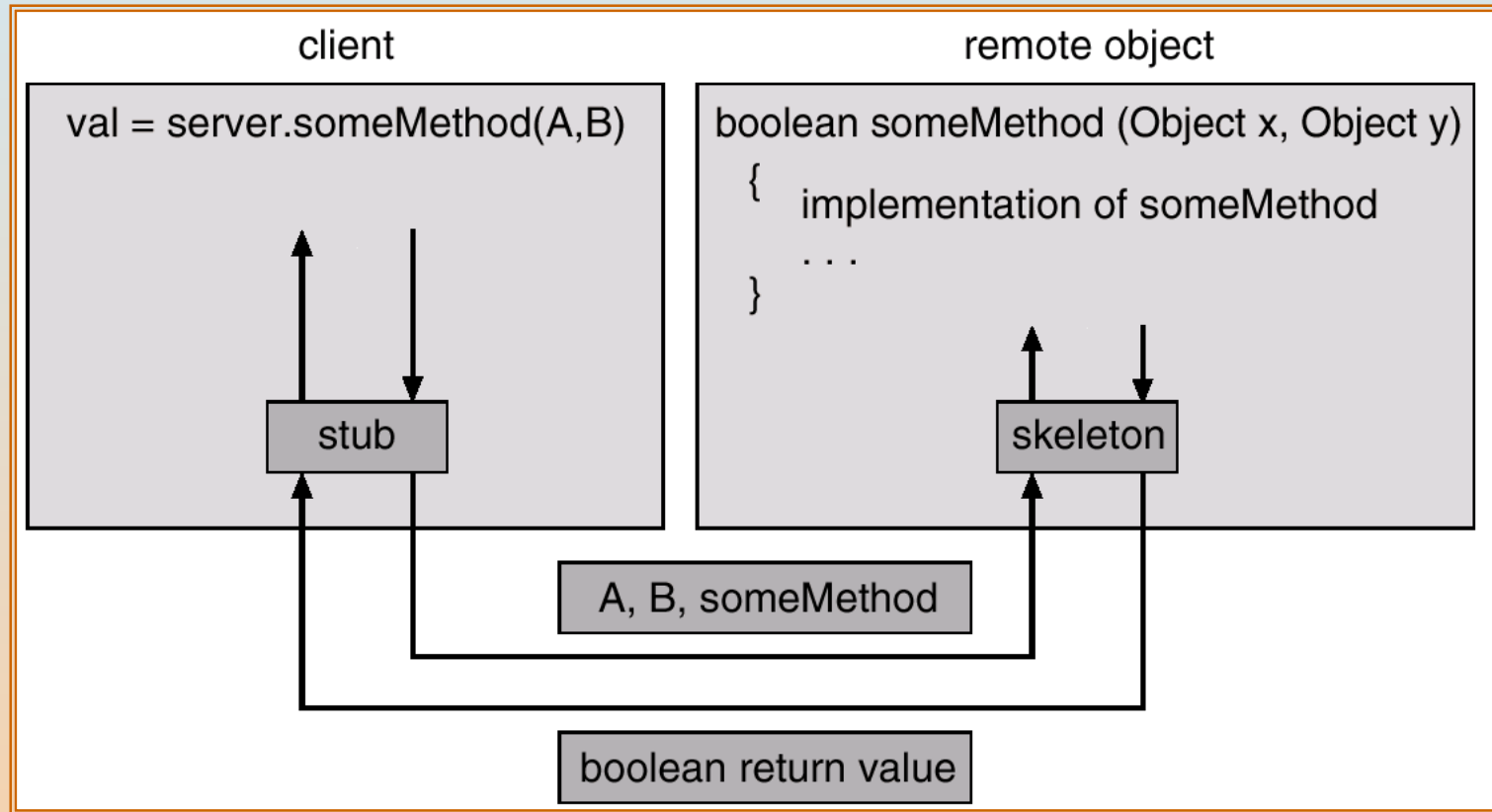
Remote Method Invocation

- Remote Method Invocation (RMI) is a Java mechanism similar to RPCs.
- RMI allows a Java program on one machine to invoke a method on a remote object.





Marshalling Parameters





Threads

- A *thread* (or *lightweight process*) is a basic unit of CPU utilization; it consists of:
 - program counter
 - register set
 - stack space
- A thread shares with its peer threads its:
 - code section
 - data section
 - operating-system resourcescollectively know as a *task*.
- A traditional or *heavyweight* process is equal to a task with one thread





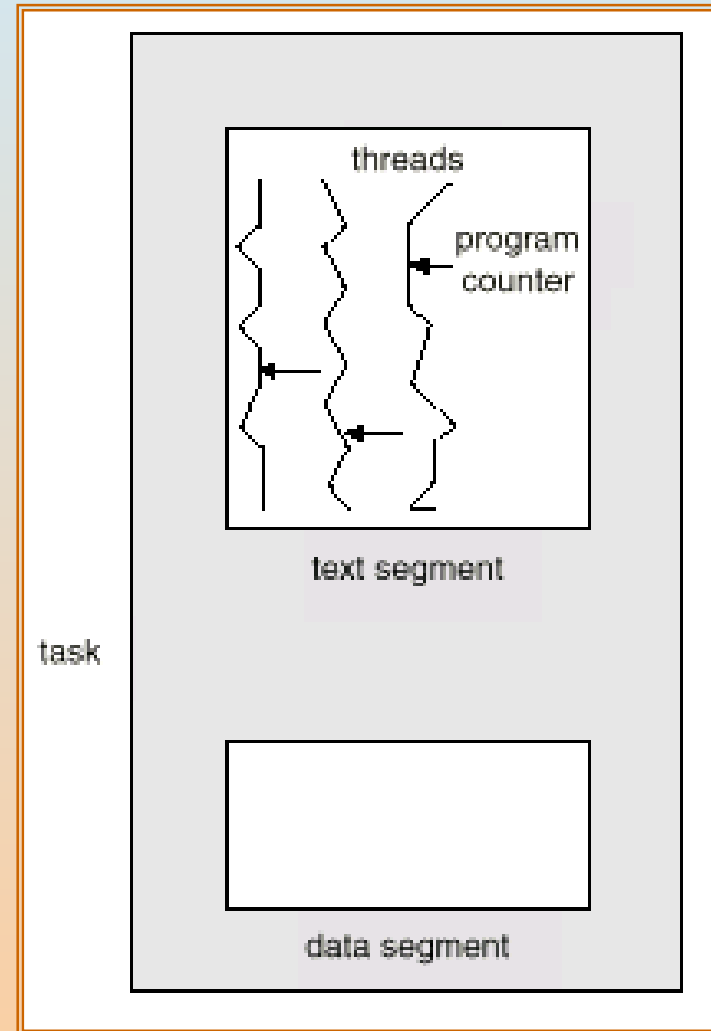
Threads (Cont.)

- In a multiple threaded task, while one server thread is blocked and waiting, a second thread in the same task can run.
 - Cooperation of multiple threads in same job confers higher throughput and improved performance.
 - Applications that require sharing a common buffer (i.e., producer-consumer) benefit from thread utilization.
- Threads provide a mechanism that allows sequential processes to make blocking system calls while also achieving parallelism.
- Kernel-supported threads (Mach and OS/2).
- User-level threads; supported above the kernel, via a set of library calls at the user level (Project Andrew from CMU).
- Hybrid approach implements both user-level and kernel-supported threads (Solaris 2).





Multiple Threads within a Task





Threads Support in Solaris 2

- Solaris 2 is a version of UNIX with support for threads at the kernel and user levels, symmetric multiprocessing, and real-time scheduling.
- LWP – intermediate level between user-level threads and kernel-level threads.
- Resource needs of thread types:
 - Kernel thread: small data structure and a stack; thread switching does not require changing memory access information – relatively fast.
 - LWP: PCB with register data, accounting and memory information,; switching between LWPs is relatively slow.
 - User-level thread: only need stack and program counter; no kernel involvement means fast switching. Kernel only sees the LWPs that support user-level threads.





Solaris 2 Threads

