Chapter 3: Processes

Chapter 3: Processes

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

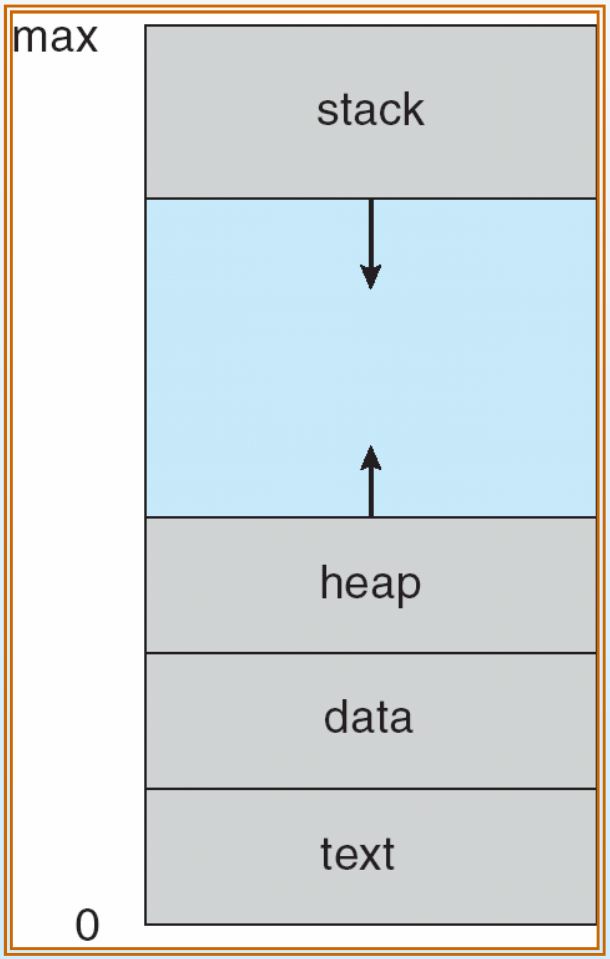
Process Concept

- 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:
  - text section: the program code
  - program counter
  - stack
  - data section

Process vs Programs

- The program itself is not a process
  - A program becomes a process only when the executable is loaded into main memory
  - A program is a passive entity whine the process in an active entity with a program counter assigned
- Two processes may be associated to the same program
  - But they are considered two separate execution sequences
  - If you invoke many copies of Firefox each of these is a separate process
    - The text sections are the same, but the stack, the data and the heap sections are different.

Process in Memory

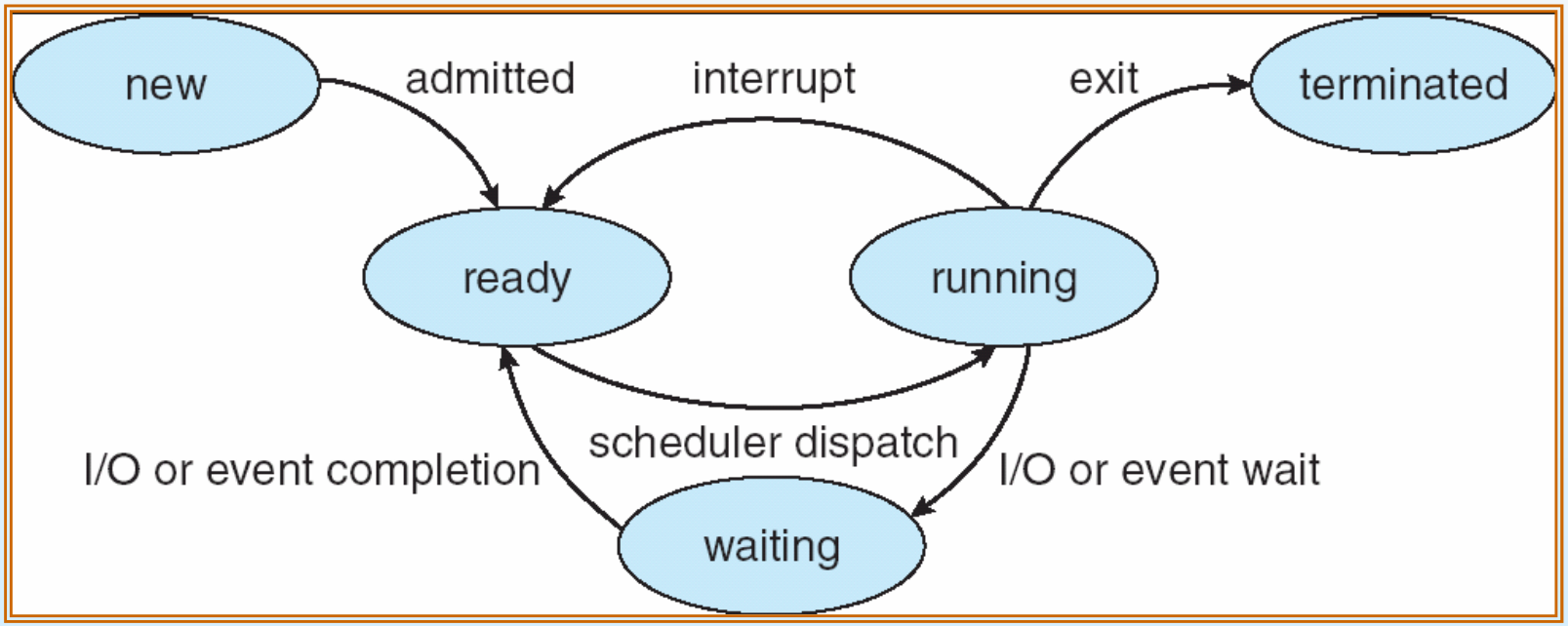


Process State

- 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 processor
  - **terminated**: The process has finished execution

- Different terminologies on these states (sleepy = waiting)
- **It is important to realize that only one process can be running on any processor at any instant.**

Diagram of Process State



Process Control Block (PCB)

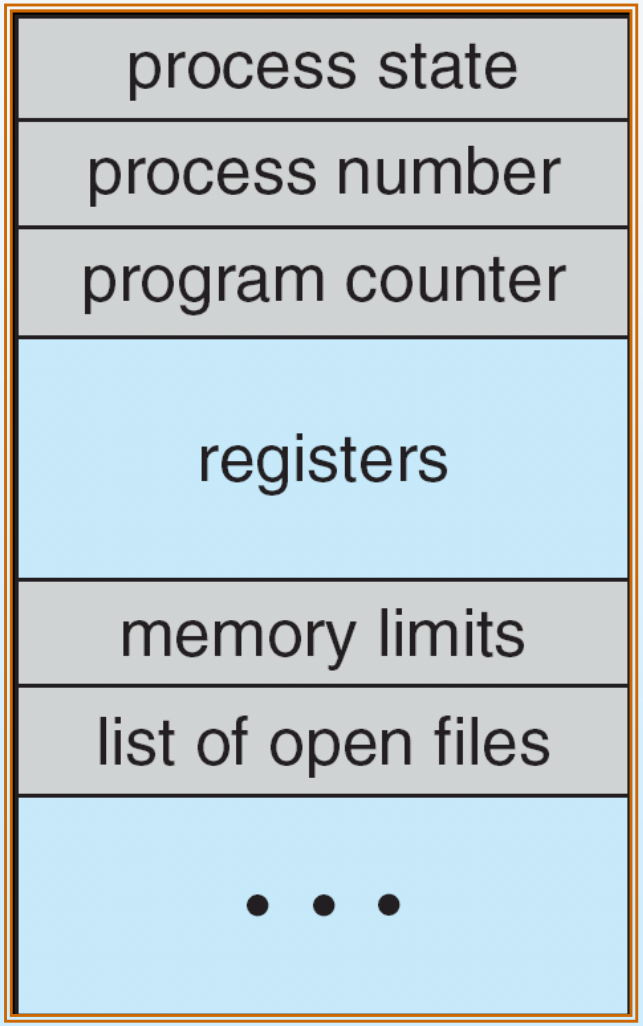
Information associated with each process

- Process state
  - The state may be new, ready, running, waiting, halted, and so on.
- Program counter
  - The counter indicates the address of the next instruction to be executed for this process.
- CPU registers
  - The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward.
- CPU scheduling information
  - This information includes a process priority,
  - Pointers to scheduling queues, and any other scheduling parameters.

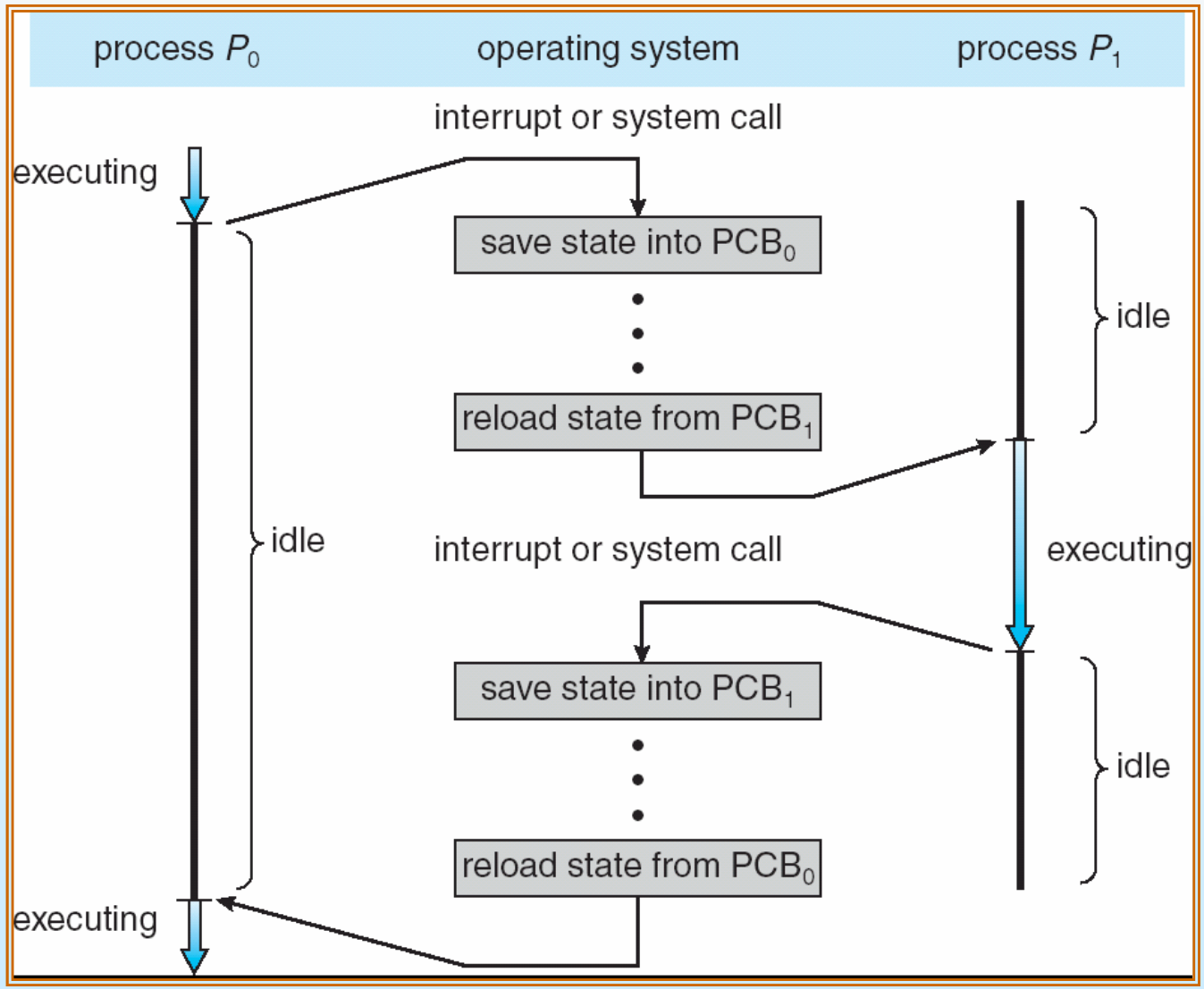
Process Control Block (PCB)

- Memory-management information
  - This information may include such information as the value of the base and limit registers, the page tables, or the segment tables depending on the memory system used by the operating system
- Accounting information
  - This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.
- I/O status information
  - The information includes the list of I/O devices (such as tape drives) allocated to this process, a list of open files, and so on.

Process Control Block (PCB)



CPU Switch From Process to Process



Single thread processes

- Single thread
  - Only one task at a time:
  - A single thread of instructions is executed.
  - For example if you are working in Word Processing, you can either type in characters of run the spell-checker. Not simultaneously!
- Multi-thread
  - Many tasks simultaneously: while you are typing another thread may perform spell checking!

Process Scheduling

- The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization.
- The objective of time sharing is to switch the CPU among processes so frequently that users can interact with each program while it is running.
- For a single-processor system, there will never be more than one running process.
- If there are more processes, the rest will have to wait until the CPU is free and can be rescheduled.

Process Scheduling Queues

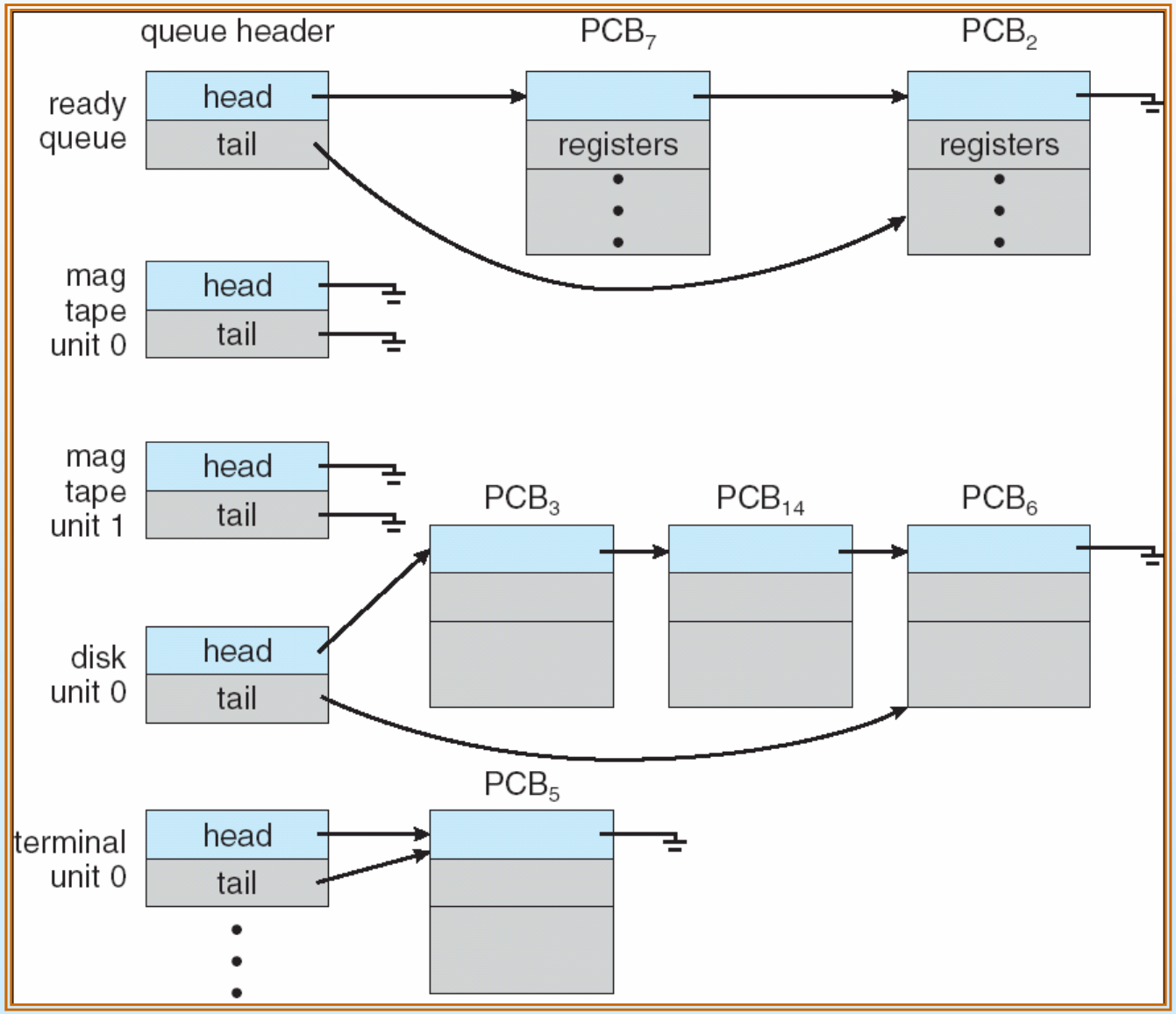
- **Job queue** – set of all processes in the system
  - As processes enter the system, they are put into a job *queue.*
- **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
  - The processes that are residing in main memory and are ready and waiting to execute are kept on this queue.
  - This queue is generally stored as a linked list. A ready-queue header will
  - Contains pointers to the first and last PCBs in the list. Each PCB has a pointer field that points to the next process in the ready queue.

Process Scheduling Queues

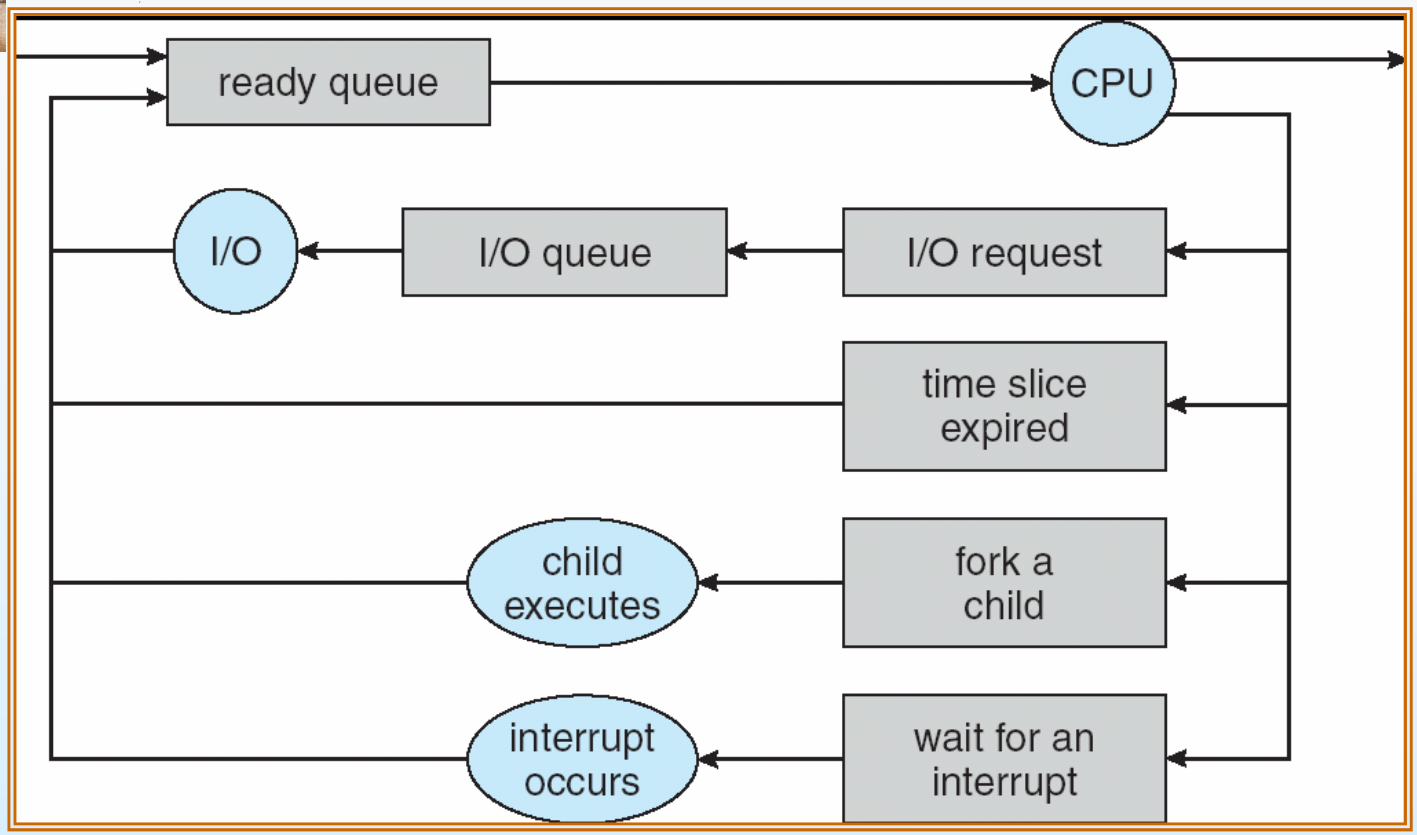
- **Device queues** – set of processes waiting for an I/O device
  - When a process is allocated the **CPU,** it executes for a while and eventually quits, is interrupted, or waits for the occurrence of a particular event, such as the completion of an I/O request.
  - In the case of an I/0 request, such a request may be to a dedicated tape drive, or to a shared device, such as a disk.
  - Since there are many processes in the system, the disk may be busy with the I/O request of some other process. The process therefore may have to wait for the disk.
  - The list of processes waiting for a particular I/0 device is called a ***device queue***. Each device has its own ***device queue***

- **Processes migrate among the various queues**

Ready Queue And Various I/O Device Queues



Representation of Process Scheduling



Queuing diagram of process scheduling

A new process is initially put in the ready queue. It waits in the ready queue until it is selected for execution (or *dispatched)* and is given the CPU.

Once the process is allocated the CPU and is executing, one of several events could occur:

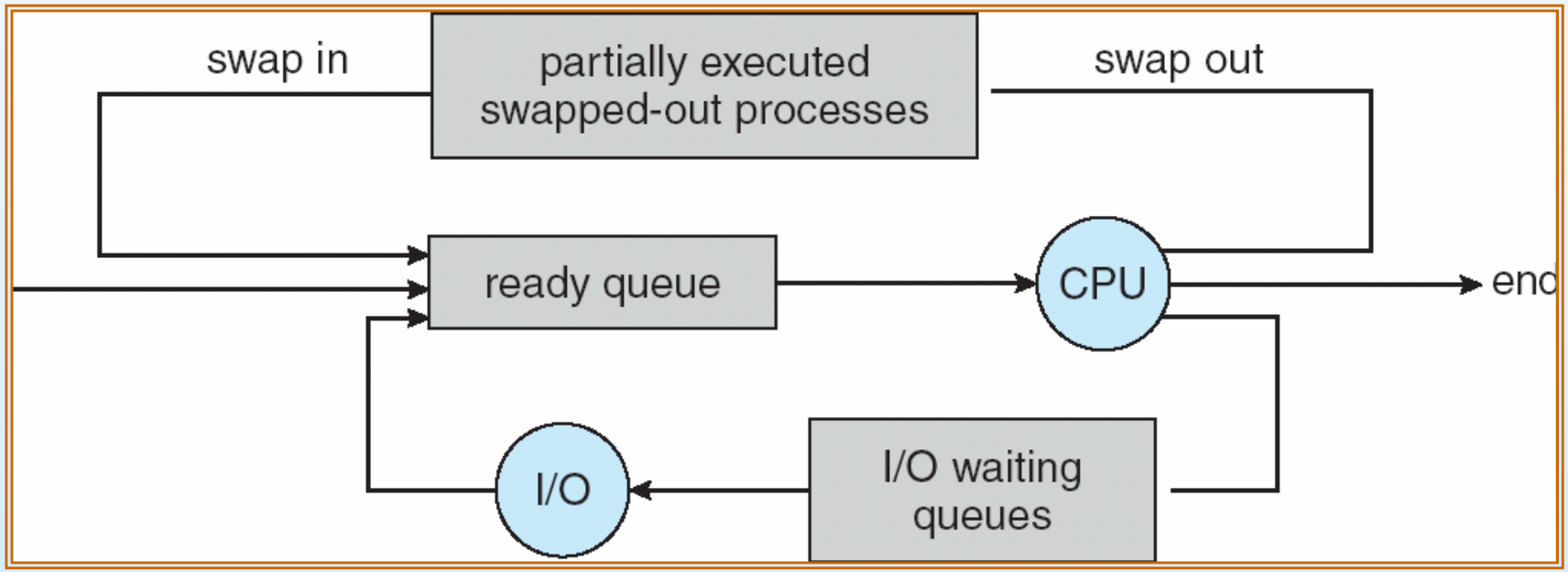
Process Scheduling

- The process could issue an I/O request, and then be placed in an I/O queue.
- The process could create a new subprocess and wait for its termination.
- The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.
- In the first two cases, the process eventually switches from the waiting state to the ready state, and is then put back in the ready queue.
- A process continues this cycle until it terminates, at which time it is removed from all queues and has its PCB and resources deallocated.

Schedulers

- A process migrates between the various scheduling queues throughout its lifetime.
- The operating system must select, for scheduling purposes, processes from these queues in some fashion.
- The selection process is carried out by the appropriate scheduler.
- **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



Batch system

- In a batch system, there are often more processes submitted than can be executed immediately.
- These processes are spooled to a mass-storage device (typically a disk), where they are kept for later execution.
- The long-term scheduler (or job scheduler) selects processes from this pool and loads them into memory for execution.
- The short-term scheduler (or *CPU* scheduler) selects from among the processes that are ready to execute, and allocates the CPU to one of them.

Long-term Vs. short-term

- The primary distinction between these two schedulers is the frequency of their execution.
- The short-term scheduler must select a new process for the CPU quite frequently. A process may execute for only a few milliseconds before waiting for an I/O request.
- Often, the short-term scheduler executes at least once every 100 milliseconds.
- Because of the short duration of time between executions, the short-term scheduler must be very fast. If it takes 10 milliseconds to decide to execute a process for 100 milliseconds, then 10/(100 + 10) = 9 percent of the CPU is being used (wasted) simply for scheduling the work.

Long-term Vs. short-term

- The long-term scheduler, on the other hand, executes much less frequently.
- There may be minutes between the creation of new processes in the system.
- The long-term scheduler controls the degree of multiprogramming (the number of processes in memory).
- If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system.
- Thus, the long-term scheduler may need to be invoked only when a process leaves the system.
- Because of the longer interval between executions, the long-term scheduler can afford to take more time to decide which process should be selected for execution.

Schedulers (Cont.)

- Short-term scheduler is invoked very frequently (milliseconds) ⇒ (must be fast)
- Long-term scheduler is invoked very infrequently (seconds, minutes) ⇒ (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
  - Its speed varies from machine to machine, depending on the memory speed, the number of registers which must be copied, and the existence of special instructions (such as a single instruction to load or store all registers).
  - Typically, the speed ranges from 1 to 1000 microseconds.
- Time dependent on hardware support
- Context switching may become a **performance bottleneck**
  - If there are more active processes than there are register sets, the system resorts to copying register data to and from memory, as before.

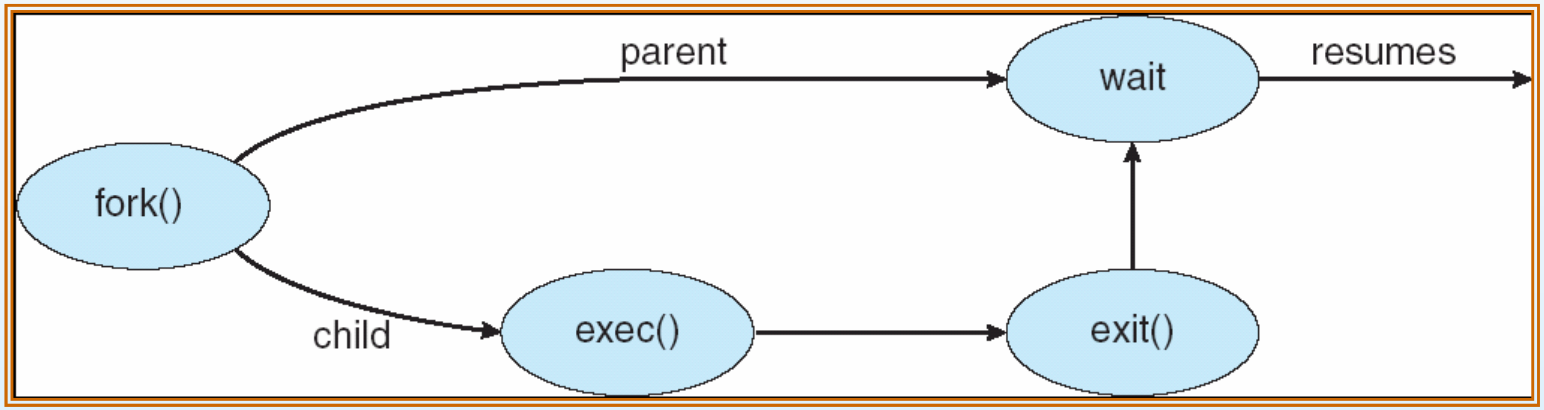
Process Creation

- 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

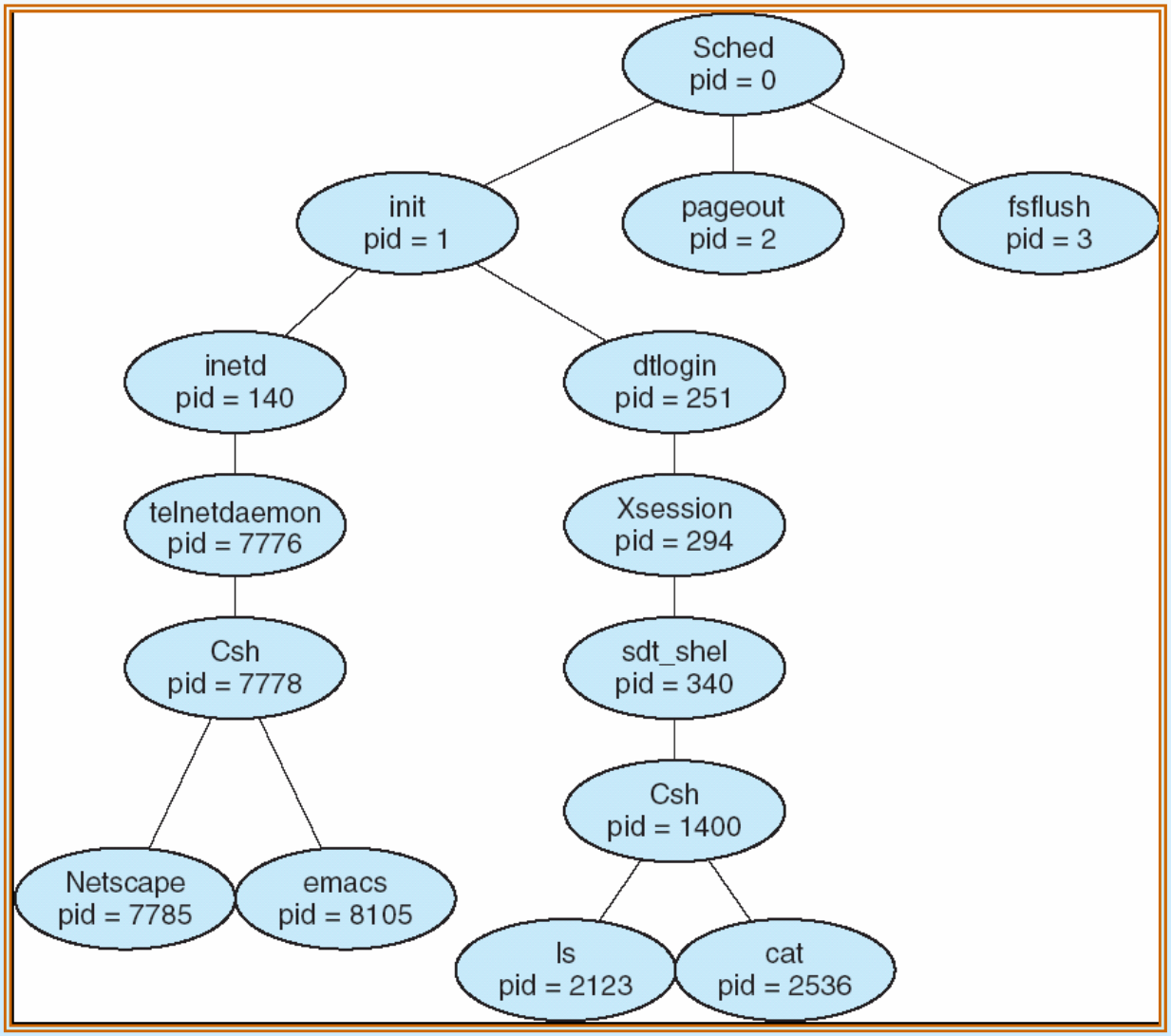
Process Creation



C Program Forking Separate Process

```
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);
        }
}
```

A tree of processes on a typical Solaris



Process Termination

- Process executes last statement and asks the operating system to delete 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
    - Information sharing: Since several users may be interested in the same piece of information (for instance, a shared file), we must provide an environment to allow concurrent access to these types of resources.
  - Computation speed-up
    - Computation speedup: If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Notice that such a speedup can be achieved only if the computer has multiple processing elements (such as CPUs or 1/0 channels).
  - Modularity
    - Modularity: We may want to construct the system in a modular fashion, dividing the system functions into separate processes
  - Convenience
    - Even **an** individual user may have many tasks to work on at one time. For instance, a user may be editing, printing, and compiling in parallel.

Producer-Consumer Problem

- **Paradigm** for cooperating processes, ***producer*** process produces information that is consumed by a ***consumer*** process
  - a print program produces characters that are consumed by the printer driver.
  - a compiler may produce assembly code, which is consumed by an assembler.
  - the assembler, in turn, may produce object modules, which are consumed by the loader.
  - CD and DVD burning programs use a buffer
    - Nero Burning
- The code must be written by the application programmer
- ***Buffer***
  - ***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

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
   . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- Solution is correct, but can only use BUFFER_SIZE-1 elements

Bounded-Buffer – Insert() Method

```
while (true) {
   /* Produce an item */
   while (((in = (in + 1) % BUFFER SIZE count)  == out)
     ;   /* do nothing -- no free buffers */
    buffer[in] = item;
    in = (in + 1) % BUFFER SIZE;
}
```

Bounded Buffer – Remove() Method

```
while (true) {
    while (in == out)
        ; // do nothing -- nothing to consume

    // remove an item from the buffer
    item = buffer[out];
    out = (out + 1) % BUFFER SIZE;
return item;
}
```

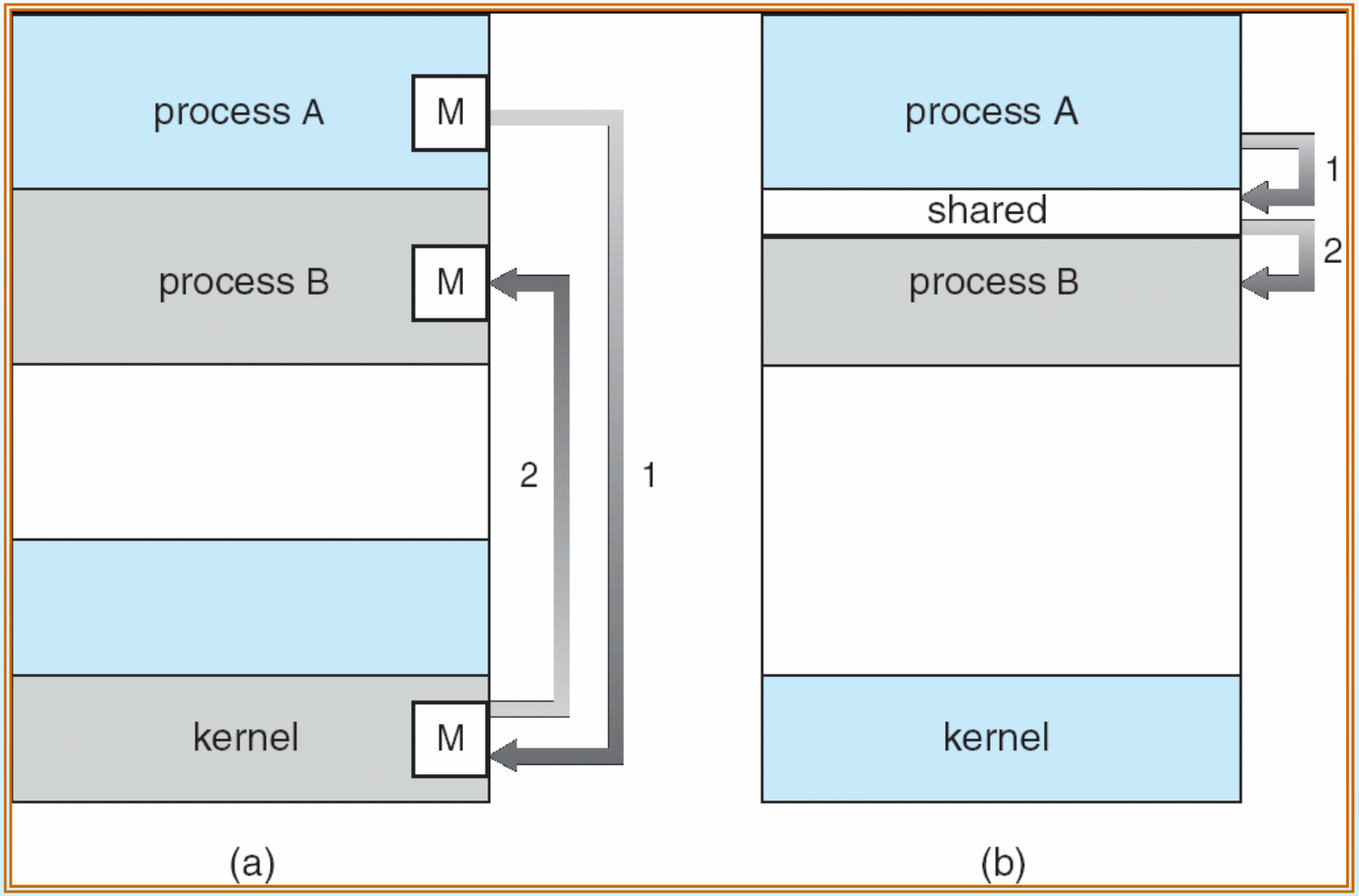
Interprocess Communication (IPC)

- Mechanism for processes to communicate and to synchronize their actions
  - Process do not share the same address space
- 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?

Communications Models



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
- This scheme exhibits a **symmetry** in addressing; that is, both the sender and the receiver processes have to name each other to communicate.
- A variant of this scheme may employ **asymmetry** in addressing. Only the sender names the recipient; the recipient is not required to name the sender.
- The disadvantage in both of these schemes (symmetric and asymmetric) is the limited modularity of the resulting process definitions.
  - Changing the name of a process may necessitate examining all other process definitions.
  - All references to the old name must be found, so that they can be modified to the new name. This situation is not desirable from the viewpoint of separate compilation.

Indirect Communication

- A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed.
- Each mailbox has a unique identification. In this scheme, a process can communicate with some other process via a number of different mailboxes.
- 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.

Mailboxes

- A mailbox may be owned either by a process or by the system. If the mailbox is owned by a process (that is, the mailbox is attached to or defined as part of the process),
  - We distinguish between the owner (who can only receive messages through this mailbox) and the user of the mailbox (who can only send messages to the mailbox).
  - Since each mailbox has a unique owner, there can be no confusion about who should receive a message sent to this mailbox.
  - When a process that owns a mailbox terminates, the mailbox disappears.
- On the other hand, a mailbox that is owned by the operating system has an existence of its own.
  - It is independent, and is not attached to any particular process.

OS mechanisms for mailboxes

- The operating system provides a mechanism that allows a process:
  - To create a new mailbox
  - To send and receive messages through the mailbox
  - To destroy a mailbox
- Since all processes with access rights to a mailbox may ultimately terminate, after some time a mailbox may no longer be accessible by any process.
  - In this case, the operating system should reclaim whatever space was used for the mailbox. This task may require some form of ***garbage collection.***

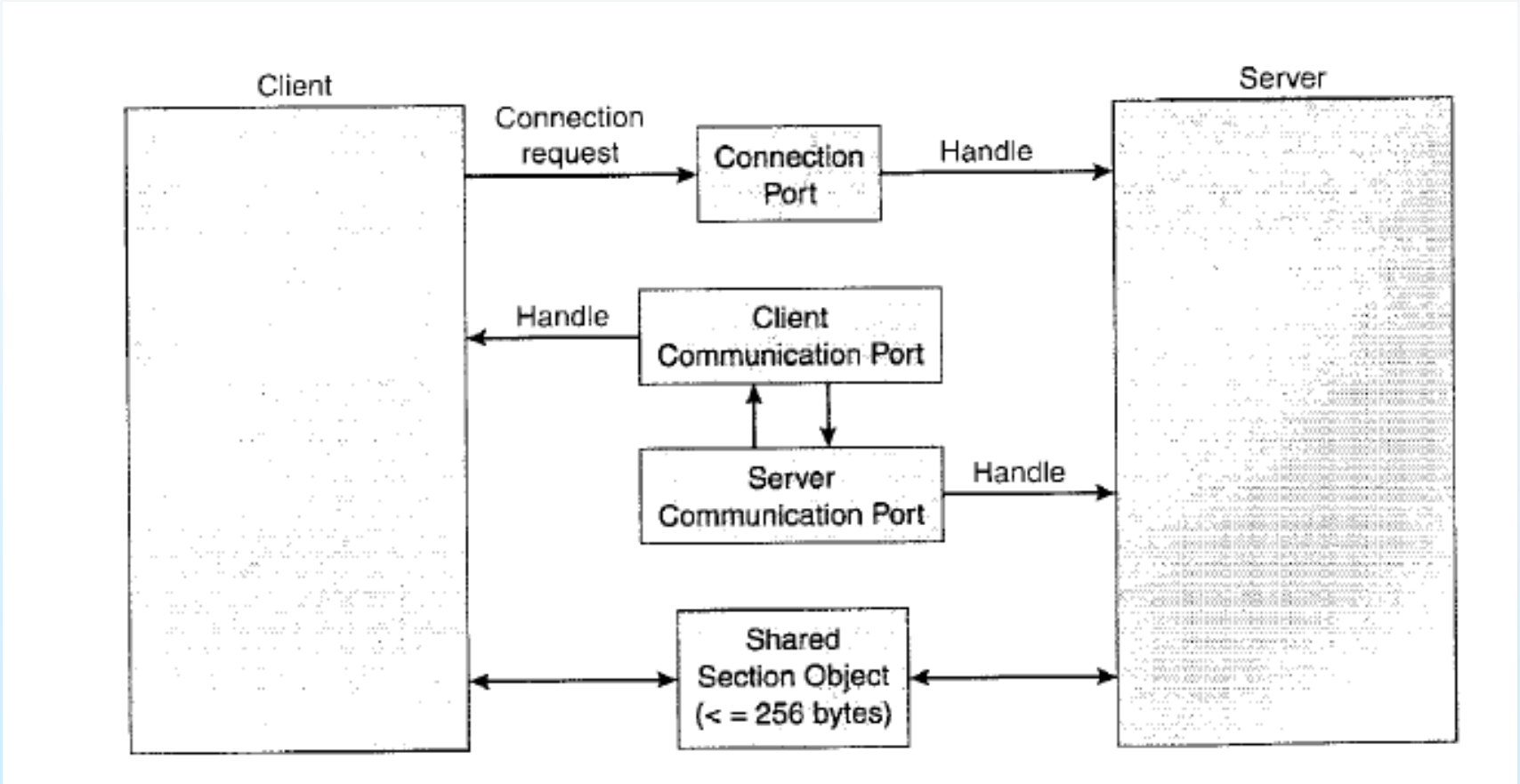
Synchronization

- 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:** the sender process sends the message and resumes operation
  - **Non-blocking:** the receiver receives 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

Windows XP


Client
Connection request
Connection Port
Handle
Server
Handle
Client Communication Port
Server Communication Port
Handle
Shared Section Object (< = 256 bytes)

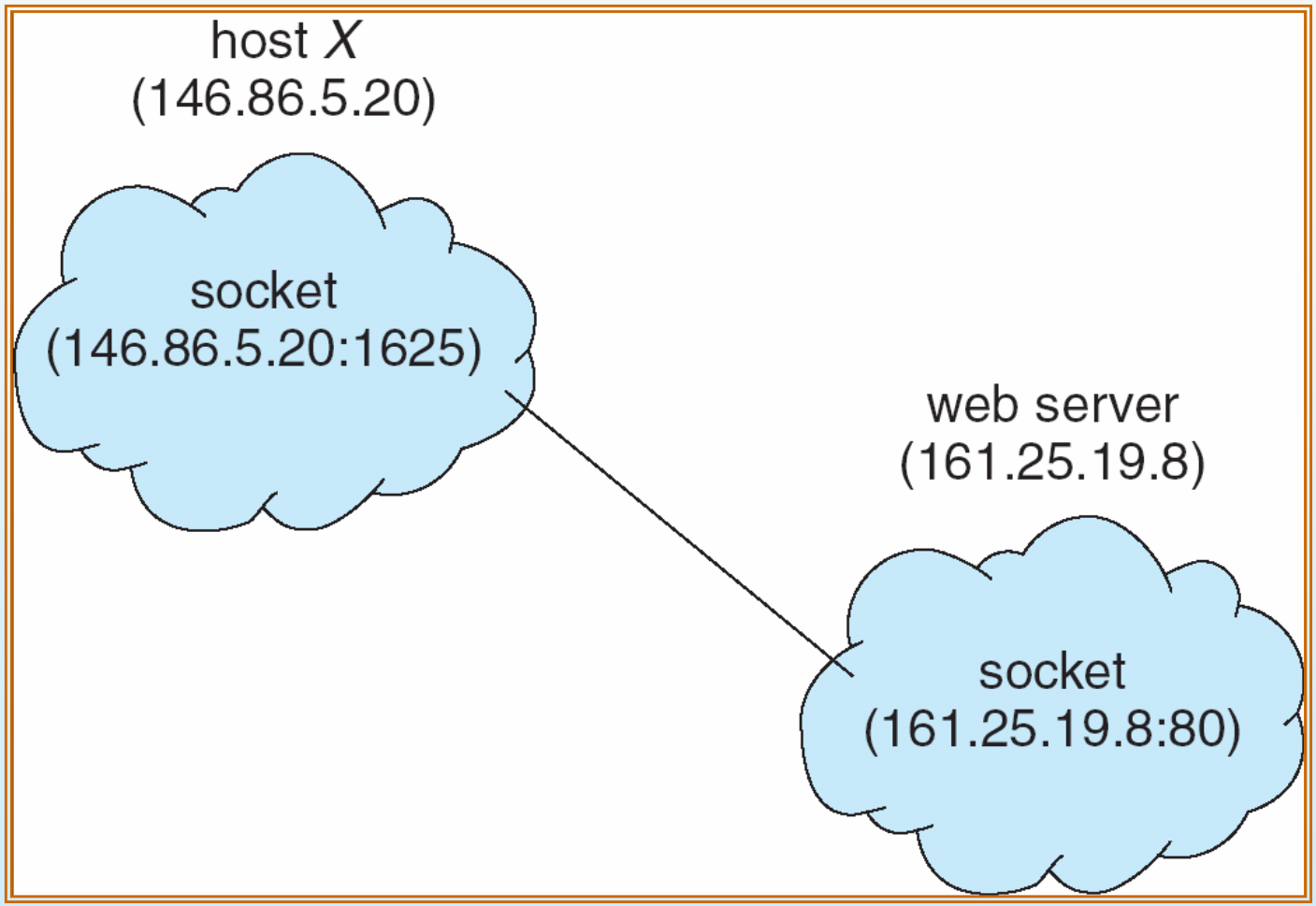
Client-Server Communication

- 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



Server

```
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);
      }
   }
}
```

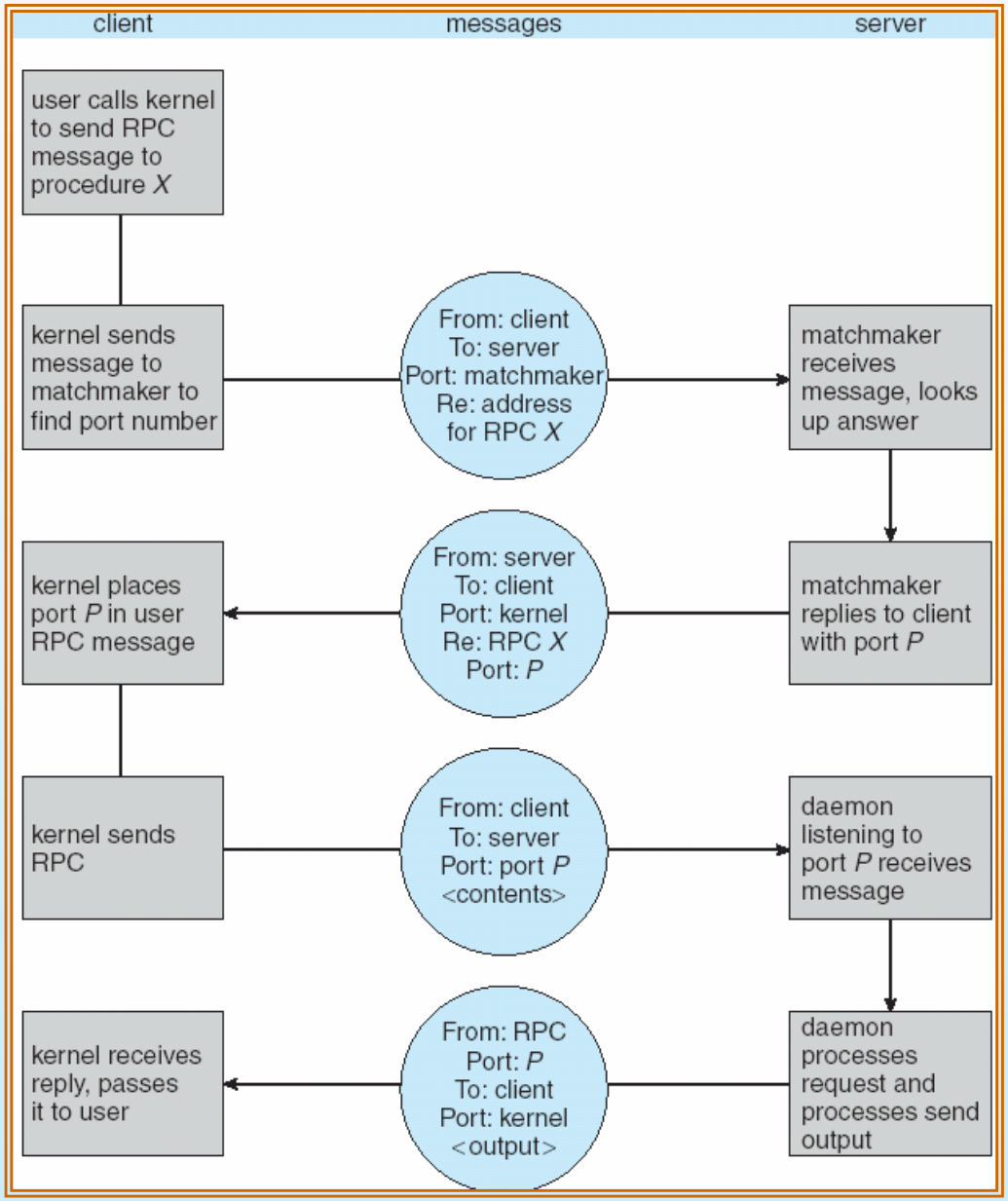
Client

```
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);
      }
   }
}
```

Remote Procedure Calls

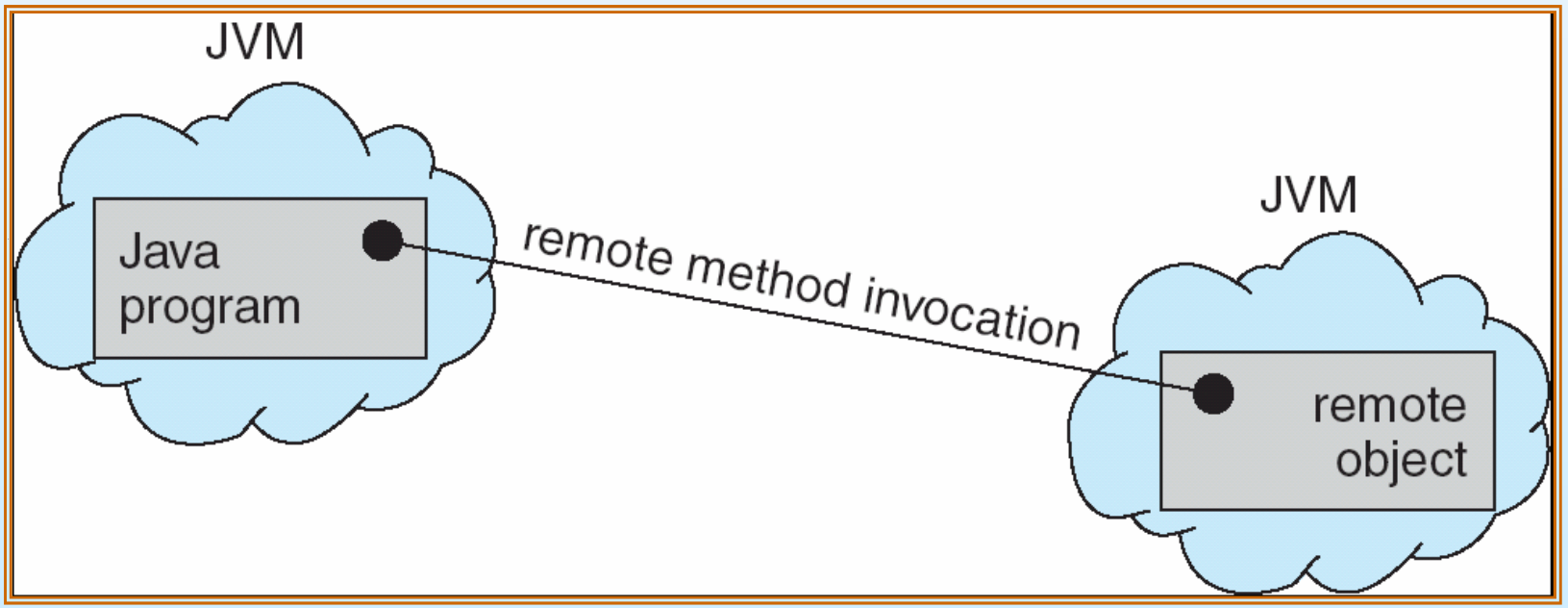
- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems.
- **Stubs** – client-side proxy for the actual procedure on the server.
- The client-side stub locates the server and *marshalls* the parameters.
- The server-side stub receives this message, unpacks the marshalled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

Execution of RPC

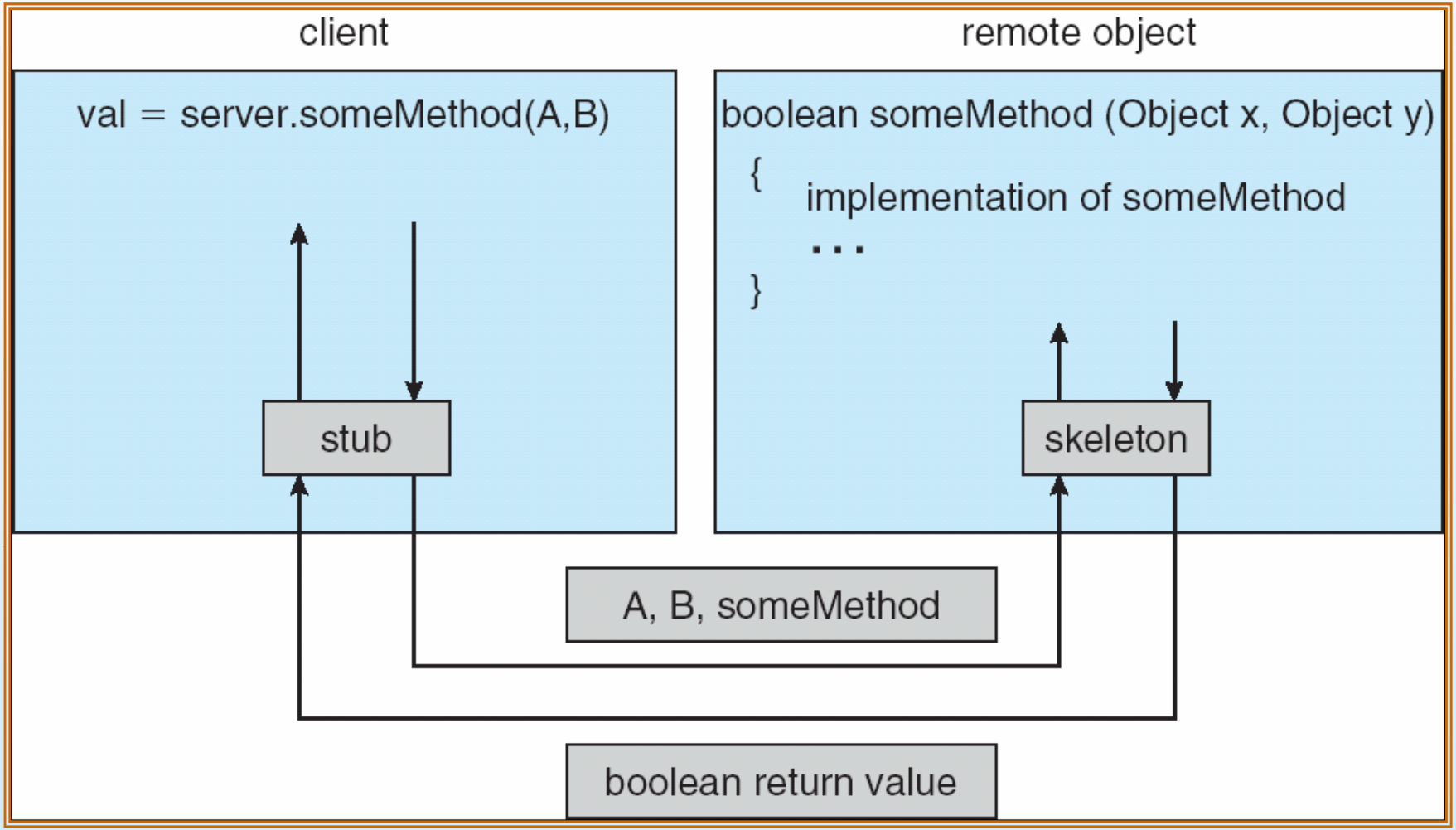


Remote Method Invocation

- 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



- Readings
  - Silberschatz, Chapter 3.

Relaxing quotation ☺

- There are known knowns; there are things we know we know. We also know there are known unknowns; that is to say we know there are some things we do not know. But there are also unknown unknowns - the ones we don't know we don't know.

- Donald Rumsfeld: Press Conference at NATO Headquarters, Brussels, Belgium, June 6, 2002
  - **Donald Rumsfeld** (born July 9, 1932) was the United States Secretary of Defense, succeeded by Robert Gates. He also served as Defense Secretary 1975–1977 under President Ford, and in other roles under various presidents.

End of Chapter 3