Chapter 3: Processes

Presented By: Dr. El-Sayed M. El-Alfy

Note: Most of the slides are compiled from the textbook and its complementary resources

Recap

- OS Services
- Major components of OS
  - User Interface, System Calls and System Programs
- Operating System Design and Implementation
- Operating System Structures
  - Simple, layered, microkernel, modules, VMs
- Operating System Generation
- System Boot

Objectives/Outline

Objectives

- Introduce the notation of a process
- Describe various features of processes including scheduling, creation and termination, and communication
- Describe communication in Client/Server systems

Outline

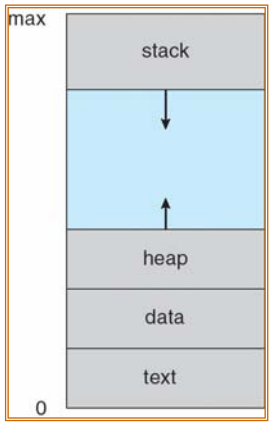
- Process Concept
- Process Scheduling
- Operations on Processes
- Cooperating Processes
- Interprocess Communication
- Communication in Client-Server Systems

Process Concept

- An operating system executes a variety of programs
- What to call all the CPU activities?
  - Batch system – jobs
  - Time-shared systems – user programs or tasks
- Textbook uses the terms *job* and *process* almost interchangeably
- Early computer systems (a single process) vs. modern time-sharing computer systems (multiple concurrent processes)
- Process – a program in execution
  - A program or executable file is not by itself a process
  - Different users may be running different copies of some program
  - The same user may be running many copies of some program
  - A process may spawn many processes as it runs
- Old OS: Process execution must progress in sequential fashion (single thread)
- Modern OS: allow multithreaded processes

Process in Memory

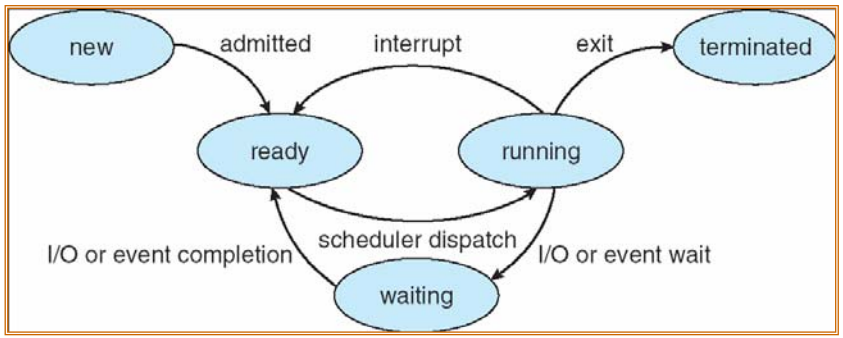
- A process includes:
  - process code (text section)
  - program counter (current activity)
  - content of the processor's registers
  - process stack – holds temporary data e.g. function parameters, return addresses, local variables
  - Data section – holds global variables
  - process heap – memory dynamically allocated during run time



Process State

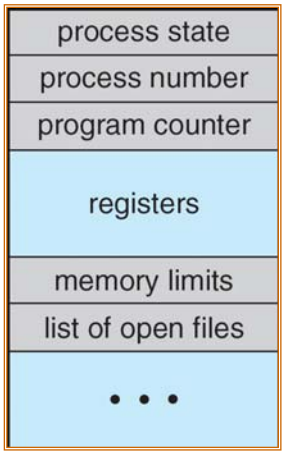
- As a process executes, it changes *state*
  - new: The process is being created
  - running: Instructions are being executed
  - waiting: The process is waiting for some event to occur (such as I/O completion)
  - ready: The process is waiting to be assigned to a processor
  - terminated: The process has finished execution (halted)
- These names are arbitrary and may vary across operating systems

Process State Diagram

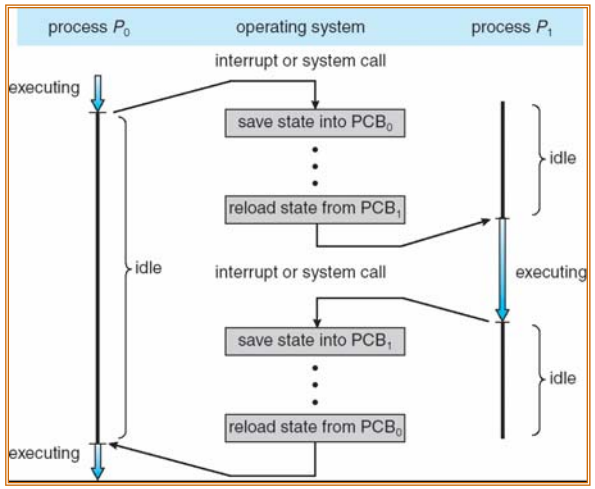


Process Control Block (PCB)

- A repository for any information associated with each process including:
  - Process state
  - Program counter (PC)
  - CPU registers
    - E.g. accumulators, index registers, general-purpose registers
  - CPU scheduling information
    - E.g. process priority, pointers to scheduling queues
  - Memory-management information
    - E.g. base and limit registers, page tables
  - Accounting information
    - E.g. account numbers, amount of CPU, time limits
  - I/O status information
    - E.g. a list of I/O devices allocated to the process, list of open files



CPU Switch from Process to Process (Sequence Diagram)



Process Scheduling

- The objective of multiprogramming is to have some process running at all the times; thus maximizing the CPU utilization
- The objective of time sharing is to switch between processes so frequently that users can interact with each program while it is running
- Process scheduler: selects a process from the available set of processes for program execution on the CPU
- A single-processor CPU will never have more than one process running at any given time; the rest of processes will be waiting until the CPU is free

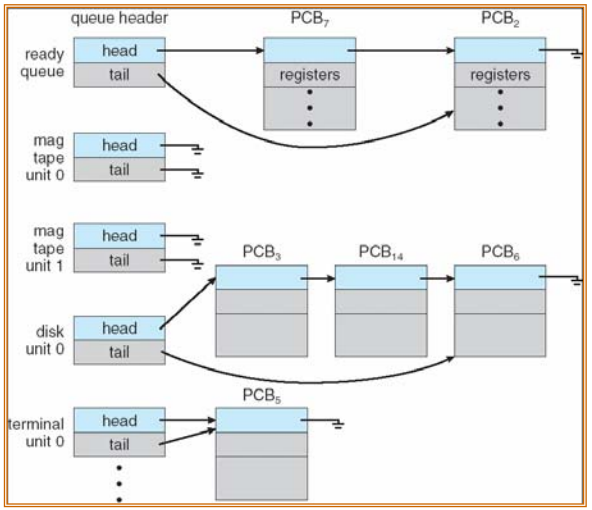
Process Scheduling Queues

- Job queue – set of all processes in the system
- Ready queue – set of all processes residing in main memory and are ready and waiting to execute
- Device queue – set of processes waiting for a particular I/O device; each I/O has its own device queue

- Processes migrate among the various queues throughout its lifetime

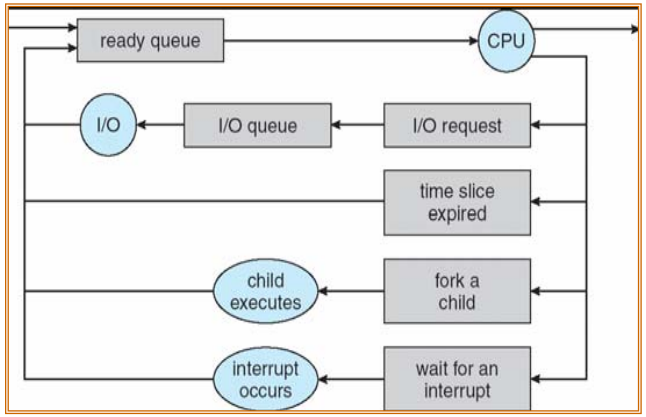
Ready Queue and Various I/O Device Queues

- A queue is generally stored as a linked list
- The queue header has a pointer to the first and final PCBs



Queueing Diagram

- A common representation of process scheduling

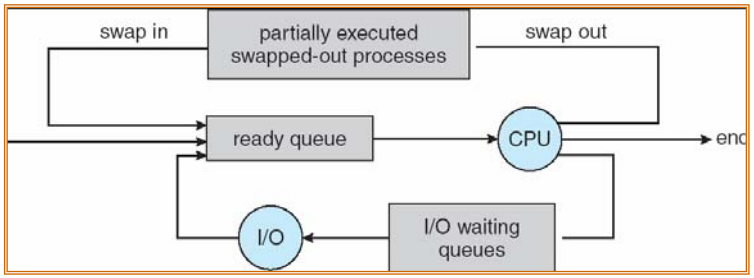


Schedulers

A scheduler can be:

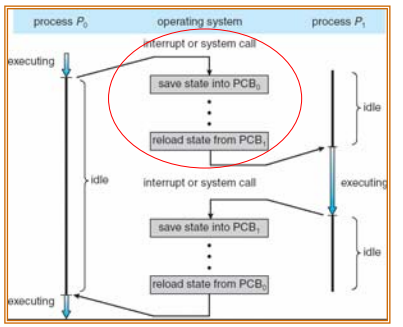
- Long-term scheduler (or job scheduler) –
  - selects which processes should be brought into the ready queue
  - executed much less frequently (seconds, minutes) ⇒ (may be slow)
  - controls the *degree of multiprogramming*
  - Select a good mix of processes: **I/O-bound process** (spends more time doing I/O than computations, many short CPU bursts) and **CPU-bound process** (spends more time doing computations; few very long CPU bursts)
- Short-term scheduler (or CPU scheduler)
  - selects which process should be executed next and allocates CPU
  - executed more frequently (milliseconds) ⇒ (must be fast)
- Medium-term scheduler
  - may exist on some time sharing systems
  - removes a process temporarily from memory or CPU contention to reduce multiprogramming and later resume execution where it is left off (swapping)

Addition of Medium-Term Scheduling



Context Switch

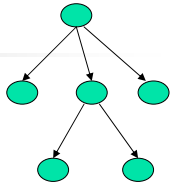
- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process
- Context-switch time is overhead; the system does no useful work while switching
- Time dependent on hardware support



W3.L2

- Process creation
- Process termination
- Interprocess communication

Process Creation

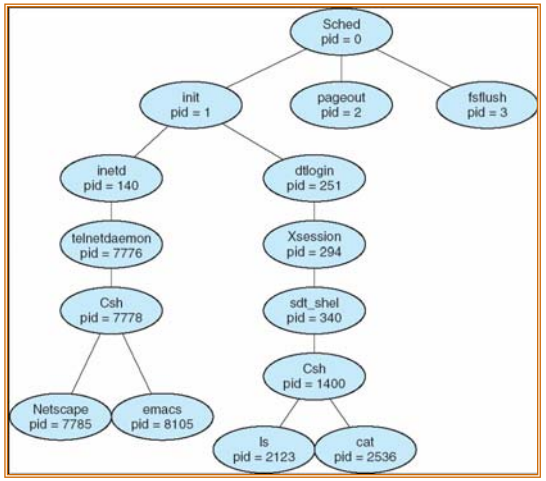


- In most OS, processes
  - can execute concurrently, and
  - may be created and deleted dynamically
- A process may create several new processes (via create-process system call)
- Parent process create children processes, which, in turn create other processes, forming a tree of processes
- Resource sharing (CPU, memory, files, I/O devices)
  - Parent and children share all resources,
  - Children share subset of parent's resources, or
  - Parent and children share no resources

Example: A tree of processes on a typical Solaris system

- Each process has a process name and a process identifier (PID)
- ps -el

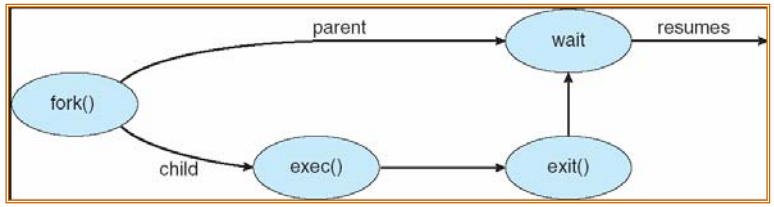
  list complete information for all active processes



Process Creation (Cont.)

- Execution
  - Parent and children execute concurrently, or
  - Parent waits until some or all of its children terminate
- 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 (overlay)

Example: Process Creation on Unix



Example: Process Creation on Unix (Cont.): C Program Forking Separate Process

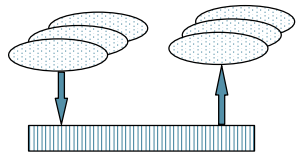
```
int main() {
Pid_t  pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
            fprintf(stderr, "Fork Failed");
            exit(-1);
    }
    else if (pid == 0) { /* child process */
            execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent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


Process Termination

- Process executes last statement and asks the operating system to delete it (via exit() system call)
  - Output data from child to parent (status code)
  - 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*

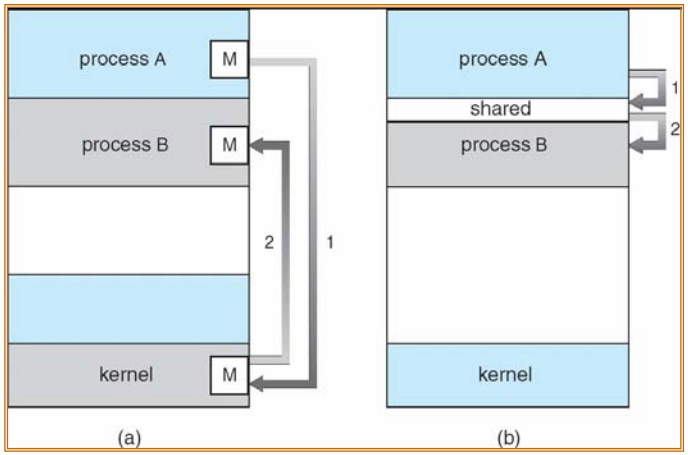
Cooperating Processes

- 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
- Producer-Consumer Problem
  - Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
  - E.g. an assembler produces object modules that are consumed by a loader.
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience



- Cooperating processes require inter-process communication (IPC) mechanism

IPC Models



(a) message passing, (b) shared memory
Many systems allow both

IPC via Shared Memory

W3.L3

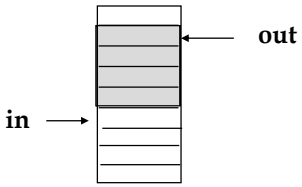
- One process creates a shared memory in its address space and allows others to use it
- Advantages
  - Allows max. speed and convenience (at memory speed on the same computer)
  - System calls are used only to establish shared memory
  - No assistance from the kernel is required after that
  - Processes have complete control and agreement on the form and location of the data
- Detriments
  - Writing must be mutually exclusive to prevent a race condition leading to inconsistent data views.
- Implementation
  - Unbounded-buffer places no practical limit on the size of the buffer
    - producer: no wait
    - consumer: wait when buffer is empty
  - Bounded-buffer assumes that there is a fixed buffer size
    - producer: wait when buffer is full
    - consumer: wait when buffer is empty

Bounded-Buffer – Shared-Memory Solution

- Shared data

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

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



- Implemented as a circular array
  - can only use BUFFER_SIZE-1 elements
  - Empty: in == out
  - Full: (in+1) % BUFFER_SIZE == out

Bounded-Buffer: Producer process

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

Bounded Buffer: Consumer Process

```
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;
 }
```


IPC via Message Passing

- Processes communicate with each other without resorting to shared variables
- If *P* and *Q* wish to communicate, they need to:
  - establish a *communication link* between them
  - exchange messages via send/receive
- Advantages
  - Communication provides a way to synchronize, or coordinate, various activities.
  - Good for exchanging smaller amount of data
  - No conflict need to be avoided
  - Easier to implement for inter-computer communication
  - Slower (via system calls)
- Detriments
  - May have deadlock - each process waiting for a message from the other process.
  - May have starvation - two processes sending a message to each other while another process waits for a message.
- Implementation of communication link
  - physical (e.g., shared memory, hardware bus, or network)
  - 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

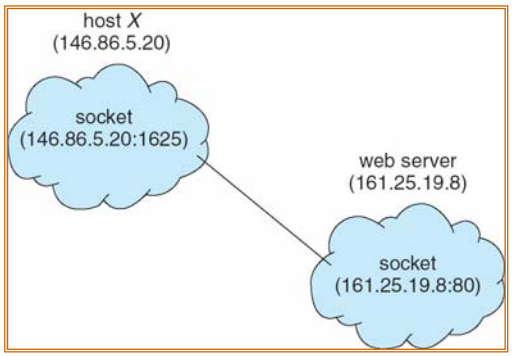
- Queue of messages attached to the link; implemented in one of three ways
  1. Zero capacity – 0 messages [no buffering]
     Sender must wait for receiver (rendezvous)
  2. Bounded capacity – finite length of *n* messages
     Sender must wait if link full
  3. Unbounded capacity – infinite length
     Sender never waits

Client-Server Communication

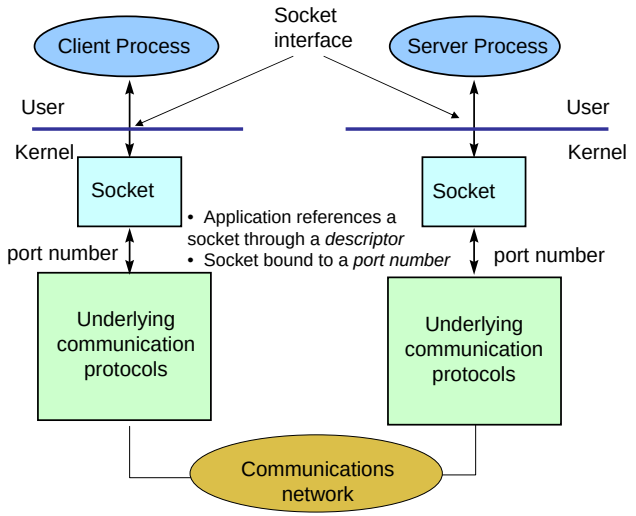
- In addition to shared memory and message passing, processes in C/S systems can communicate using:
  - Sockets
  - Remote Procedure Calls (RPC)
  - Remote Method Invocation (RMI) in Java

Socket Communication

- 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



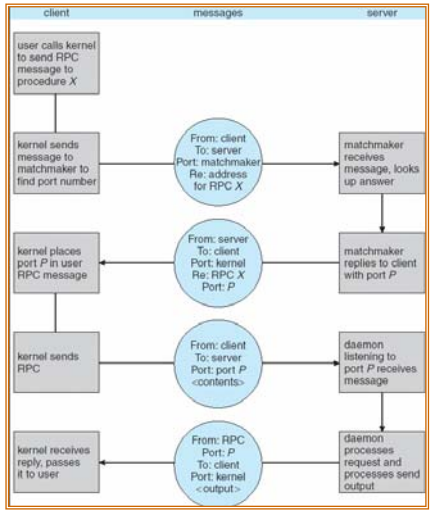
Socket Communication (Cont.)



Remote Procedure Calls

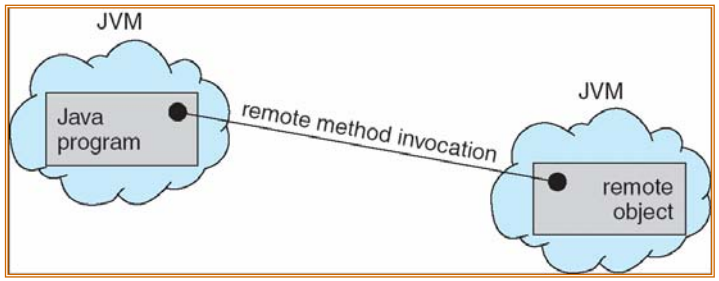
- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems.
  - A client can invoke a procedure on a remote host as it does locally
- Similar in many ways to IPC by passing messages and usually built on top of such systems
  - Messages have no longer been just packets of data
  - Messages are addressed to an RPC daemon listening to a port on the remote system
  - Each message is well structured and contains: identifier of the function to be executed, parameters passed to that function, the output is sent back on a separate message
- Stubs – client-side proxy for the actual procedure on the server.
- The client-side stub locates the server and marshals the parameters.
- The server-side stub receives this message, unpacks the marshaled 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 (in a different JVM).

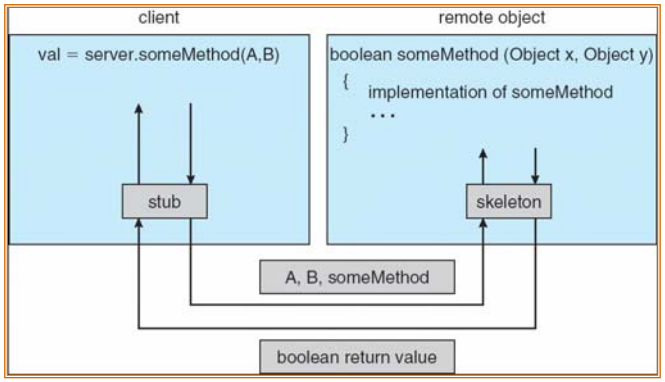


RMI vs. RPC

- RPC supports procedural programming (only remote procedures or functions can be called)
- RMI is object based (invoke methods on remote objects)
- Parameters to RPC are primitive or ordinary data structures
- Parameters to RMI can be objects as well
- RMI allow developing distributed applications across a network

Marshalling Parameters

- RMI is implemented using stubs and skeletons to be transparent for the client and the server
  - A stub is a proxy for the remote object; resides on the client side; creates a parcel and sends it to the skeleton
  - A skeleton resides on the server; receives and unpacks the parcel; invokes the method; marchals the return value



End of Chapter 3

Operating System Concepts, 7th Ed. A. Siblerschatz, P. Galvin, and G. Gagne. Addison Wesley, 2005