SNS COLLEGE OF TECHNOLOGY

Coimbatore-35
An Autonomous Institution

Accredited by NBA – AICTE and Accredited by NAAC – UGC with 'A+' Grade
Approved by AICTE, New Delhi & Affiliated to Anna University, Chennai

DEPARTMENT OF INFORMATION TECHNOLOGY

23CST202 – Operating Systems

II YEAR - IV SEM

UNIT 1 – Overview and Process Management

TOPIC 2 – Process

Syllabus

UNIT I **OVERVIEW AND PROCESS MANAGEMENT** **9**

Introduction - Computer System Organization, Architecture, Operation, Process Management – Memory Management – Storage Management – Operating System – Process concept – Process scheduling – Operations on processes – Cooperating processes – Inter process communication. Threads - Multi-threading Models – Threading issues.

UNIT II **PROCESS SCHEDULING AND SYNCHRONIZATION** **10**

CPU Scheduling - Scheduling criteria – Scheduling algorithms – Multiple-processor scheduling – Real time scheduling – Algorithm Evaluation. Process Synchronization - The critical-section problem – Synchronization hardware – Semaphores – Classical problems of synchronization. Deadlock - System model – Deadlock characterization – Methods for handling deadlocks – Deadlock prevention – Deadlock avoidance – Deadlock detection – Recovery from deadlock.

UNIT III **MEMORY MANAGEMENT** **9**

Memory Management - Background – Swapping – Contiguous memory allocation – Paging – Segmentation – Segmentation with paging. Virtual Memory - Background – Demand paging – Process creation – Page replacement – Allocation of frames – Thrashing.

UNIT IV **FILE SYSTEMS** **8**

File concept - Access methods – Directory structure – Files System Mounting – File Sharing – Protection. File System Implementation - Directory implementation – Allocation methods – Free-space management.

UNIT V **I/O SYSTEMS** **9**

I/O Systems - I/O Hardware – Application I/O interface – Kernel I/O subsystem – Streams – Performance. Mass-Storage Structure: Disk scheduling – Disk management – Swap-space management – RAID – Disk attachment – Stable storage – Tertiary storage. Case study: Implementation of Distributed File system in Cloud OS / Mobile OS.

L :45 P:0 T: 45 PERIODS

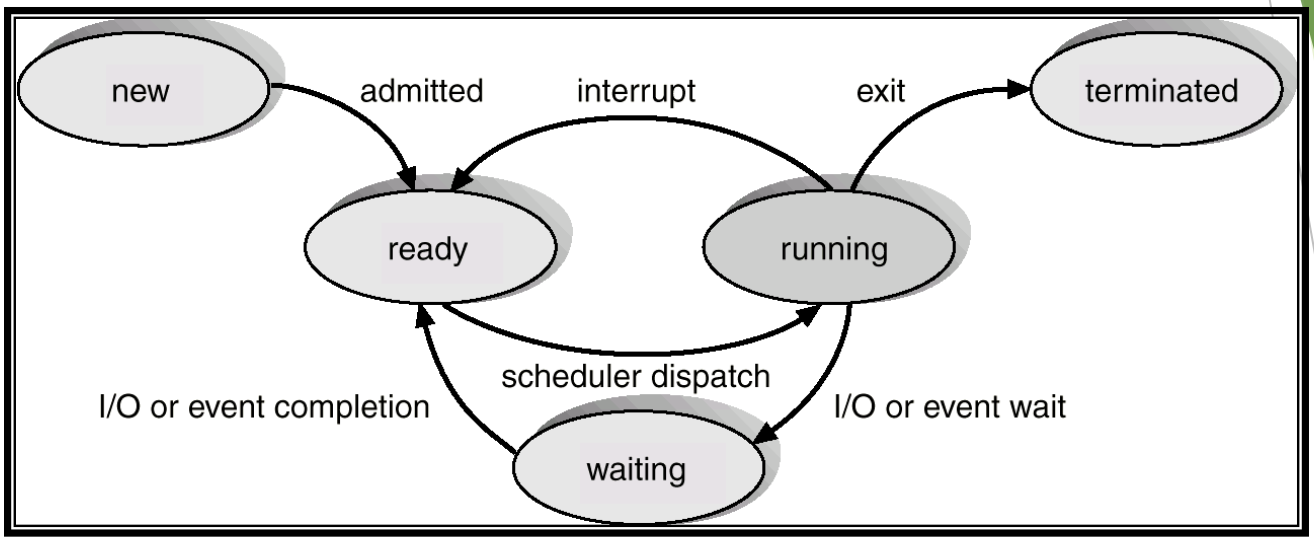
Process Concept

- 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

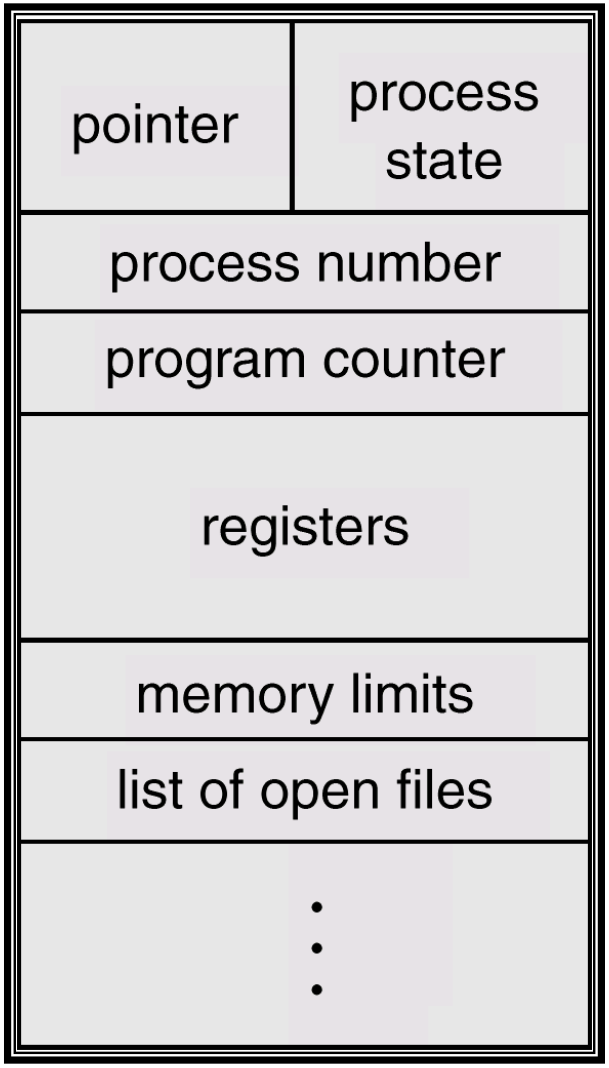


Process Control Block (PCB)

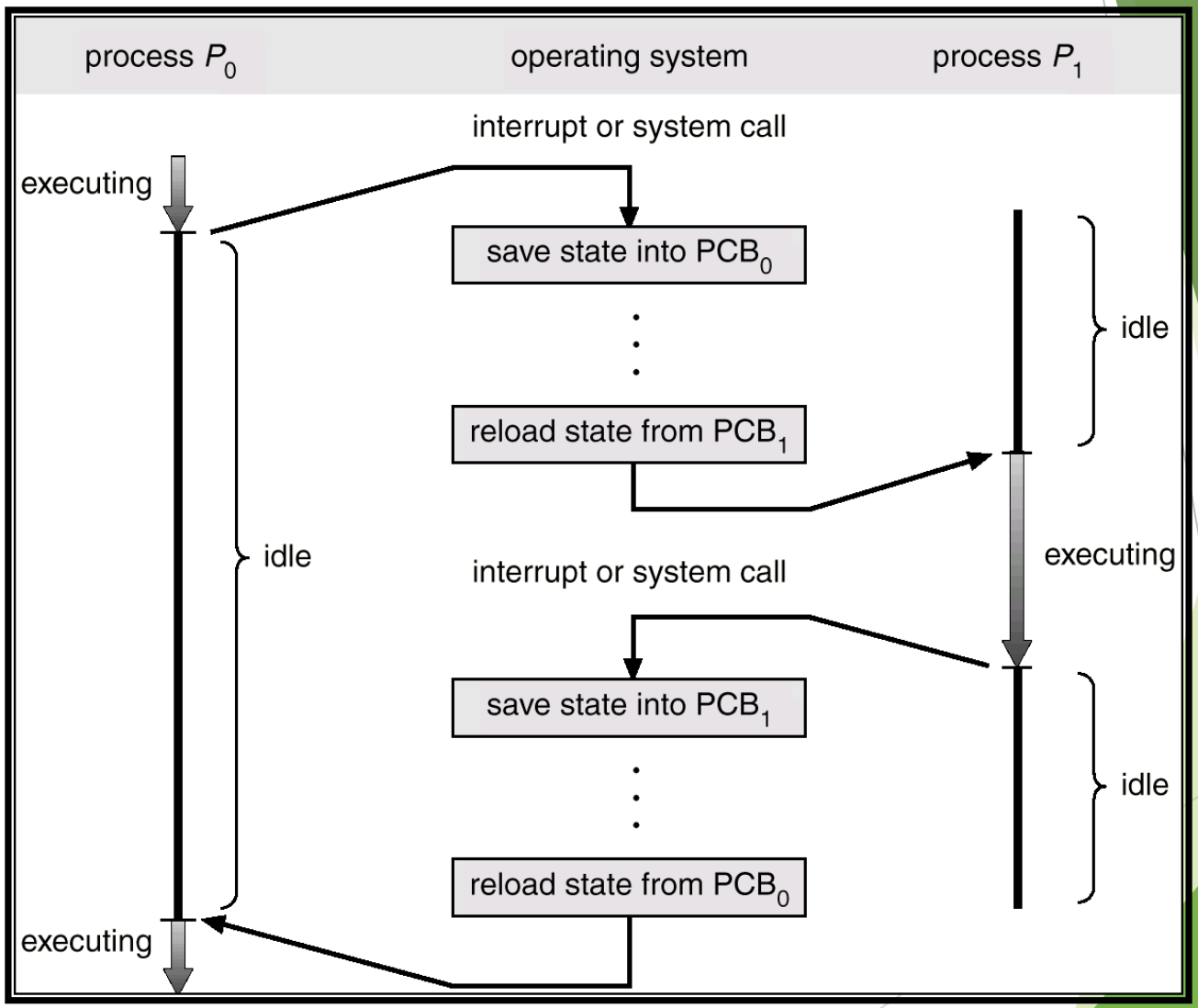
Information associated with each process.

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

Process Control Block (PCB)



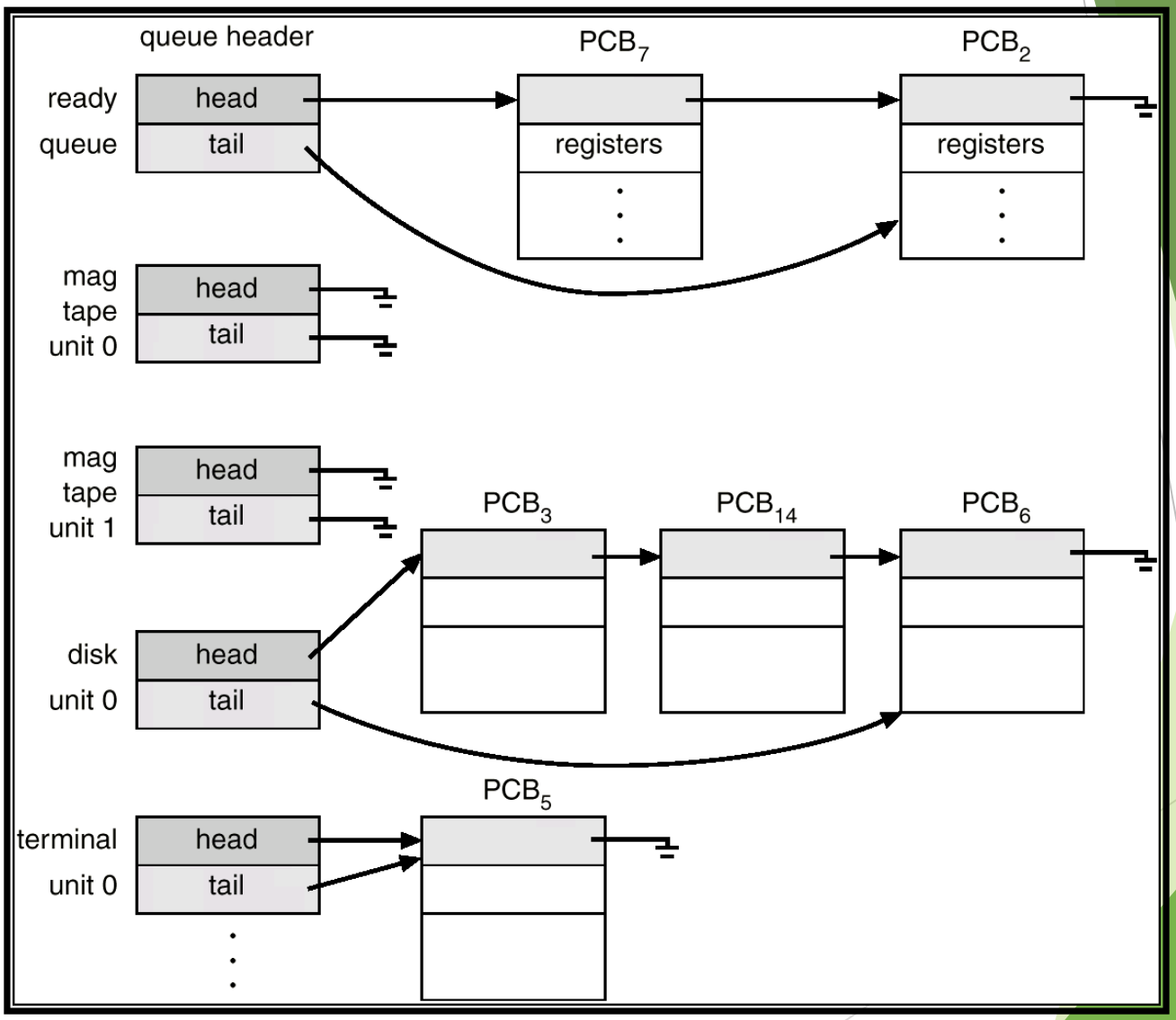
CPU Switch From Process to Process



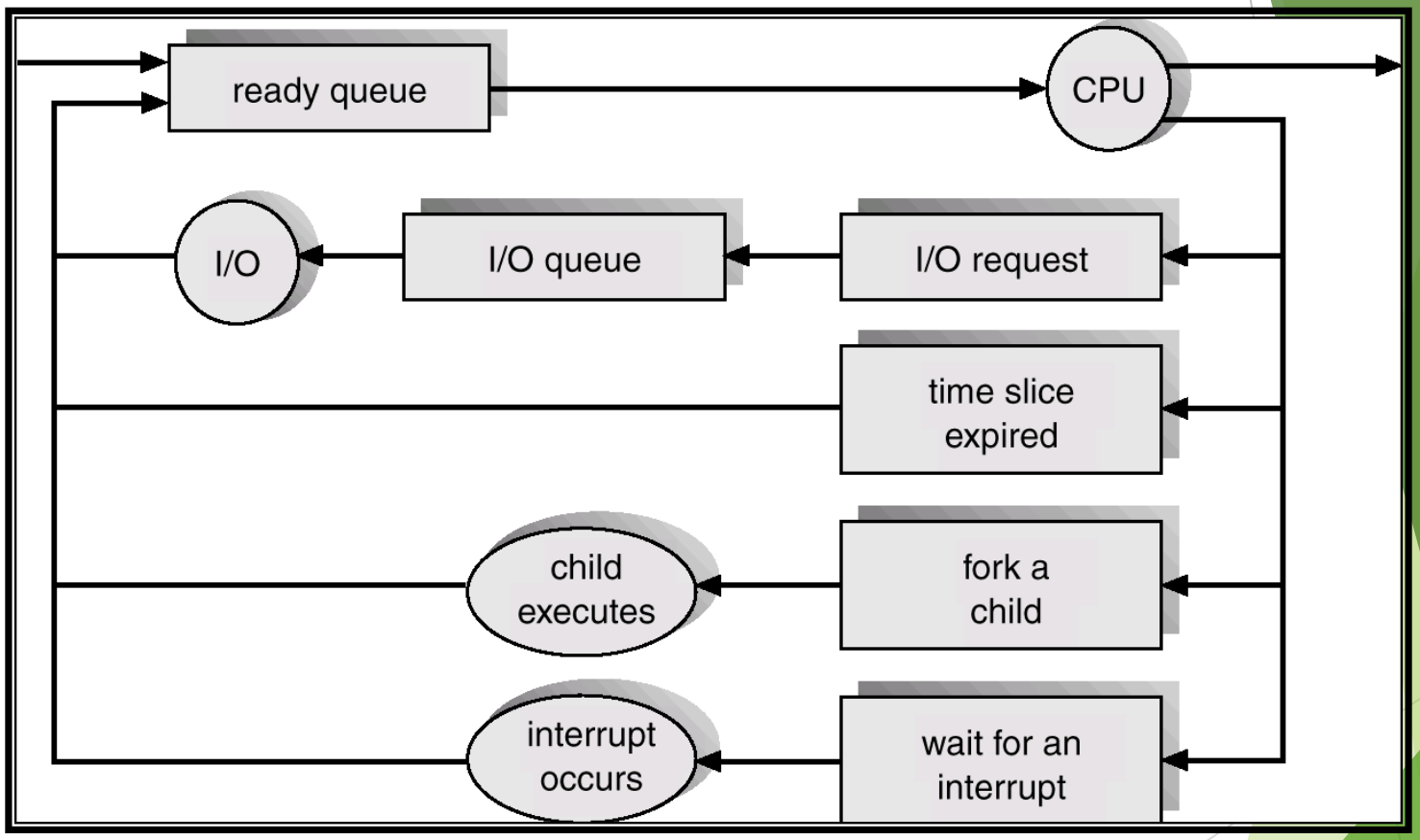
Process Scheduling Queues

- 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



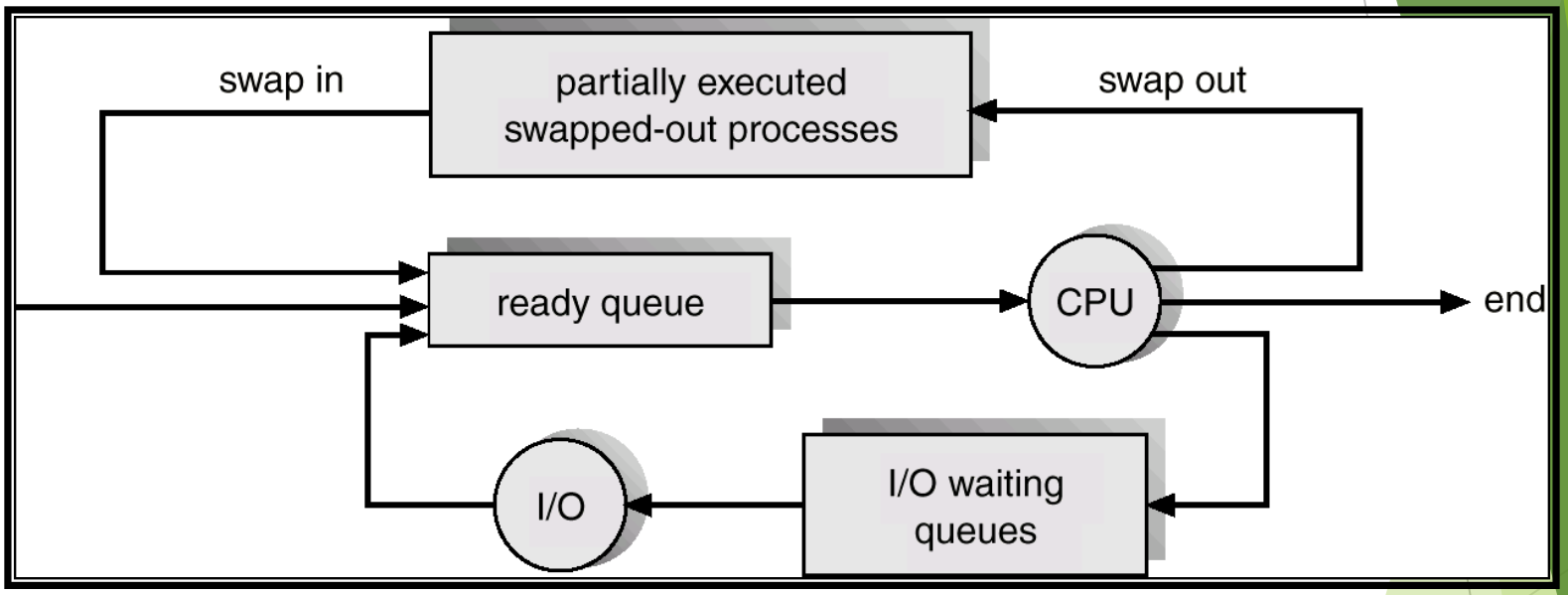
Representation of Process Scheduling



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) ⇒ (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.
- Time dependent on hardware support.

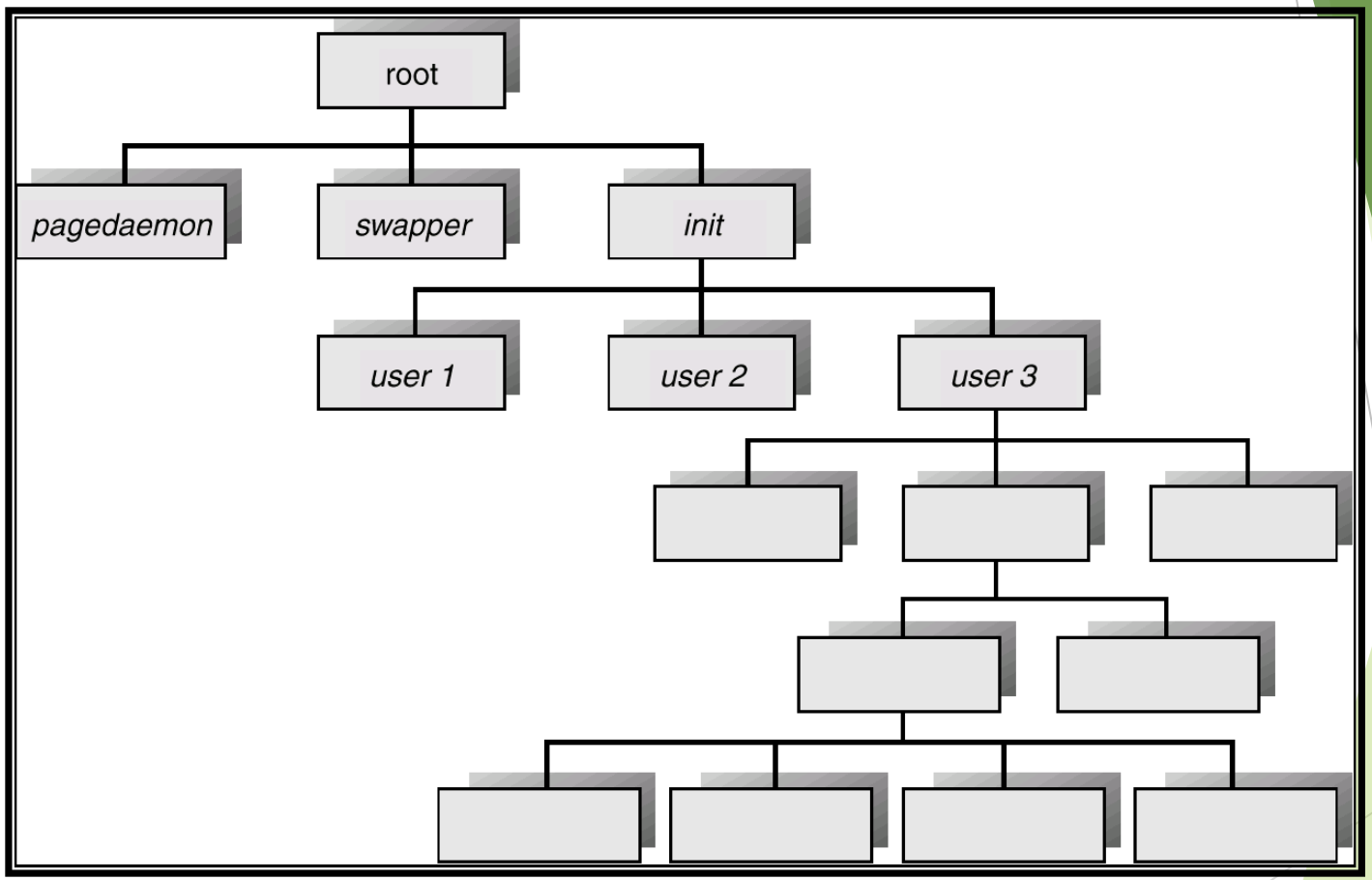
Process Creation

- 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.

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.
  - Parent is exiting.
    - Operating system does not allow child to continue if its parent terminates.
    - 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

- 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 – Producer Process

```
item nextProduced;

while (1) {
  while (((in + 1) % BUFFER_SIZE) == out)
        ; /* do nothing */
  buffer[in] = nextProduced;
  in = (in + 1) % BUFFER_SIZE;
}
```

Bounded-Buffer – Consumer Process

```
item nextConsumed;

while (1) {
  while (in == out)
        ; /* do nothing */
  nextConsumed = buffer[out];
  out = (out + 1) % BUFFER_SIZE;
}
```

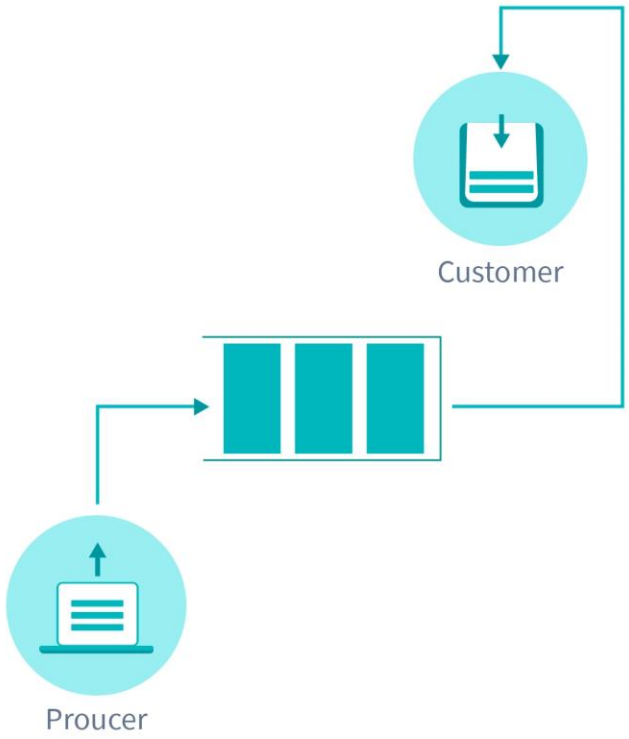
Producer-Consumer Problem

- Producer-Consumer problem is a classical synchronization problem in the operating system.
- With the presence of more than one process and limited resources in the system the synchronization problem arises.
- If one resource is shared between more than one process at the same time then it can lead to data inconsistency.
- In the producer-consumer problem, the producer produces an item and the consumer consumes the item produced by the producer.

Producer-Consumer Problem

- **What is Producer Consumer Problem?**
- Before knowing what is Producer-Consumer Problem we have to know what are Producer and Consumer.
- In operating System **Producer is a process which is able to produce data/item**.
- Consumer is a **Process that is able to consume the data/item produced by the Producer**.
- Both Producer and Consumer share a common memory buffer.
- This buffer is a space of a certain size in the memory of the system which is used for storage.
- The producer produces the data into the buffer and the consumer consumes the data from the buffer.

Producer-Consumer Problem



SCALER Topics

Producer-Consumer Problem

- So, what are the Producer-Consumer Problems?

- Producer Process should not produce any data when the shared buffer is full.
- Consumer Process should not consume any data when the shared buffer is empty.
- The access to the shared buffer should be mutually exclusive i.e at a time only one process should be able to access the shared buffer and make changes to it.
- For consistent data synchronization between Producer and Consumer, the above problem should be resolved.

Producer-Consumer Problem

- Solution For Producer Consumer Problem
- To solve the Producer-Consumer problem three semaphores variable are used :

- Semaphores are variables used to indicate the number of resources available in the system at a particular time. semaphore variables are used to achieve `Process Synchronization.

Producer-Consumer Problem

- Solution For Producer Consumer Problem
- To solve the Producer-Consumer problem three semaphores variable are used :

- Semaphores are variables used to indicate the number of resources available in the system at a particular time. semaphore variables are used to achieve `Process Synchronization.
  - FULL
  - EMPTY
  - MUTEX

Producer-Consumer Problem

- **Full**
- The full variable is used to track the space filled in the buffer by the Producer process. It is initialized to 0 initially as initially no space is filled by the Producer process.
- **Empty**
- The Empty variable is used to track the empty space in the buffer. The Empty variable is initially initialized to the **BUFFER-SIZE** as initially, the whole buffer is empty.

Producer-Consumer Problem

- **Mutex**
- Mutex is used to achieve mutual exclusion. mutex ensures that at any particular time only the producer or the consumer is accessing the buffer.
- **Mutex** - mutex is a binary semaphore variable that has a value of 0 or 1.
- We will use the Signal() and wait() operation in the above-mentioned semaphores to arrive at a solution to the Producer-Consumer problem.
- **Signal()** - The signal function increases the semaphore value by 1. **Wait()** - The wait operation decreases the semaphore value by 1.

Producer-Consumer Problem

- Let's look at the code of Producer-Consumer Process
- The code for Producer Process is as follows :

```
void Producer(){
        while(true){
        // producer produces an item/data
        wait(Empty);
        wait(mutex);
        add();
        signal(mutex);
        signal(Full);
}}
```

Producer-Consumer Problem

- Let's understand the above Producer process code :
- **wait(Empty)**:
- Before producing items, the producer process checks for the empty space in the buffer.
- If the buffer is full producer process waits for the consumer process to consume items from the buffer.
- so, the producer process executes wait(Empty) before producing any item.

Producer-Consumer Problem

- **wait(mutex)**:
- Only one process can access the buffer at a time.
- So, once the producer process enters into the critical section of the code it decreases the value of mutex by executing wait(mutex) so that no other process can access the buffer at the same time.

Producer-Consumer Problem

- **add()** - This method adds the item to the buffer produced by the Producer process. once the Producer process reaches add function in the code, it is guaranteed that no other process will be able to access the shared buffer concurrently which helps in data consistency.
- **signal(mutex)** - Now, once the Producer process added the item into the buffer it increases the mutex value by 1 so that other processes which were in a busy-waiting state can access the critical section.
- **signal(Full)** - when the producer process adds an item into the buffer spaces is filled by one item so it increases the Full semaphore so that it indicates the filled spaces in the buffer correctly.

Interprocess Communication (IPC)

- Processes need to communicate with each other in many situations, for example, to count occurrences of a word in text file, output of grep command needs to be given to wc command, something like grep -o -i <word> <file> | wc -l.
- Inter-Process Communication or IPC is a mechanism that allows processes to communicate.
- It helps processes synchronize their activities, share information, and avoid conflicts while accessing shared resources.

Interprocess Communication (IPC)

Types of Process

Let us first talk about types of types of processes.

- **Independent process:** An independent process is not affected by the execution of other processes. Independent processes are processes that do not share any data or resources with other processes.
- No inter-process communication required here.

Interprocess Communication (IPC)

- **Co-operating process:**
- Interact with each other and share data or resources.
- A co-operating process can be affected by other executing processes.
- Inter-process communication (IPC) is a mechanism that allows processes to communicate with each other and synchronize their actions.
- The communication between these processes can be seen as a method of cooperation between them.

Interprocess Communication (IPC)

•Inter process communication (IPC) allows different programs or processes running on a computer to share information with each other.

•IPC allows processes to communicate by using different techniques like sharing memory, sending messages, or using files.

•It ensures that processes can work together without interfering with each other.

•Cooperating processes require an Inter Process Communication (IPC) mechanism that will allow them to exchange data and information.

Interprocess Communication (IPC)

•The two fundamental models of Inter Process Communication are:

•Shared Memory

•Message Passing

•Figure below shows a basic structure of communication between processes via the shared memory method and via the message passing method.

Interprocess Communication (IPC)

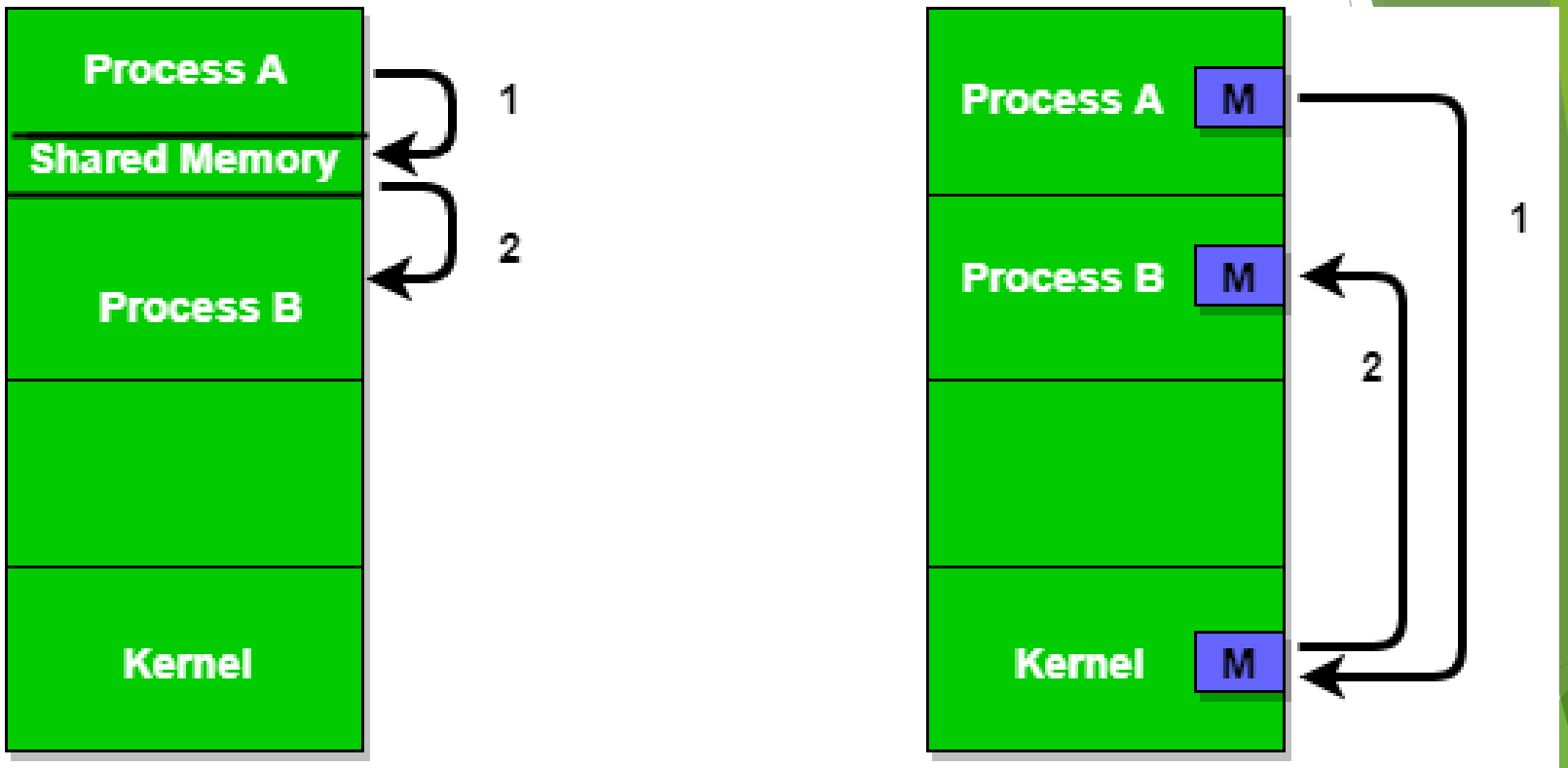


Figure 1 - Shared Memory and Message Passing

Interprocess Communication (IPC)

- An operating system can implement both methods of communication.
- First, we will discuss the shared memory methods of communication and then message passing.
- Communication between processes using shared memory requires processes to share some variable, and it completely depends on how the programmer will implement it. .

Interprocess Communication (IPC)

•One way of communication using shared memory can be imagined like this:

•Suppose process1 and process2 are executing simultaneously, and they share some resources or use some information from another process.

•Process1 generates information about certain computations or resources being used and keeps it as a record in shared memory.

•When process2 needs to use the shared information, it will check in the record stored in shared memory and take note of the information generated by process1 and act accordingly.

•Processes can use shared memory for extracting information as a record from another process as well as for delivering any specific information to other processes.

Interprocess Communication (IPC)

- Methods in Inter process Communication
- Inter-Process Communication refers to the techniques and methods that allow processes to exchange data and coordinate their activities.
- Since processes typically operate independently in a multitasking environment, IPC is essential for them to communicate effectively without interfering with one another.
- There are several methods of IPC, each designed to suit different scenarios and requirements.
- These methods include shared memory, message passing, semaphores, and signals, etc.

Interprocess Communication (IPC)

- Role of Synchronization in IPC
- In IPC, synchronization is essential for controlling access to shared resources and guaranteeing that processes do not conflict with one another.
- Data consistency is ensured and problems like race situations are avoided with proper synchronization.

Interprocess Communication (IPC)

- Advantages of IPC
- Enables processes to communicate with each other and share resources, leading to increased efficiency and flexibility.
- Facilitates coordination between multiple processes, leading to better overall system performance.
- Allows for the creation of distributed systems that can span multiple computers or networks.
- Can be used to implement various synchronization and communication protocols, such as semaphores, pipes, and sockets.

Interprocess Communication (IPC)

Disadvantages of IPC

•Increases system complexity, making it harder to design, implement, and debug.

•Can introduce security vulnerabilities, as processes may be able to access or modify data belonging to other processes.

•Requires careful management of system resources, such as memory and CPU time, to ensure that IPC operations do not degrade overall system performance.
Can lead to data inconsistencies if multiple processes try to access or modify the same data at the same time.

•Overall, the advantages of IPC outweigh the disadvantages, as it is a necessary mechanism for modern operating systems and enables processes to work together and share resources in a flexible and efficient manner. However, care must be taken to design and implement IPC systems carefully, in order to avoid potential security vulnerabilities and performance issues.

Interprocess Communication (IPC)

- **Conclusion**
- A fundamental component of contemporary operating systems, IPC allows processes to efficiently coordinate operations, share resources, and communicate.
- IPC is beneficial for developing adaptable and effective systems, despite its complexity and possible security threats.

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**
- **Non-blocking** is considered **asynchronous**
- **send** and **receive** primitives may be either blocking or non-blocking.

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.

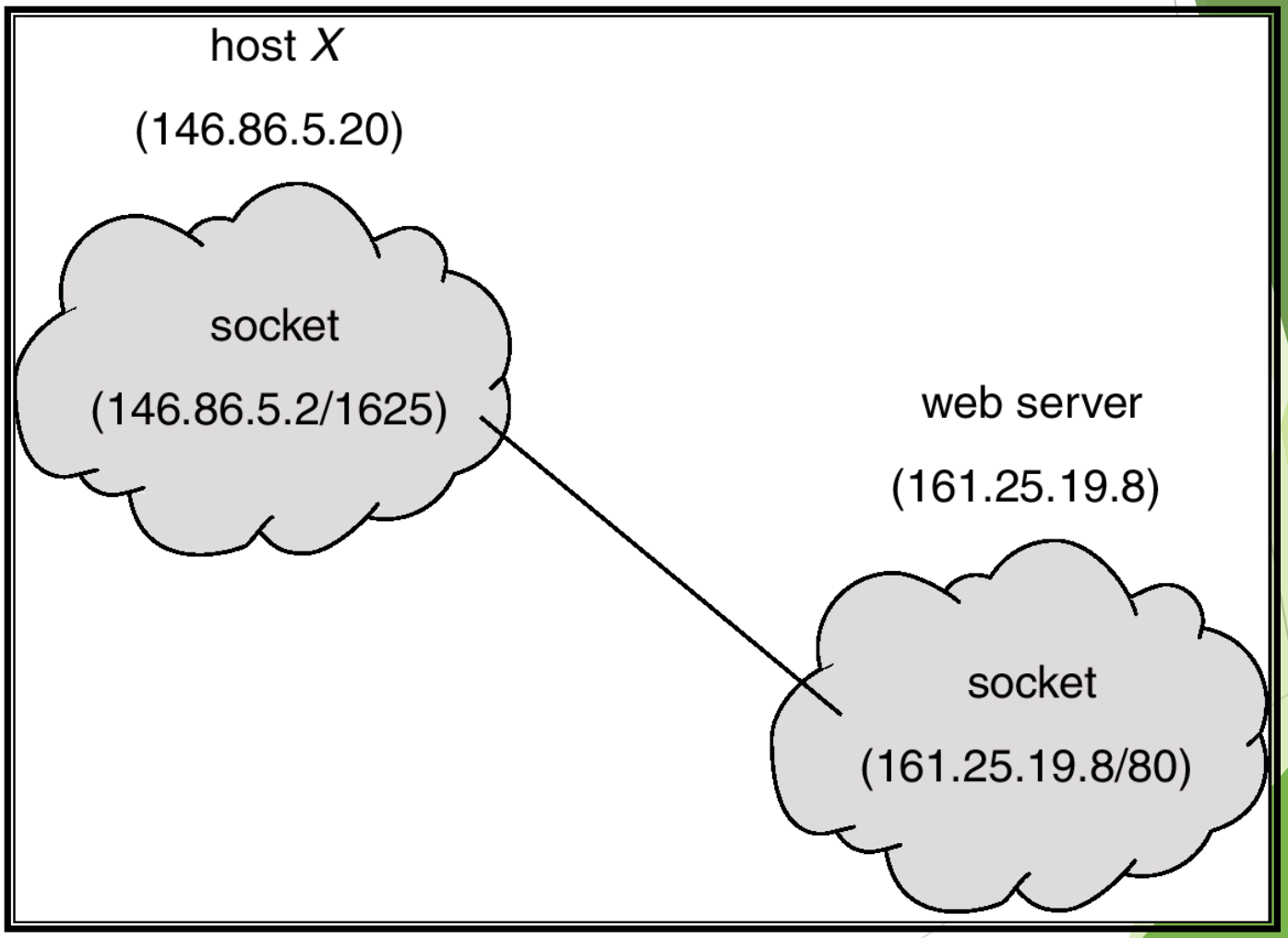
Client-Server Communication

- 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.

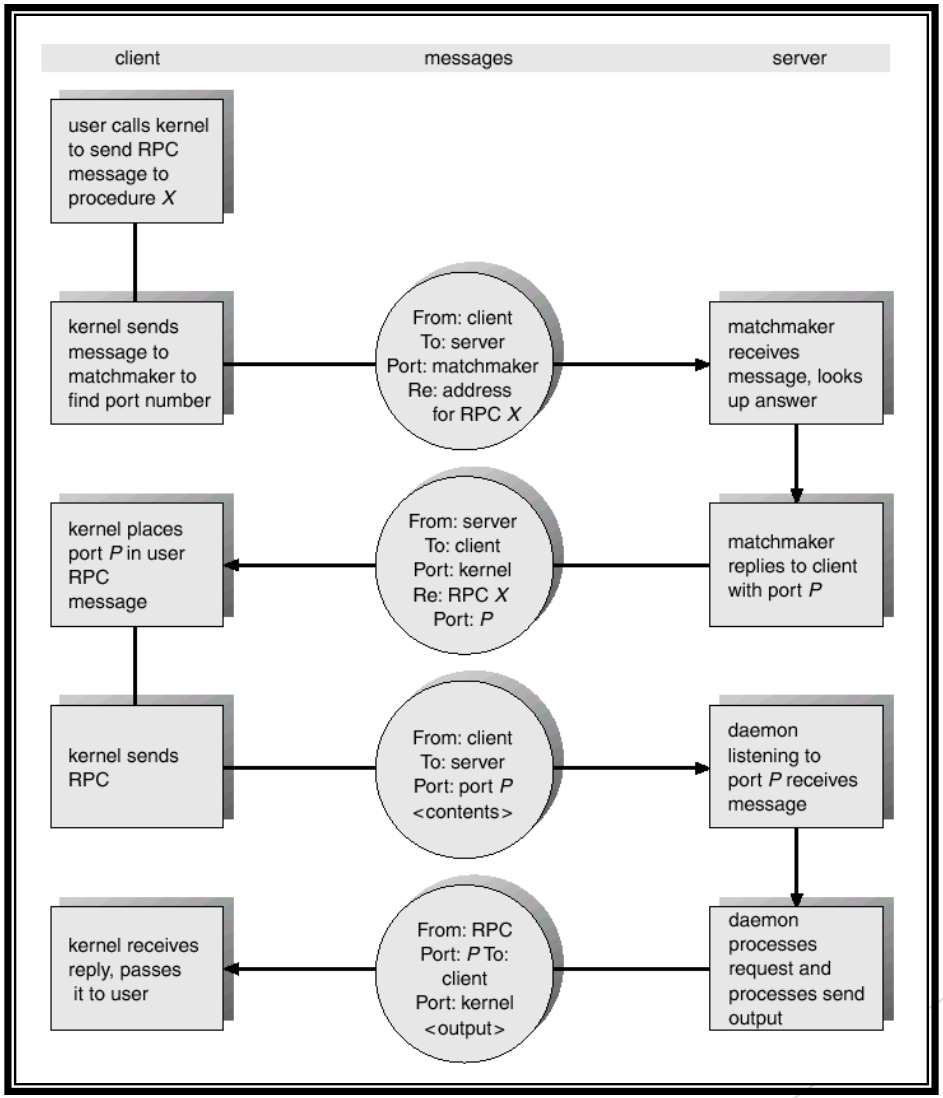
Socket Communication



Remote Procedure Calls

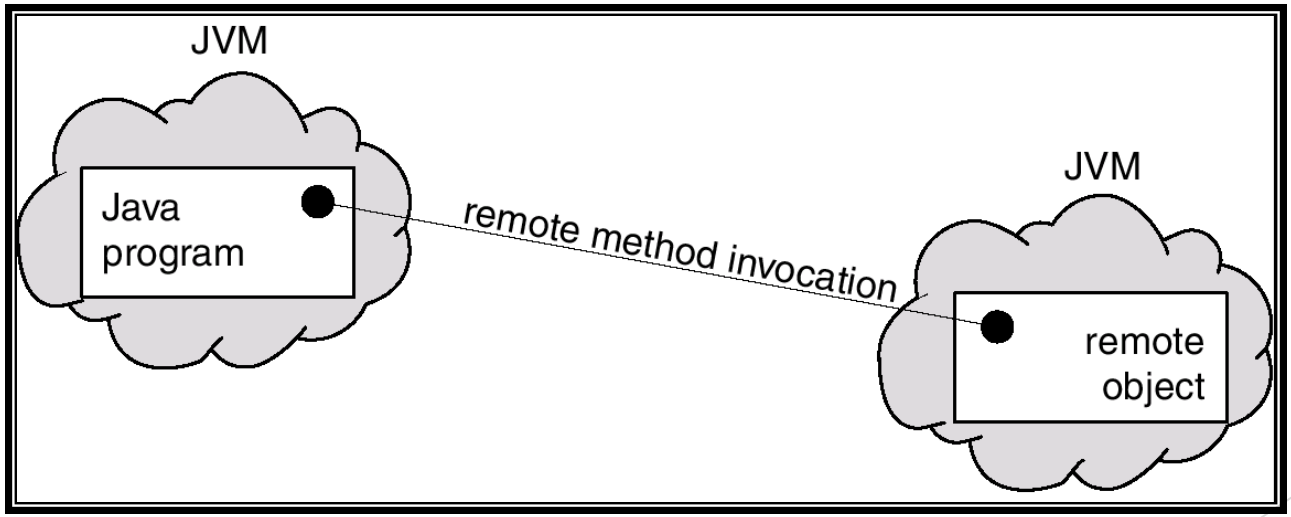
- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems.
- **Stubs** – client-side proxy for the actual procedure on the server.
- The client-side stub locates the server and *marshalls* the parameters.
- The server-side stub receives this message, unpacks the marshalled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