CHAPTER - 3

Processes

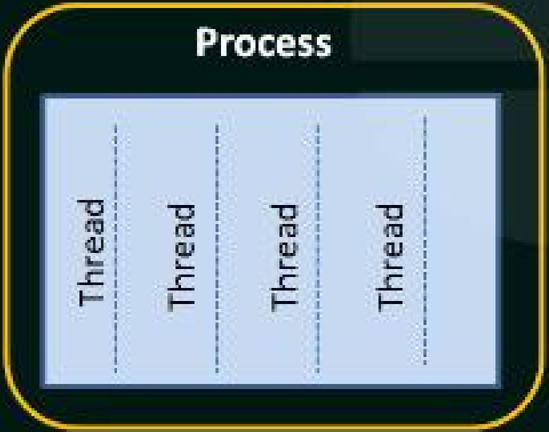
Neso Academy

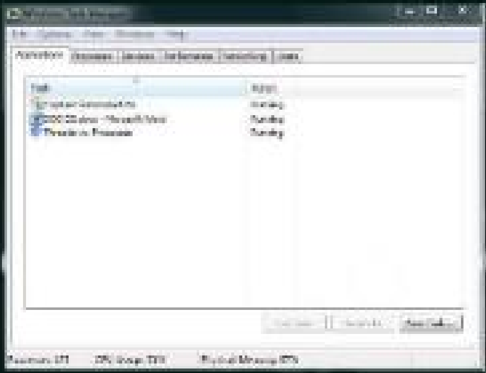
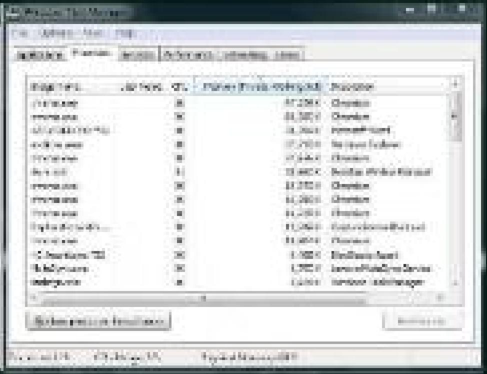
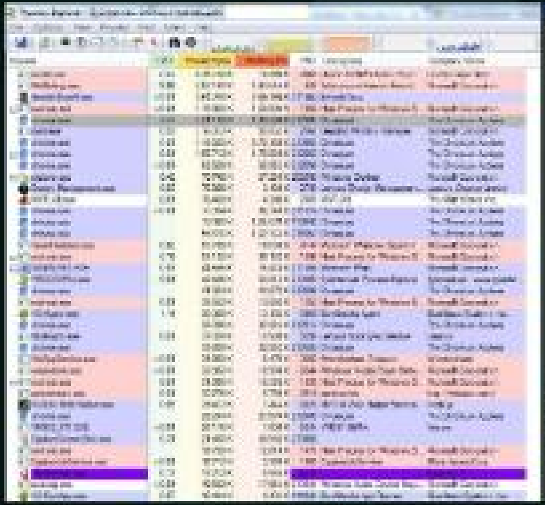
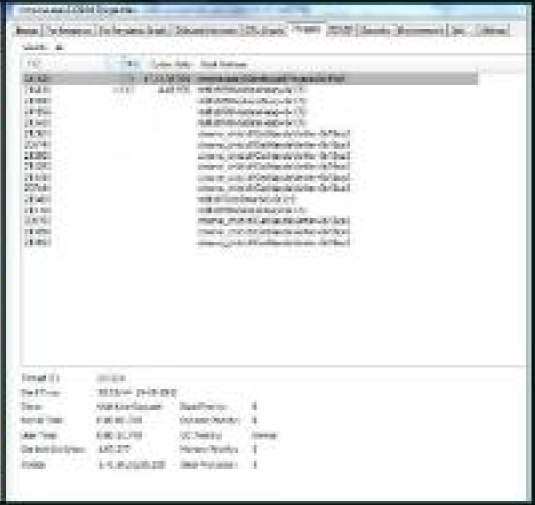
Process Management

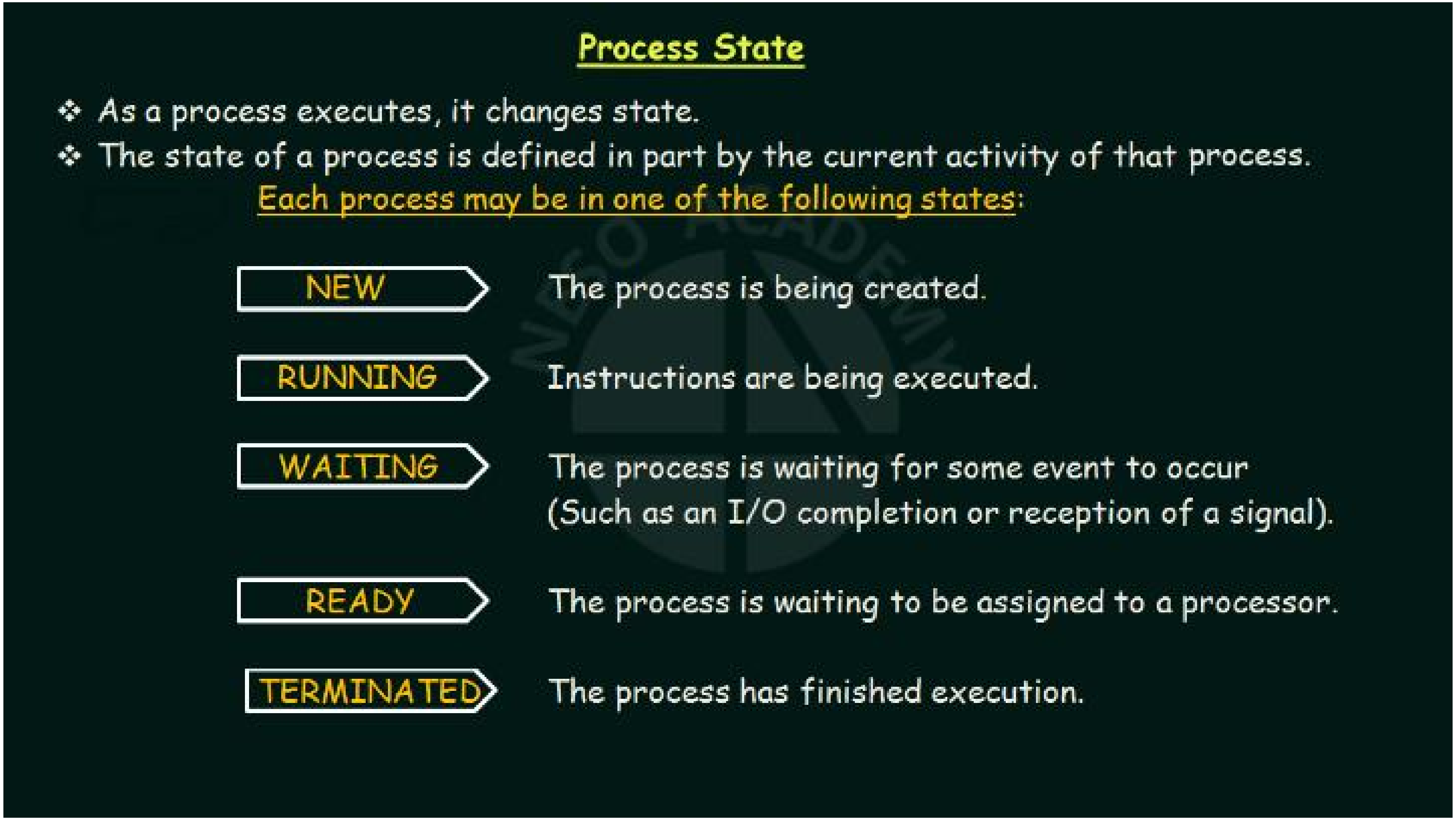
(Processes and Threads)

Process:
A process can be thought of as a program in execution.

Thread:
A thread is the unit of execution within a process. A process can have anywhere from just one thread to many threads.





Process State

- As a process executes, it changes state.
- The state of a process is defined in part by the current activity of that process.

Each process may be in one of the following states:

NEW	The process is being created.
RUNNING	Instructions are being executed.
WAITING	The process is waiting for some event to occur (Such as an I/O completion or reception of a signal).
READY	The process is waiting to be assigned to a processor.
TERMINATED	The process has finished execution.

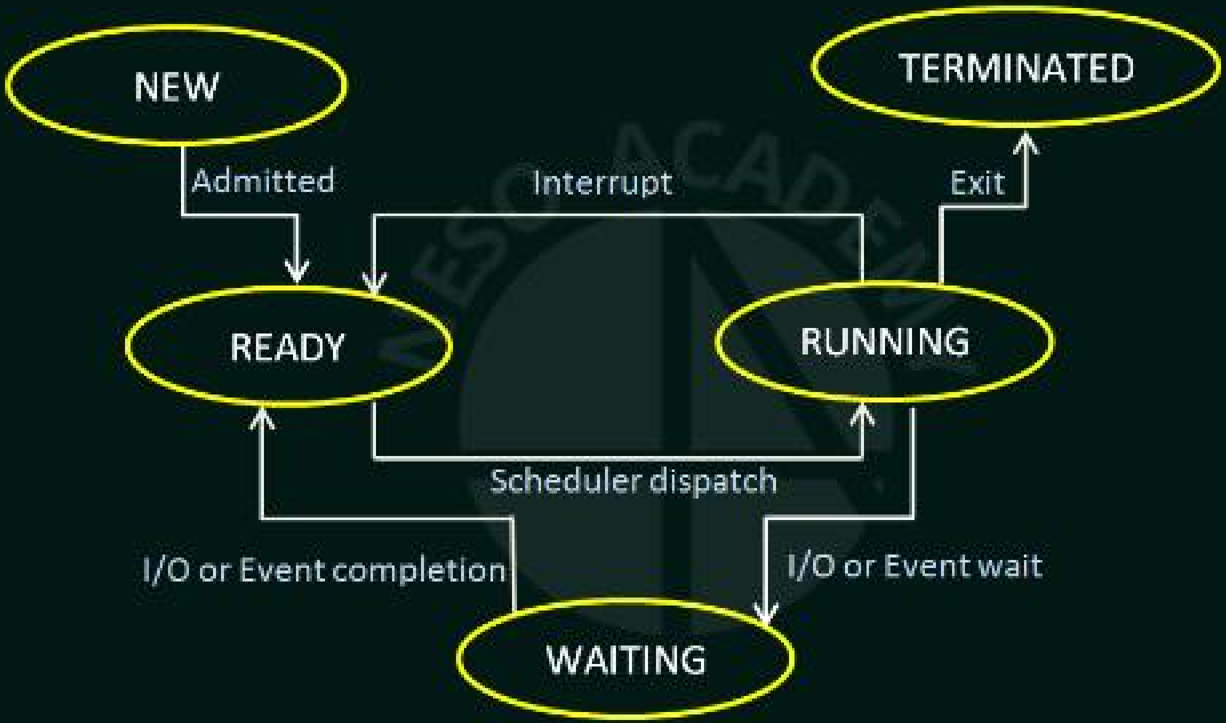
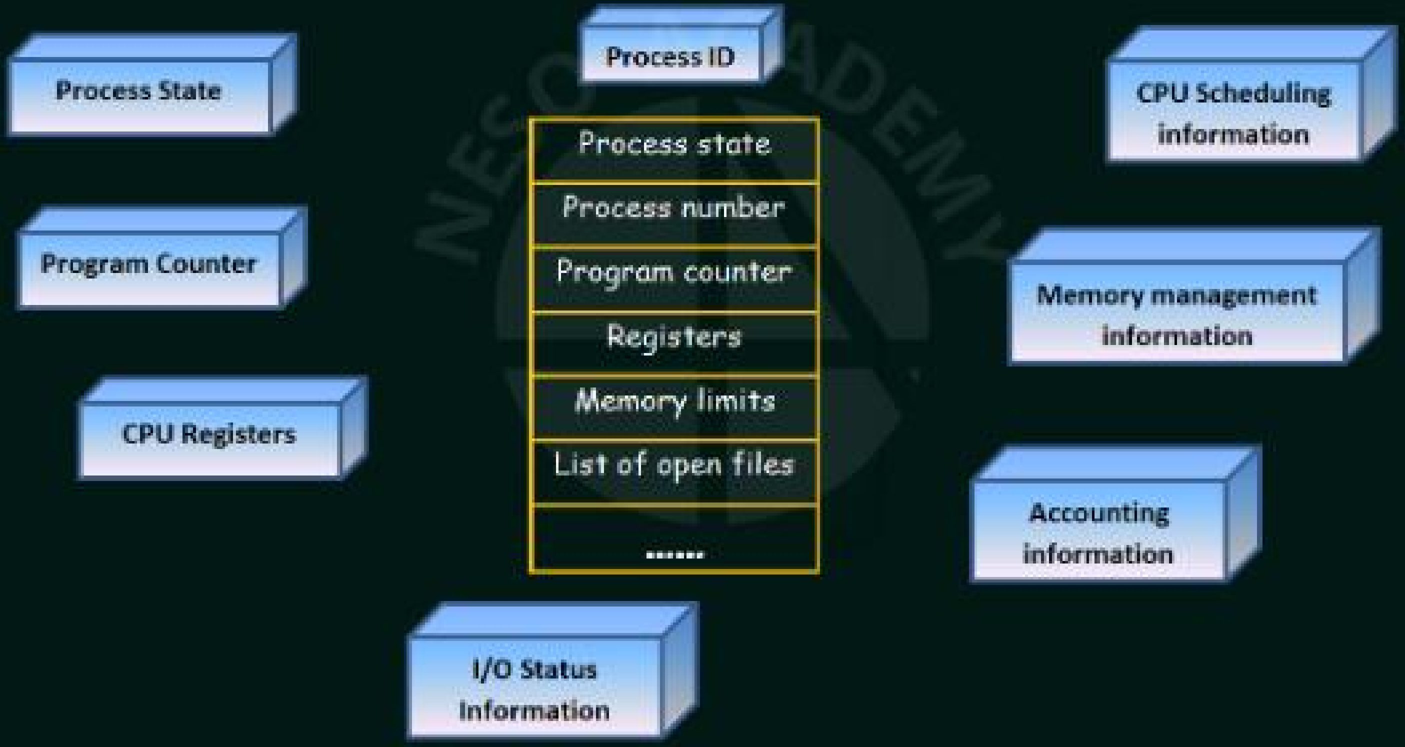


Fig: Diagram of process state

Process Control Block

Each process is represented in the operating system by a Process Control Block (PCB) — also called a task control block.



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.

- To meet these objectives, the process scheduler selects an available process (possibly from a set of several available processes) for program execution on the CPU.
    - 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.

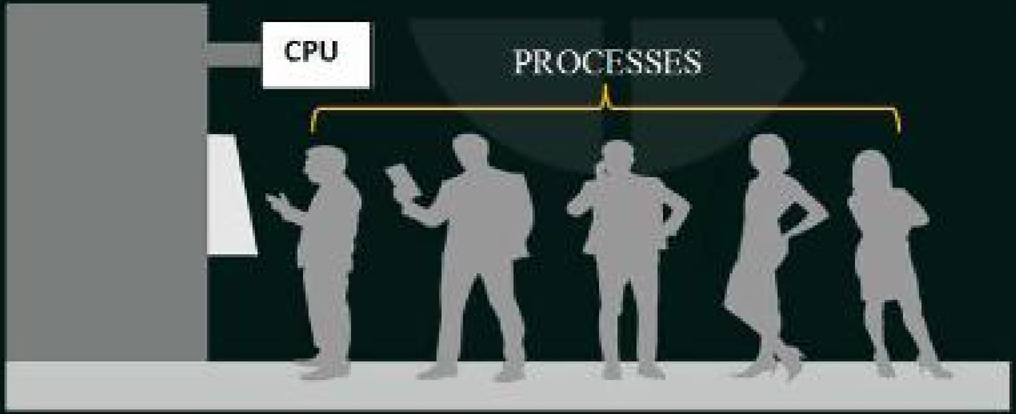
Scheduling Queues

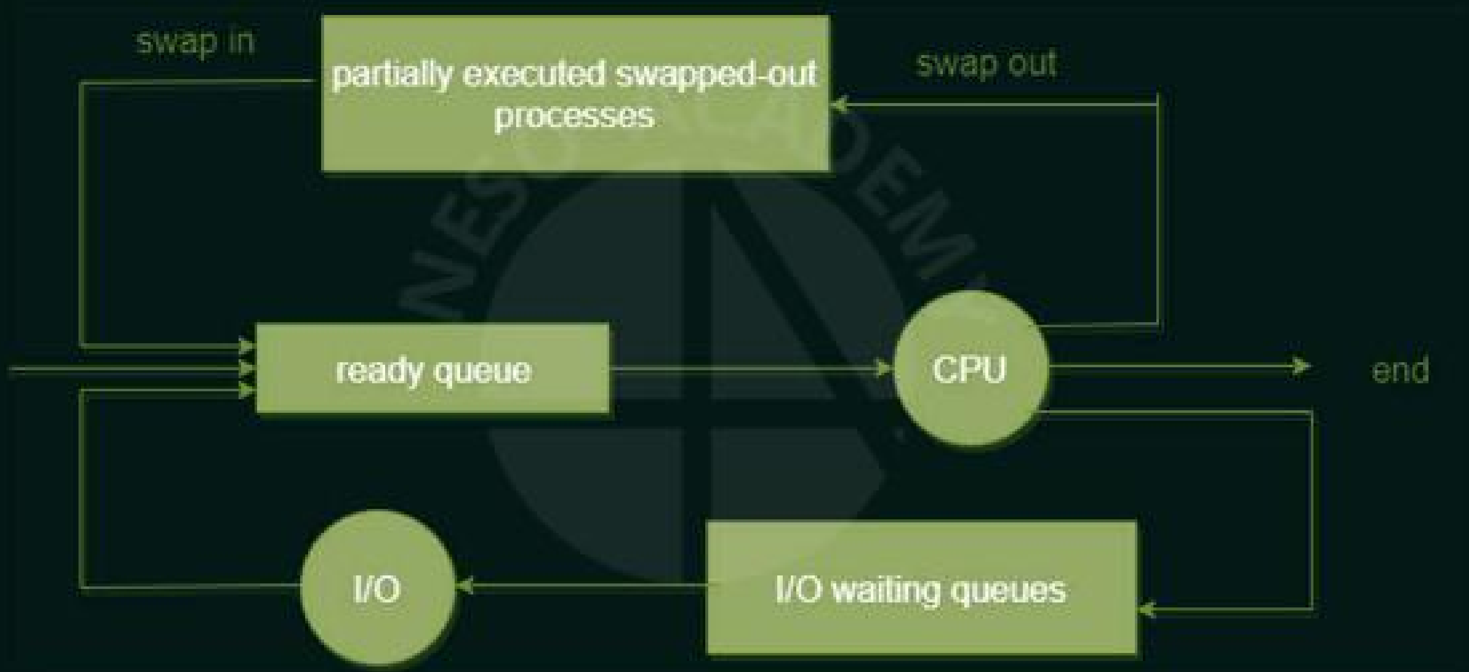
JOB QUEUE

As processes enter the system, they are put into a **job queue,** which consists of all processes in the system.

READY QUEUE

The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the **ready queue.**




swap in
partially executed swapped-out processes
swap out
ready queue
CPU
end
I/O
I/O waiting queues

Context Switch

- Interrupts cause the operating system to change a CPU from its current task and to run a kernel routine.

- Such operations happen frequently on general-purpose systems.

When an interrupt occurs, the system needs to save the current **context** of the process currently running on the CPU so that it can restore that context when its processing is done, essentially suspending the process and then resuming it.

- The context is represented in the PCB of the process

Process Control Block

Switching the CPU to another process requires performing a state save of the current process and a state restore of a different process.

State Save

State Restore

This task is known as a **context switch**.

- Context-switch time is pure overhead, because the system does no useful work while switching.
- Its speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied, and the existence of special instructions (such as a single instruction to load or store all registers).
- Typical speeds are a few milliseconds.

Operations on Processes

(Process Creation)

- A process may create several new processes, via a create-process system call, during the course of execution.
- The creating process is called a parent process, and the new processes are called the children of that process.
- Each of these new processes may in turn create other processes, forming a tree of processes.

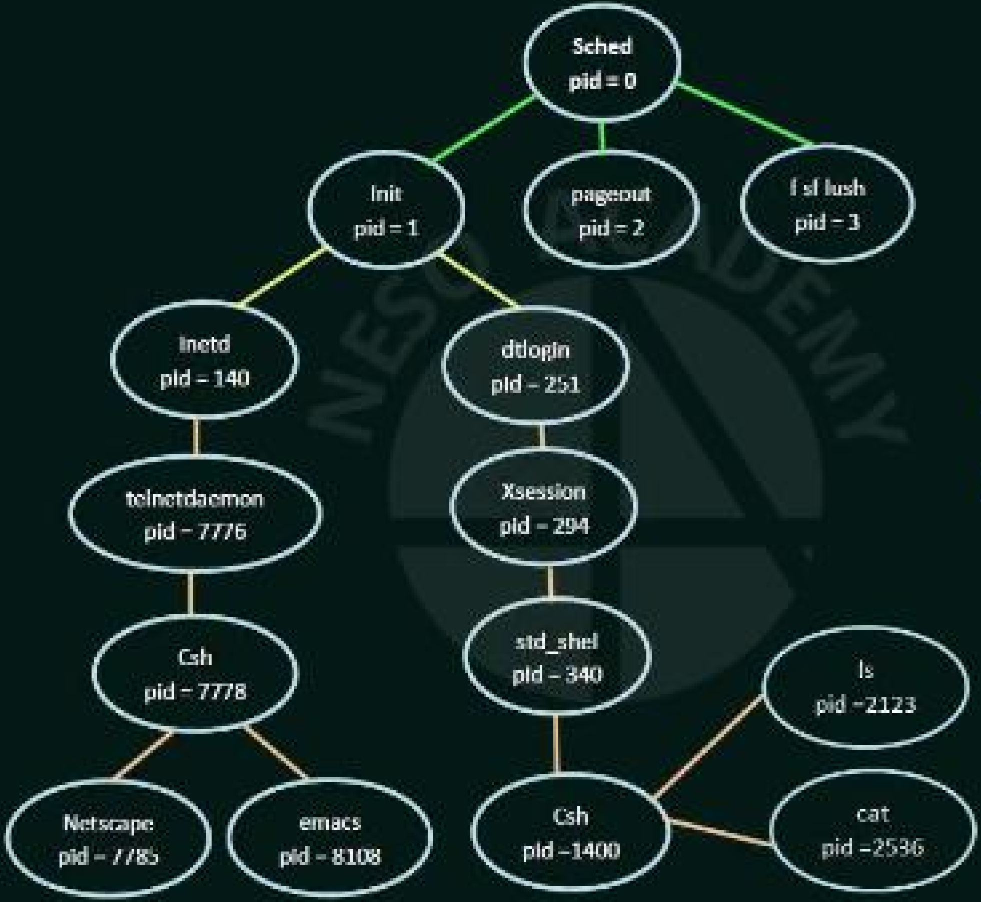


Figure:
A tree of processes
on a
typical Solaris system

When a process creates a new process, two possibilities exist in terms of execution:

1. The parent continues to execute concurrently with its children.

2. The parent waits until some or all of its children have terminated.

There are also two possibilities in terms of the address space of the new process:

1. The child process is a duplicate of the parent process (it has the same program and data as the parent).

2. The child process has a new program loaded into it.

Operations on Processes

(Process Termination)

- A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the exit () system call.
- At that point, the process may return a status value (typically an integer) to its parent process (via the wait() system call).
- All the resources of the process—including physical and virtual memory, open files, and I/O buffers—are deallocated by the operating system.

Termination can occur in other circumstances as well:

- A process can cause the termination of another process via an appropriate system call.
- Usually, such a system call can be invoked only by the parent of the process that is to be terminated.
- Otherwise, users could arbitrarily kill each other's jobs.

A parent may terminate the execution of one of its children for a variety of reasons, such as these:

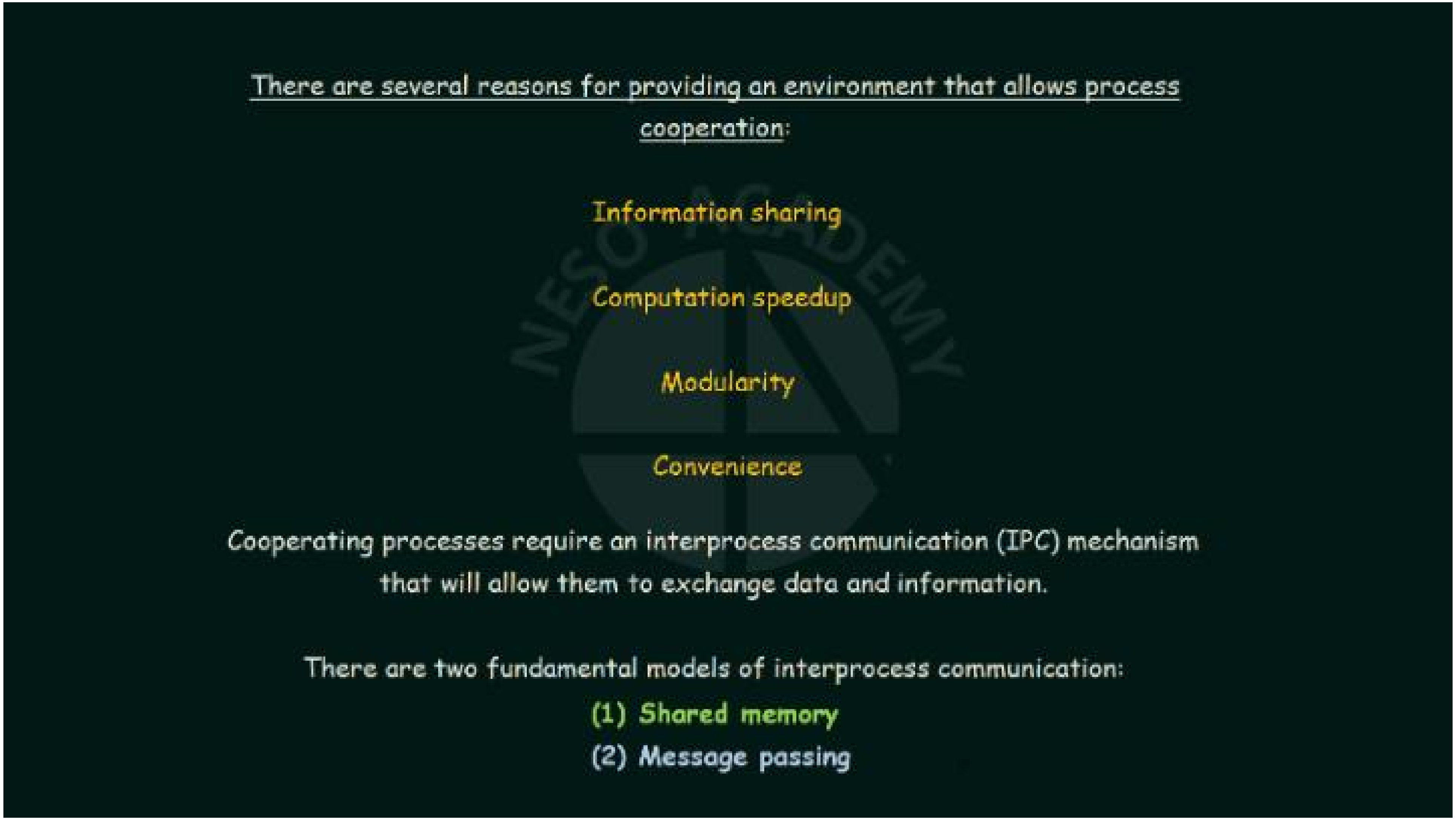
- The child has exceeded its usage of some of the resources that it has been allocated. *(To determine whether this has occurred, the parent must have a mechanism to inspect the state of its children.)*
- The task assigned to the child is no longer required.
- The parent is exiting, and the operating system does not allow a child to continue if its parent terminates.

Interprocess Communication

Processes executing concurrently in the operating system may be either independent processes or cooperating processes.

Independent processes -	They cannot affect or be affected by the other processes executing in the system.
Cooperating processes -	They can affect or be affected by the other processes executing in the system.

Any process that shares data with other processes is a cooperating process.



There are several reasons for providing an environment that allows process cooperation:

Information sharing

Computation speedup

Modularity

Convenience

Cooperating processes require an interprocess communication (IPC) mechanism that will allow them to exchange data and information.

There are two fundamental models of interprocess communication:

(1) Shared memory

(2) Message passing

- In the shared-memory model, a region of memory that is shared by cooperating processes is established.
  Processes can then exchange information by reading and writing data to the shared region.

- In the message passing model, communication takes place by means of messages exchanged between the cooperating processes.

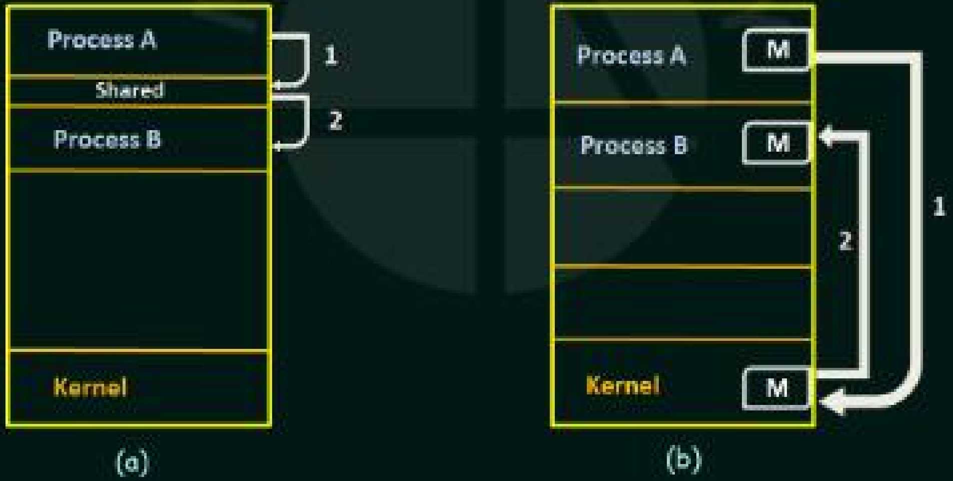
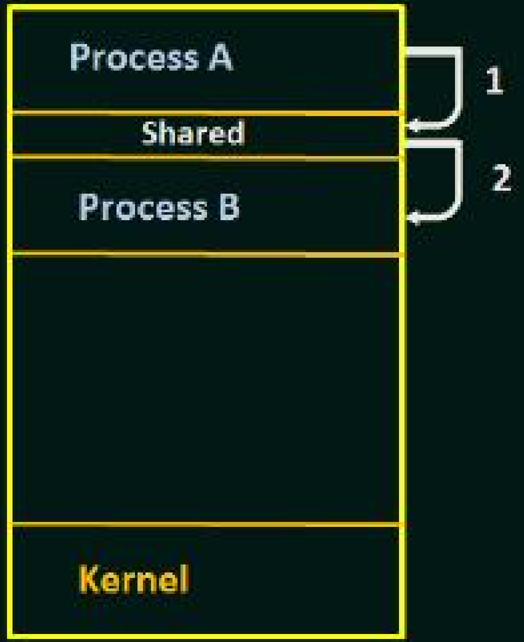
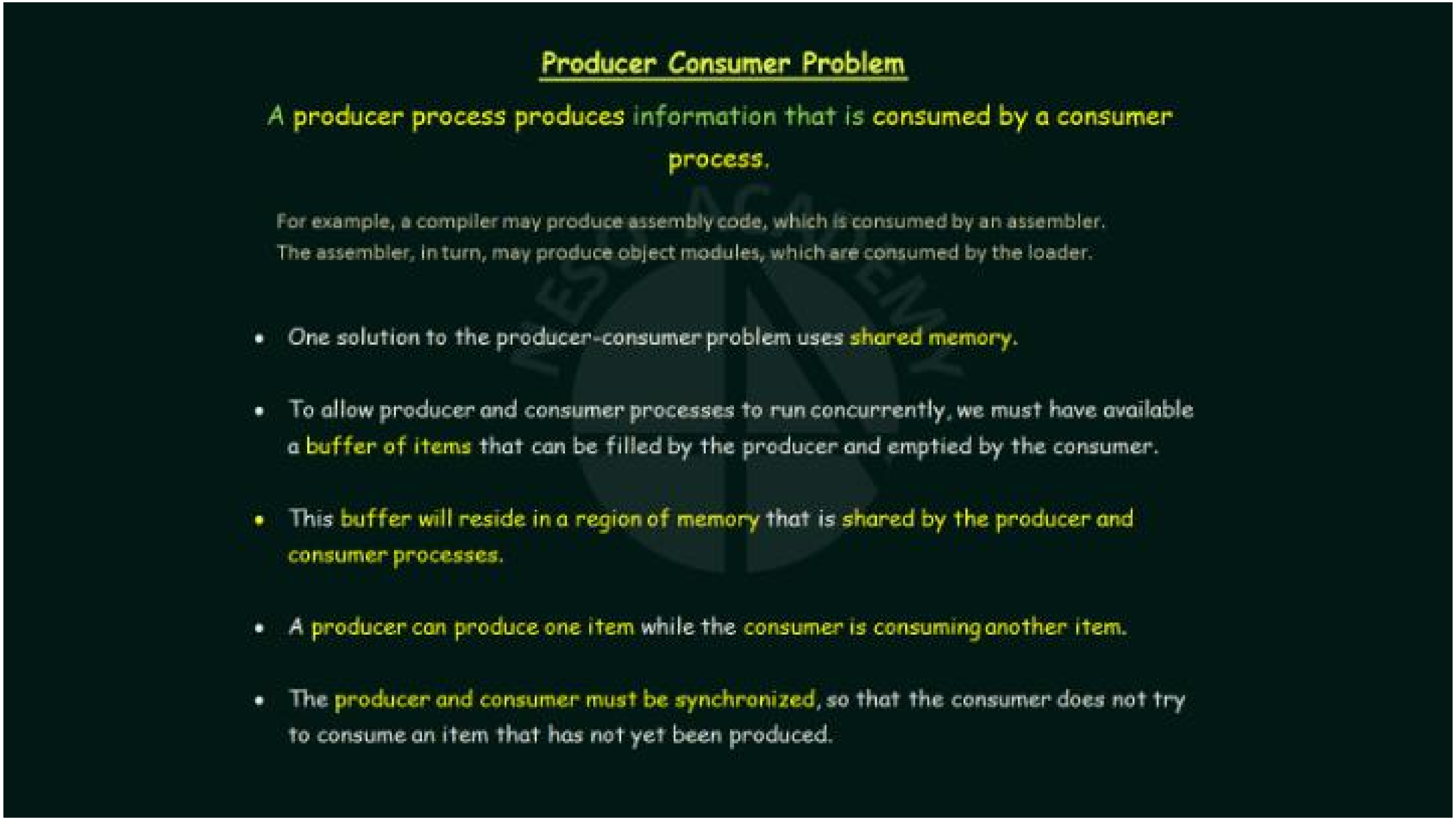


Fig: Communications models, (a) Shared memory, (b) Message Passing.

Shared Memory Systems

- Interprocess communication using shared memory requires communicating processes to establish a region of shared memory.
- Typically, a shared-memory region resides in the address space of the process creating the shared-memory segment.
- Other processes that wish to communicate using this shared-memory segment must attach it to their address space.
- Normally, the operating system tries to prevent one process from accessing another process's memory.
- Shared memory requires that two or more processes agree to remove this restriction.





Producer Consumer Problem

A producer process produces information that is consumed by a consumer process.

For example, 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.

- One solution to the producer-consumer problem uses shared memory.
- To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer.
- This buffer will reside in a region of memory that is shared by the producer and consumer processes.
- A producer can produce one item while the consumer is consuming another item.
- The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.

Two kinds of buffers:

Unbounded buffer

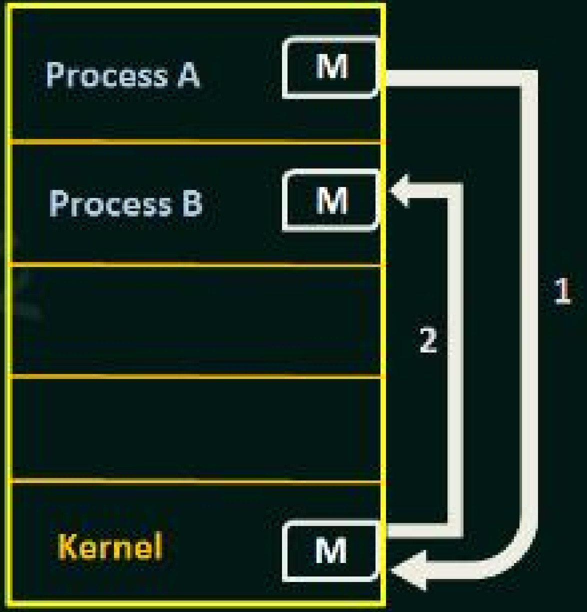
Places no practical limit on the size of the buffer. The consumer may have to wait for new items, but the producer can always produce new items.

Bounded buffer

Assumes a fixed buffer size. In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.

Message-Passing Systems (Part-1)

Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space and is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network.



A message-passing facility provides at least two operations:

- send (message)

  and

- receive (message)

Messages sent by a process can be of either fixed or variable size.

FIXED SIZE: The system-level implementation is straightforward. But makes the task of programming more difficult.

VARIABLE SIZE: Requires a more complex system-level implementation. But the programming task becomes simpler.

If processes P and Q want to communicate, they must send messages to and receive messages from each other.

A $\boxed{\text{communication link}}$ must exist between them.

This link can be implemented in a variety of ways. There are several methods for logically implementing a link and the **send( ) /receive( )** operations, like:

- Direct or indirect communication
- Synchronous or asynchronous communication
- Automatic or explicit buffering

There are several issues related with features like:

- **Naming**
- **Synchronization**
- **Buffering**

Naming

Processes that want to communicate must have a way to refer to each other. They can use either **direct** or **indirect** communication.

Under direct communication- Each process that wants to communicate must explicitly name the recipient or sender of the communication.

- send (P, message) - Send a message to process P.
- receive (Q, message) - Receive a message from process Q.

A communication link in this scheme has the following properties:

- A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate.
- A link is associated with exactly two processes.
- Between each pair of processes, there exists exactly one link.

This scheme exhibits **symmetry in addressing**; that is, both the sender process and the receiver process must name the other to communicate

Another variant of Direct Communication- Here, only the sender names the recipient; the recipient is not required to name the sender.

- send (P, message) - Send a message to process P.
- receive (id, message) - Receive a message from any process; *the variable id is set to the name of the process with which communication has taken place.*

This scheme employs **asymmetry in addressing**.

The disadvantage in both of these schemes (symmetric and asymmetric) is the limited modularity of the resulting process definitions. Changing the identifier of a process may necessitate examining all other process definitions.

With indirect communication:

The messages are sent to and received from **mailboxes**, or ports.

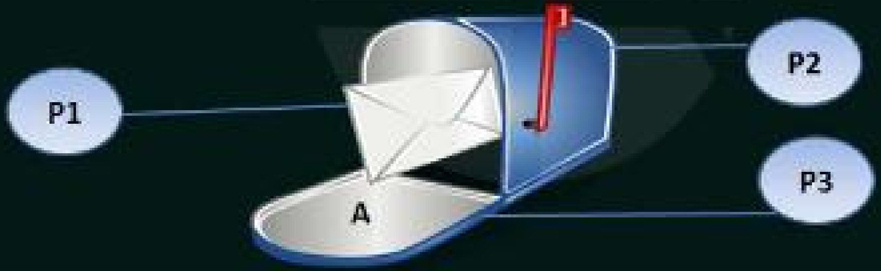
- A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed.
- Each mailbox has a unique identification.
- Two processes can communicate only if the processes have a shared mailbox

  - send (A, message) — Send a message to mailbox A.
  - receive (A, message) — Receive a message from mailbox A.

A communication link in this scheme has the following properties:

- A link is established between a pair of processes only if both members of the pair have a shared mailbox.
- A link may be associated with more than two processes.
- Between each pair of communicating processes, there may be a number of different links, with each link corresponding to one mailbox.

Now suppose that processes P1, P2, and P3 all share mailbox A



Process P1 sends a message to A, while both P2 and P3 execute a receive() from A. **Which process will receive the message sent by P1?**

The answer depends on which of the following methods we choose:

- Allow a link to be associated with two processes at most.
- Allow at most one process at a time to execute a receive () operation.
- Allow the system to select arbitrarily which process will receive the message (that is, either P2 or P3, but not both, will receive the message). The system also may define an algorithm for selecting which process will receive the message (that is, round robin where processes take turns receiving messages). The system may identify the receiver to the sender.

A **mailbox** may be **owned** either by a **process** or by the **operating system.**

Synchronization

Communication between processes takes place through calls to send() and receive () primitives. There are different design options for implementing each primitive.

Message passing may be either **blocking** or **nonblocking**— also known as **synchronous** and **asynchronous**.

Blocking send:	The sending process is blocked until the message is received by the receiving process or by the mailbox.
Nonblocking send:	The sending process sends the message and resumes operation.
Blocking receive:	The receiver blocks until a message is available.
Nonblocking receive:	The receiver retrieves either a valid message or a null.

Buffering

Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue.

Basically, such queues can be implemented in three ways:

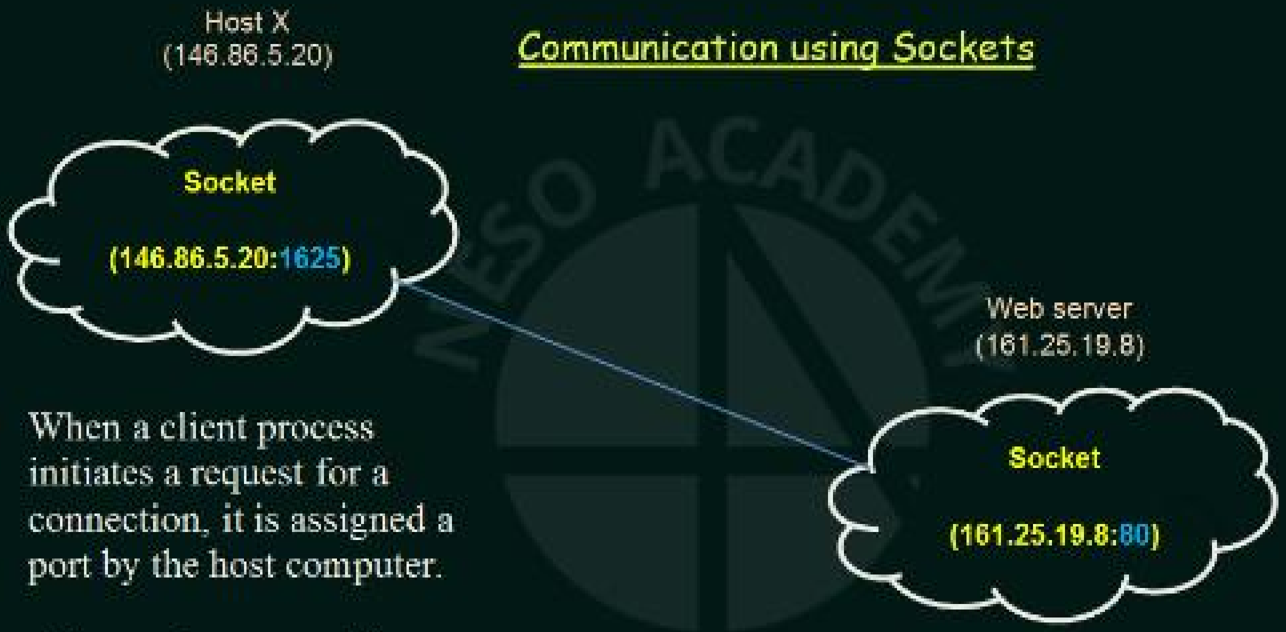
Zero capacity:	The queue has a maximum length of zero; thus, the link cannot have any messages waiting in it. In this case, the sender must block until the recipient receives the message.
Bounded capacity:	The queue has finite length n; thus, at most n messages can reside in it. If the queue is not full when a new message is sent, the message is placed in the queue and the sender can continue execution without waiting. The links capacity is finite, however. If the link is full, the sender must block until space is available in the queue.
Unbounded capacity:	The queues length is potentially infinite; thus, any number of messages can wait in it. The sender never blocks.

Sockets

Used for communication in Client-Server Systems

- A socket is defined as an endpoint for communication.
- A pair of processes communicating over a network employ a pair of sockets—one for each process.
- A socket is identified by an IP address concatenated with a port number.
- The server waits for incoming client requests by listening to a specified port. Once a request is received, the server accepts a connection from the client socket to complete the connection.
- Servers implementing specific services (such as telnet, ftp, and http) listen to well-known ports
  (a telnet server listens to port 23, an ftp server listens to port 21, and a web, or http, server listens to port 80).
- All ports below 1024 are considered well known; we can use them to implement standard services

Communication using Sockets



- When a client process initiates a request for a connection, it is assigned a port by the host computer.
- This port is some arbitrary number greater than 1024.

The packets traveling between the hosts are delivered to the appropriate process based on the destination port number.

Remote Procedure Calls (RPC)

Remote Procedure Call (RPC) is a protocol that one program can use to request a service from a program located in another computer on a network without having to understand the network's details.

- It is similar in many respects to the IPC mechanism.
- However,because we are dealing with an environment in which the processes are executing on separate systems, we must use a $\boxed{\text{message based communication scheme}}$ to provide remote service.
- In contrast to the IPC facility, the messages exchanged in RPC communication are well structured and are thus no longerjust packets of data.
- Each message is addressed to an RPC daemon listening to a port on the remote system, and each contains an identifier of the function to execute and the parameters to pass to that function.
- The function is then executed as requested, and any output is sent back to the requester in a separate message.

The semantics of RPCs allow a client to invoke a procedure on a remote host as it would invoke a procedure locally

- The RPC system hides the details that allow communication to take place by providing a stub on the client side.
- Typically, a separate stub exists for each separate remote procedure.
- When the client invokes a remote procedure, the RPC system calls the appropriate stub, passing it the parameters provided to the remote procedure. This stub locates the port on the server and marshals the parameters.
- Parameter marshalling involves packaging the parameters into a form that can be transmitted over a network.
- The stub then transmits a message to the server using message passing.
- A similar stub on the server side receives this message and invokes the procedure on the server.
- If necessary, return values are passed back to the client using the same technique.

Issues in RPC and how they are resolved

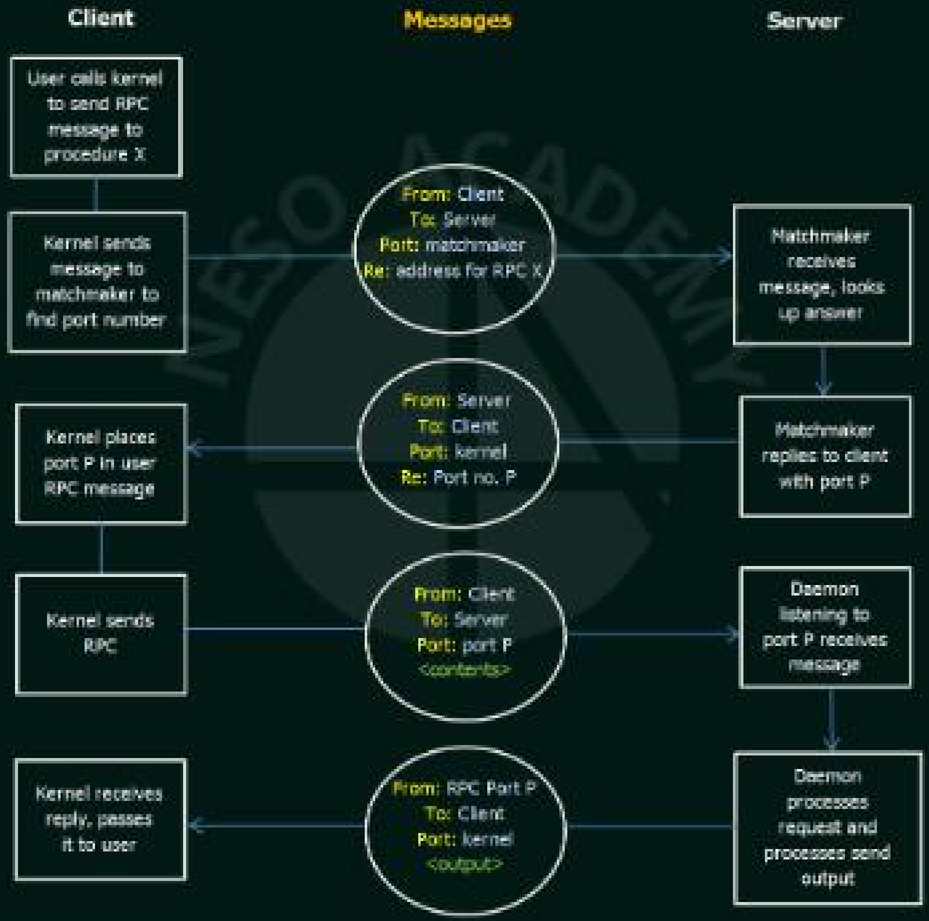
Issues	How they are resolved
Differences in data representation on the client and server machines. Eg. Representation of 32-bit integers: Some systems (known as big-endian) use the high memory address to store the most significant byte, while other systems (known as little-endian) store the least significant byte at the high memory address.	RPC systems define a machine-independent representation of data. One such representation is known as external data representation (XDR). On the client side, parameter marshalling involves converting the machine dependent data into XDR before they are sent to the server. On the server side, the XDR data are unmarshalled and converted to the machine-dependent representation for the server.
Whereas local procedure calls fail only under extreme circumstances, RPCs can fail, or be duplicated and executed more than once, as a result of common network errors.	The operating system must ensure that messages are acted on exactly once, rather than at most once. Most local procedure calls have the "exactly once" functionality, but it is more difficult to implement.

With standard procedure calls, some form of binding takes place during link, load, or execution time so that a procedure call's name is replaced by the memory address of the procedure call. The RPC scheme requires a similar binding of the client and the server port, but how does a client know the port numbers on the server? Neither system has full information about the other because they do not share memory.

1) The binding information may be predetermined, in the form of fixed port addresses. At compile time, an RPC call has a fixed port number associated with it. Once a program is compiled, the server cannot change the port number of the requested service.

2) Binding can be done dynamically by a rendezvous mechanism. Typically, an operating system provides a rendezvous (also called a matchmaker) daemon on a fixed RPC port. A client then sends a message containing the name of the RPC to the rendezvous daemon requesting the port address of the RPC it needs to execute. The port number is returned, and the RPC calls can be sent to that port until the process terminates (or the server crashes).

Execution of a remote procedure call (RPC)



Client
Messages
Server
User calls kernel to send RPC message to procedure X
Kernel sends message to matchmaker to find port number
From: Client To: Server Port: matchmaker Re: address for RPC X
Matchmaker receives message, looks up answer
Matchmaker replies to client with port P
From: Server To: Client Port: kernel Re: Port no. P
Kernel places port P in user RPC message
Kernel sends RPC
From: Client To: Server Port: port P <contents>
Daemon listening to port P receives message
Daemon processes request and processes send output
From: RPC Port P To: Client Port: kernel <output>
Kernel receives reply, passes it to user