PROCESS MANAGEMENT

- A *process* can be thought of as a program in execution.
- A process will need certain resources—such as CPU time, memory, files, and I/O devices —to accomplish its task.
- These resources are allocated to the process either when it is created or while it is executing.

1. Process Concept

a. The Process :

- A process is a program in execution.
- A process is more than the program code, which is sometimes known as the **text section**.
- It also includes the current activity, as represented by the value of the **program counter** and the contents of the processor's registers.
- A process generally also includes the process **stack**, which contains temporary data (such as function parameters, return addresses, and local variables), and a **data section**, which contains global variables. A process may also include a **heap**,which is memory that is dynamically allocated during process run time.
- The structure of a process in memory is shown in Figure .

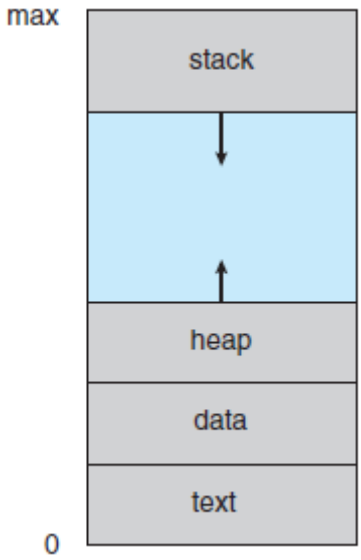

Fig : Process in Memory

- We emphasize that a program by itself is not a process.
- A program is a ***passive*** entity, such as a file containing a list of instructions stored on disk (often called an **executable file**).

- In contrast, a process is an ***active*** entity, with a program counter specifying the next instruction to execute and a set of associated resources.
- A program becomes a process when an executable file is loaded into memory.
- Two common techniques for loading executable files are **double-clicking an icon** representing the executable file and **entering the name of the executable file on the command line** (as in prog.exe or a.out).

b. 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.
- A 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.

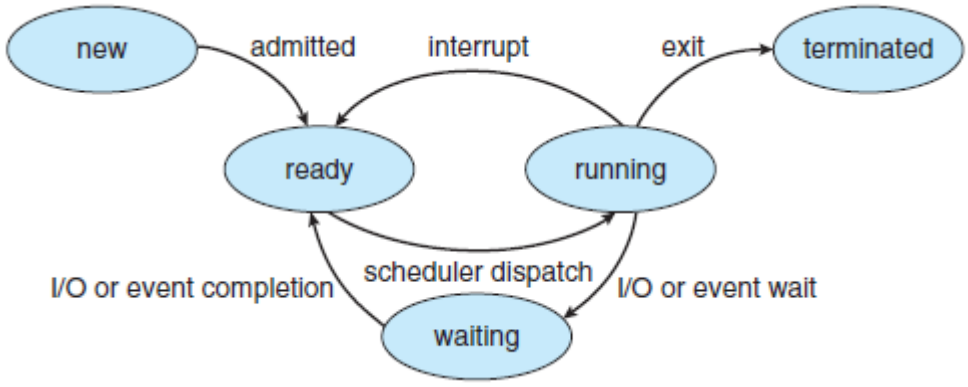

Fig: Process State

c. Process Control Block

- Each process is represented in the operating system by a **process control block (PCB)**—also called a **task control block**.
- A PCBis shown in Figure .

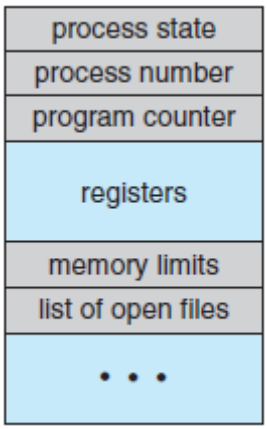

Fig : Process Control Block

- It contains many pieces of information associated with a specific process, including these:

✓ **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. (Fig : CPU switch from process to process)

✓ **CPU-scheduling information**. This information includes a process priority,pointers to scheduling queues, and any other scheduling parameters.

✓ **Memory-management information**. This information may include such items as the value of the base and limit registers and 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**. This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

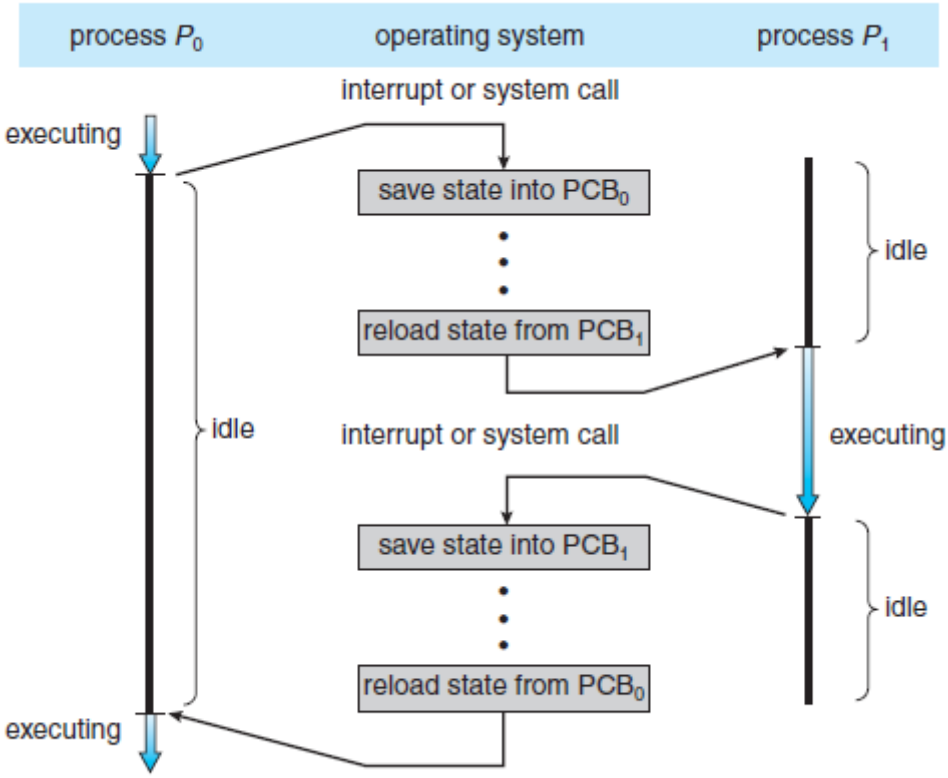

Fig : CPU switch from process to process

d. Threads :

- A process is a program that performs a single **thread** of execution.
- For example, when a process is running a word-processor program, a **single thread** of instructions is being executed.
- This single thread of control allows the process to perform only one task at a time. The user cannot simultaneously type in characters and run the spell checker within the same process
- For example. Most modern operating systems have extended the process concept to allow a process to have **multiple threads** of execution and thus to perform more than one task at a time.
- This feature is especially beneficial on multicore systems, where multiple threads can run in parallel.
- On a system that supports threads, the PCB is expanded to include information for each thread.

2. 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.

a. Scheduling Queues

Some scheduling queues maintained by os are

- ✓ **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

- As processes enter the system, they are put into a **job queue**, which consists of all processes in the system.
- The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the **ready queue**.
- This queue is generally stored as a linked list. A ready-queue header contains pointers to the first and final PCBs in the list.
- Suppose the process makes an I/O request 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/O device is called a **device queue**. Each device has its own device queue (as shown in below Figure)

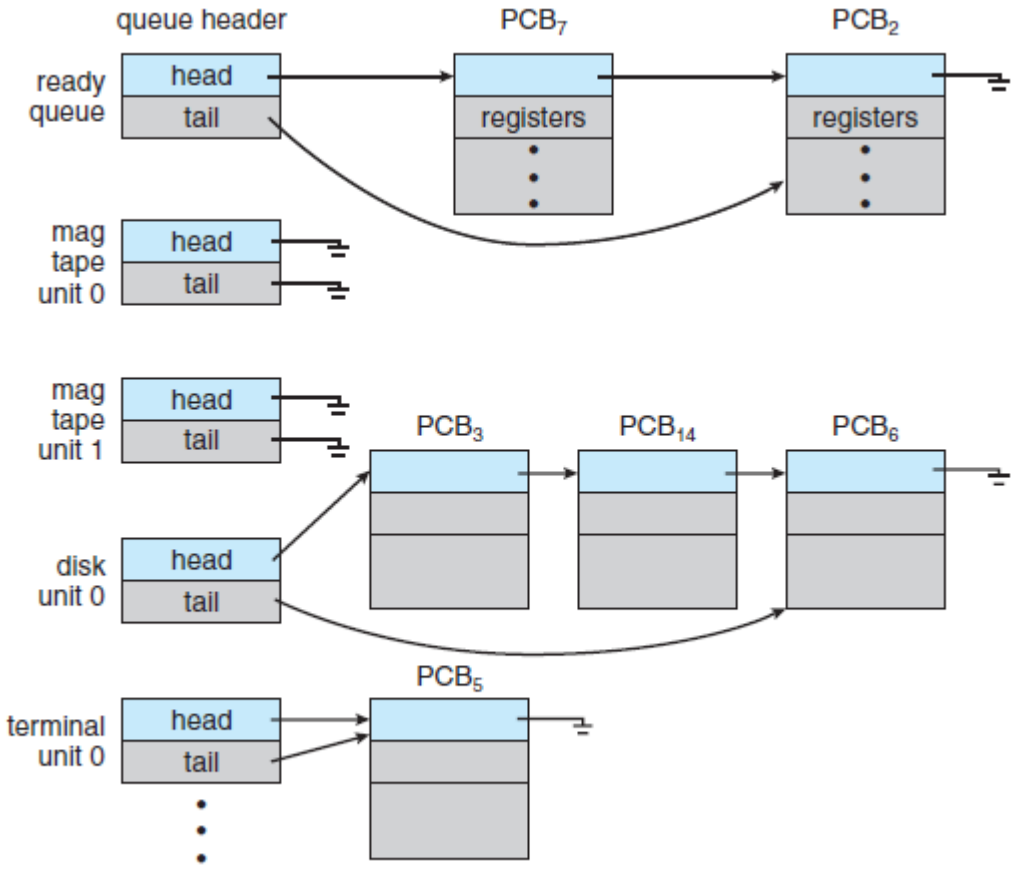

Fig : The ready queue and various I/O device queues.

- A common representation of process scheduling is a **queueing diagram**, as shown in below Figure .
- Each rectangular box represents a queue.
- Two types of queues are present: **the ready queue and a set of device queues.**
- The circles represent the resources that serve the queues, and the arrows indicate the flow of processes in the system.
- A new process is initially put in the ready queue.
- It waits there until it is selected for execution, or **dispatched**.
- Once the process is allocated the CPU and is executing, one of several events could occur:
  - The process could issue an I/O request and then be placed in an I/O queue.
  - The process could create a new child process and wait for the child's 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.

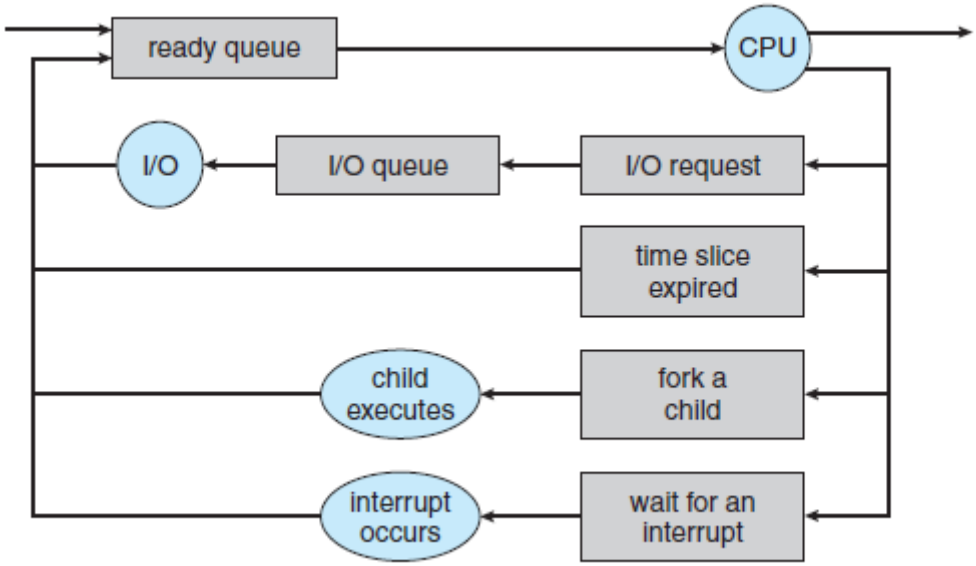

Fig : Queueing-diagram representation of process scheduling

b. Schedulers

- A process migrates among 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**.

- **Types of scheduler :**
  - Long-term scheduler
  - Short – term scheduler

- 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.

- 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

- Some operating systems, such as time-sharing systems, may introduce an additional, intermediate level of scheduling. This **medium-term scheduler** is shown in below Figure .
- The key idea behind a medium-term scheduler is that sometimes it can be advantageous to remove a process from memory (and from active contention for the CPU) and thus reduce the degree of multiprogramming.
- Later, the process can be reintroduced into memory, and its execution can be continued where it left off. This scheme is called **swapping**.
- The process is swapped out, and is later swapped in, by the medium-term scheduler.

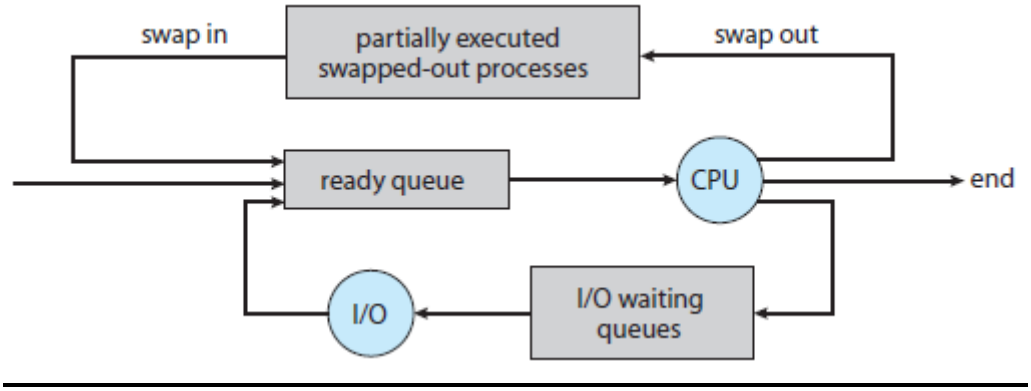

Fig : Addition of medium-term scheduling to the queueing diagram

c. 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 via a context switch
- Context of a process represented in the PCB
- Context-switch time is overhead; the system does no useful work while switching
- Time dependent on hardware support.

3. Operations on Processes

a. Process Creation :

- During the course of execution, a process may create several new processes.
- 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.
- Most operating systems (including UNIX, Linux, and Windows) identify processes according to a unique **process identifier** (or **pid**), which is typically an integer number
- In general, when a process creates a child process, that child process will need certain resources (CPU time, memory, files, I/O devices) to accomplish its task.
- A child process may be able to obtain its resources directly from the operating system, or it may be constrained to a subset of the resources of the parent process.
- The parent may have to partition its resources among its children, or it may be able to share some resources (such as memory or files) among several of its children
- When the process is created, it will get, as an input from its parent process, the name of the file ***image.jpg***.
- Using that file name, it will open the file and write the contents out. It may also get the name of the output device.
- When a process creates a new process, two possibilities for execution exist:
    - The parent continues to execute concurrently with its children.
    - The parent waits until some or all of its children have terminated.
- There are also two address-space possibilities for the new process:

- The child process is a duplicate of the parent process (it has the same program and data as the parent).
- The child process has a new program loaded into it.

- A new process is created by the **fork()** system call.
- The new process consists of a copy of the address space of the original process. This mechanism allows the parent process to communicate easily with its child process. Both processes (the parent and the child) continue execution at the instruction after the fork(), with one difference: the return code for the fork() is zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent.
- After a fork() system call, one of the two processes typically uses the exec() system call to replace the process's memory space with a new program.
- The exec() system call loads a binary file into memory (destroying the memory image of the program containing the exec() system call) and starts its execution.
- In this manner, the two processes are able to communicate and then go their separate ways. The parent can then create more children; or, if it has nothing else to do while the child runs, it can issue a wait() system call to move itself off the ready queue until the termination of the child.

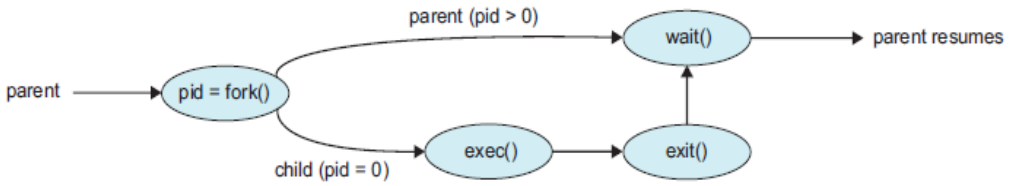

Fig : Process Creation using the fork() system call.

Example : Creating a separate process using the UNIX fork() system call.

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main()
{
pid t pid;
/* fork a child process */
pid = fork();
if (pid < 0) { /* error occurred */
fprintf(stderr, "Fork Failed");
```

```
return 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");
}
return 0;
}
```

b. 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.

- 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.

- Some systems do not allow a child to exist if its parent has terminated. In such systems, if a process terminates (either normally or abnormally),then all its children must also be terminated. This phenomenon, referred to as **Cascading termination**, is normally initiated by the operating system.

- When a process terminates, its resources are deallocated by the operating system. However, its entry in the process table must remain there until the parent calls wait(), because the process table contains the process's exit status. A process that has terminated, but whose parent has not yet called wait(), is known as a **zombie** process.

4. Interprocess Communication:

- Processes executing concurrently in the operating system may be either independent processes or cooperating processes.
- A process is ***independent*** if it cannot affect or be affected by the other processes executing in the system i.e., Any process that does not share data with any other process is independent.
- process is ***cooperating*** if it can affect or be affected by the other processes executing in the system. Clearly, 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**. 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 such information.

  • **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 cores.

  • **Modularity**. We may want to construct the system in a modular fashion,dividing the system functions into separate processes or threads.

  • **Convenience**. Even an individual user may work on many tasks at the same time. For instance, a user may be editing, listening to music, and compiling in parallel.

  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: **shared memory** and **message passing**. (Refer Module 1 Notes)

Several methods for logically implementing a link and the send()/receive() operations:

1. 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.

- In this scheme, the send() and receive() primitives are defined as:

  - **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.

- A variant of this scheme employs ***asymmetry*** in addressing. Here, only the sender names the recipient; the recipient is not required to name the sender. In this scheme, the send() and receive() primitives are defined as follows:

  - **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.

- 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.
- All references to the old identifier must be found, so that they can be modified to the new identifier.

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

- A process can communicate with another process via a number of different mailboxes, but two processes can communicate only if they have a shared mailbox.

- The send() and receive() primitives are defined as follows:

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

- In this scheme, a communication link 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, a number of different links may exist, with each link corresponding to one mailbox.

- Now suppose that processes *P*1, *P*2, and *P*3 all share mailbox *A*. Process *P*1 sends a message to *A*, while both *P*2 and *P*3 execute a receive() from *A*. Which process will receive the message sent by *P*1?

- 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 *P*2 or *P*3, but not both, will receive the message).

- The system may define an algorithm for selecting which process will receive the message. The system may identify the receiver to the sender.

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

- If the **mailbox is owned by a process** (that is, the mailbox is part of the address space of the process), then we distinguish between the owner (which can only receive messages through this mailbox) and the user (which can only send messages to the mailbox). Since each mailbox has a unique owner, there can be no confusion about which process should receive a message sent to this mailbox. When a **process that owns a mailbox** terminates, the mailbox disappears. Any process that

subsequently sends a message to this mailbox must be notified that the mailbox no longer exists.

- In contrast, 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. The operating system then must provide a mechanism that allows a process to do the following:

  • Create a new mailbox.
  • Send and receive messages through the mailbox.
  • Delete a mailbox.

2. 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.

3. Buffering

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

- 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 (either the message is copied or a

pointer to the message is kept), and the sender can continue execution without waiting. The link's capacity is finite, however. If the link is full, the sender must block until space is available in the queue.

- ✓ **Unbounded capacity**. The queue's length is potentially infinite; thus, any number of messages can wait in it. The sender never blocks.

Communication in Client–Server Systems

1. Sockets

- A **socket** is defined as an endpoint for communication.
- A pair of processes communicating over a network employs a pair of sockets—one for each process.
- A socket is identified by an IP address concatenated with a port number.
- In general, sockets use a client–server architecture.
- 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***
- When a client process initiates a request for a connection, it is assigned a port by its host computer.
- This port has some arbitrary number greater than 1024.
- For example, if a client on host X with IP address 146.86.5.20 wishes to establish a connection with a web server (which is listening on port 80) at address 161.25.19.8, host X may be assigned port 1625.
- The connection will consist of a pair of sockets: (146.86.5.20:1625) on host X and (161.25.19.8:80) on the web server.
- This situation is shown in below Figure .
- The packets traveling between the hosts are delivered to the appropriate process based on the destination port number.
- All connections must be unique. Therefore, if another process also on host X wished to establish another connection with the same web server, it would be assigned a port number greater than 1024 and not equal to 1625.
- This ensures that all connections consist of a unique pair of sockets.

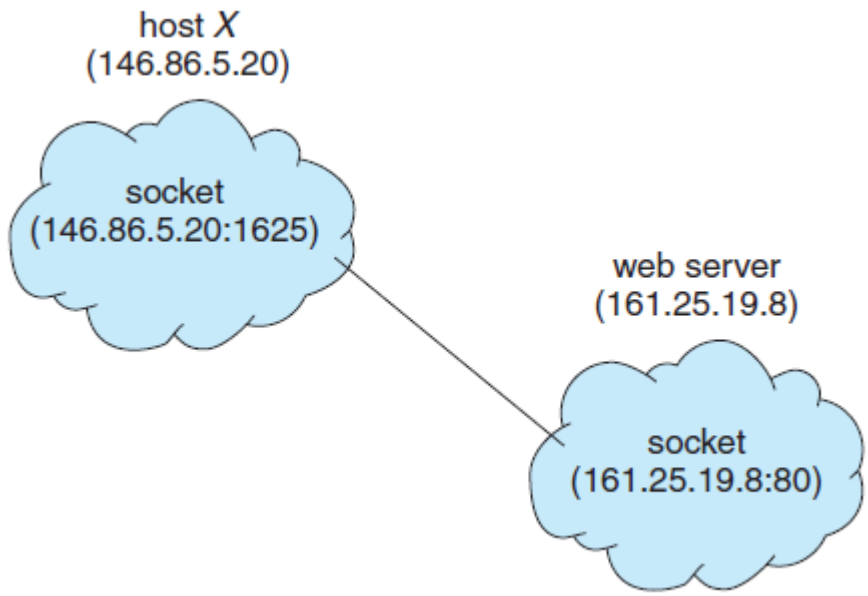

Fig : Communication using Socket.

2. Remote Procedure Calls

Remote Procedure Call (RPC) provides a different paradigm for accessing network services. Instead of accessing remote services by sending and receiving messages, a client invokes services by making a local procedure call. The local procedure hides the details of the network communication.

When making a remote procedure call:

1. The calling environment is suspended, procedure parameters are transferred across the network to the environment where the procedure is to execute, and the procedure is executed there.
2. When the procedure finishes and produces its results, its results are transferred back to the calling environment, where execution resumes as if returning from a regular procedure call.

The main goal of RPC is to hide the existence of the network from a program. As a result, RPC doesn't quite fit into the OSI model:

1. The message-passing nature of network communication is hidden from the user. The user doesn't first open a connection, read and write data, and then close the connection. Indeed, a client often does not even know they are using the network!

2. RPC often omits many of the protocol layers to improve performance. Even a small performance improvement is important because a program may invoke RPCs often. For example, on (diskless) Sun workstations, every file access is made via an RPC.

RPC is especially well suited for client-server (e.g., query-response) interaction in which the flow of control alternates between the caller and callee. Conceptually, the client and server do not both execute at the same time. Instead, the thread of execution jumps from the caller to the callee and then back again.

The following steps take place during an RPC:

1. A client invokes a ***client stub*** procedure, passing parameters in the usual way. The client stub resides within the client's own address space.
2. The client stub ***marshalls*** the parameters into a message. Marshalling includes converting the representation of the parameters into a standard format, and copying each parameter into the message.
3. The client stub passes the message to the transport layer, which sends it to the remote server machine.
4. On the server, the transport layer passes the message to a ***server stub,*** which **demarshalls** the parameters and calls the desired server routine using the regular procedure call mechanism.
5. When the server procedure completes, it returns to the server stub (e.g., via a normal procedure call return), which marshalls the return values into a message. The server stub then hands the message to the transport layer.
6. The transport layer sends the result message back to the client transport layer, which hands the message back to the client stub.
7. The client stub demarshalls the return parameters and execution returns to the caller.

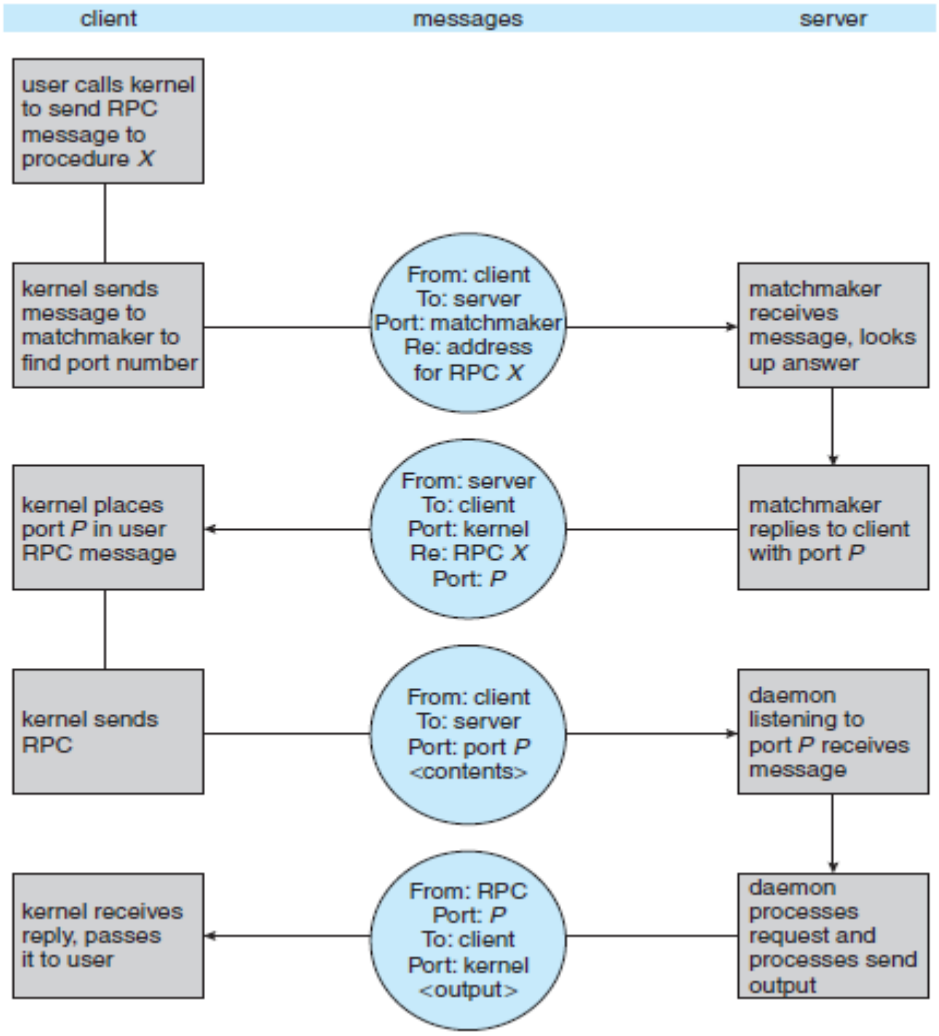

Fig: Execution of RPC