COURSE NAME: OPERATIN SYSTEMS

COURSE CODE: BCS303

SEMESTER: 3

MODULE: 2

NUMBER OF HOURS:08

CONTENTS:

- **Process Management:**
  - Process concept
  - Process scheduling
  - Operations on processes
  - Inter process communication

- **Multi-threaded Programming:**
  - Multithreading models
  - Thread Libraries
  - Threading issues

- **Process Scheduling:**
  - Basic concepts
  - Scheduling Criteria;
  - Scheduling Algorithms;
  - Multiple-processor scheduling;
  - Thread scheduling.

PROCESS MANAGEMENT

Process Concept

- A process is a program under execution.
- Its current activity is indicated by PC (Program Counter) and the contents of the processor's registers.

The Process

Process memory is divided into four sections as shown in the figure below:

- The **stack** is used to store temporary data such as local variables, function parameters, function return values, return address etc.
- The **heap** which is memory that is dynamically allocated during process run time
- The **data** section stores global variables.
- The **text** section comprises the compiled program code.
- Note that, there is a free space between the stack and the heap. When the stack is full, it grows downwards and when the heap is full, it grows upwards.

Process State

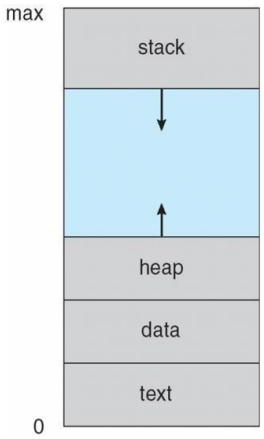

Figure: Process in memory.

Q) Illustrate with a neat sketch, the process states and process control block.

Process State

A Process has 5 states. Each process may be in one of the following states –

1. **New** - The process is in the stage of being created.
2. **Ready** - The process has all the resources it needs to run. It is waiting to be assigned to the processor.
3. **Running** – Instructions are being executed.
4. **Waiting** - The process is waiting for some event to occur. For example, the process may be waiting for keyboard input, disk access request, inter-process messages, a timer to go off, or a child process to finish.
5. **Terminated** - The process has completed its execution.

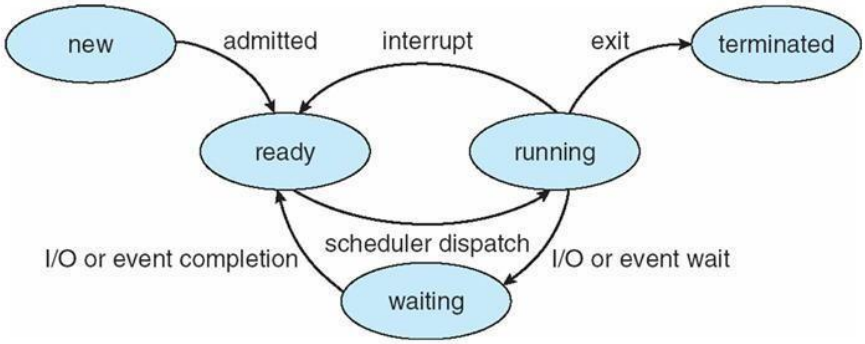

Figure: Diagram of process state

Process Control Block

For each process there is a Process Control Block (PCB), which stores the process-specific information as shown below –

- **Process State** – The state of the process may be new, ready, running, waiting, 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. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward.
- **CPU scheduling information**- This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
- **Memory-management information** – This includes information such as the value of the base and limit registers, the page tables, or the segment tables.
- **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.

The PCB simply serves as the repository for any information that may vary from process to process.

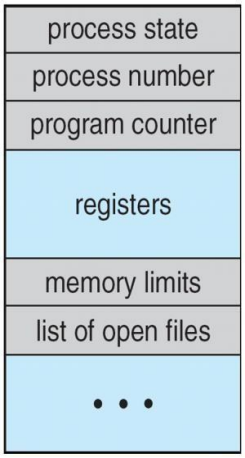

Figure: Process control block (PCB)

CPU Switch from Process to Process

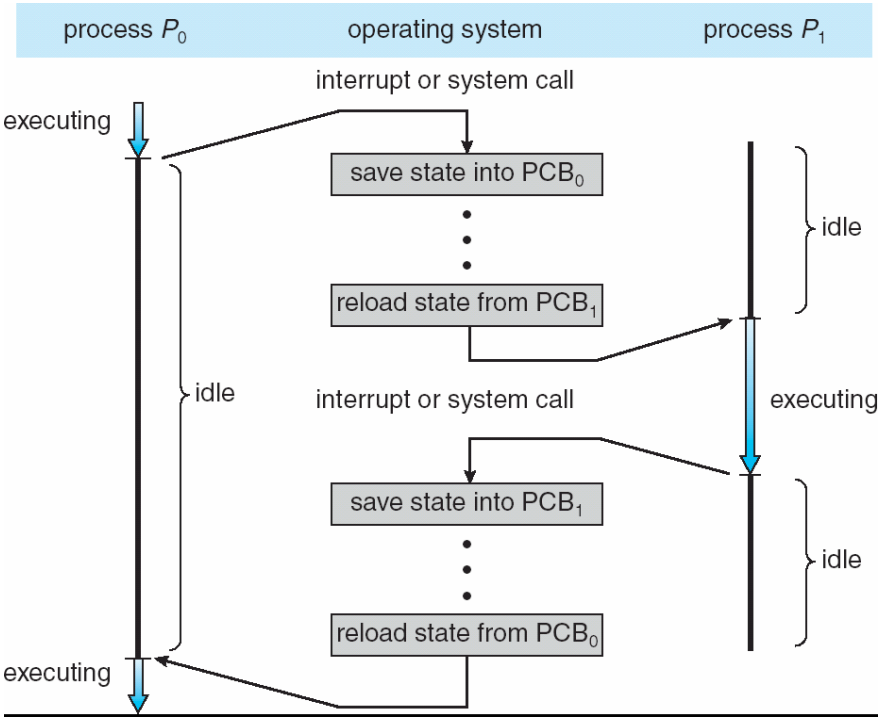

Figure: Diagram showing CPU switch from process to process.

Process Scheduling

Scheduling Queues

- 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. Each PCB includes a pointer field that points to the next PCB in the ready queue.

Ready Queue and Various I/O Device Queues

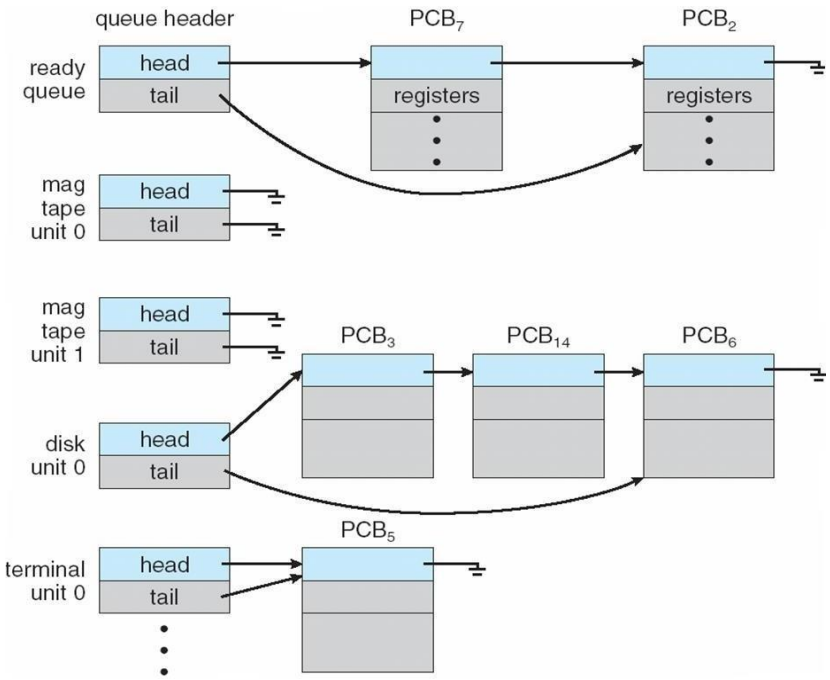

Figure: The ready queue and various I/O device queues

- A common representation of process scheduling is a ***queueing diagram***. Each rectangular box in the diagram 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 in the ready queue until it is selected for execution and is given the CPU. 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 subprocess and wait for its termination.
    - The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.

In the first two cases, the process eventually switches from the waiting state to the ready state, and is then put back in the ready queue. A process continues this cycle until it terminates, at which time it is removed from all queues.

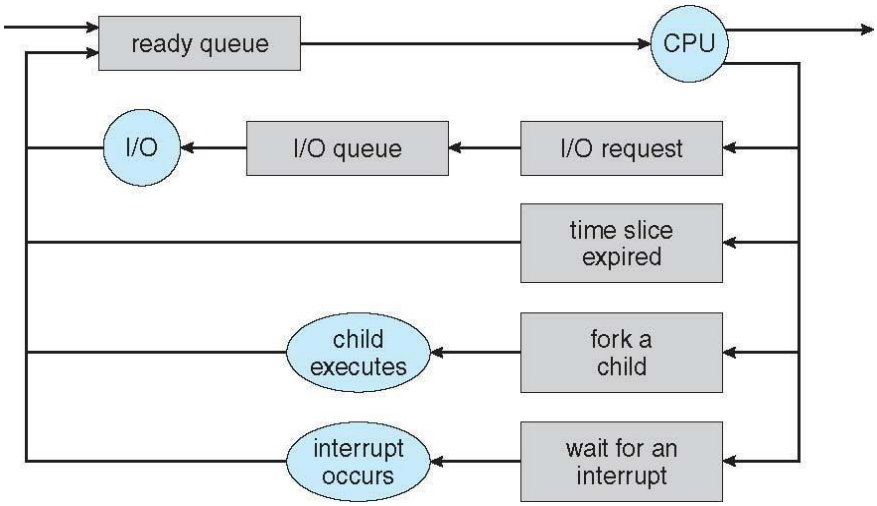

Figure: Queueing-diagram representation of process scheduling.

Schedulers

Schedulers are software which selects an available program to be assigned to CPU.

- A **long-term scheduler or Job scheduler** – selects jobs from the job pool (of secondary memory, disk) and loads them into the memory.
  If more processes are submitted, than that can be executed immediately, such processes will be in secondary memory. It runs infrequently, and can take time to select the next process.

- **The short-term scheduler, or CPU Scheduler** – selects job from memory and assigns the CPU to it. It must select the new process for CPU frequently.
- **The medium-term scheduler** - selects the process in ready queue and reintroduced into the memory.

Processes can be described as either:

- I/O-bound process – spends more time doing I/O than computations,
- CPU-bound process – spends more time doing computations and few I/O operations.

An efficient scheduling system will select a good mix of **CPU-bound** processes and **I/O bound** processes.

- If the scheduler selects **more I/O bound process**, then I/O queue will be full and ready queue will be empty.
- If the scheduler selects **more CPU bound process**, then ready queue will be full and I/O queue will be empty.

Time sharing systems employ a **medium-term scheduler**. It swaps out the process from ready queue and swap in the process to ready queue. When system loads get high, this scheduler will swap one or more processes out of the ready queue for a few seconds, in order to allow smaller faster jobs to finish up quickly and clear the system.

Advantages of medium-term scheduler –

- To remove process from memory and thus reduce the degree of multiprogramming (number of processes in memory).
- To make a proper mix of processes (CPU bound and I/O bound)

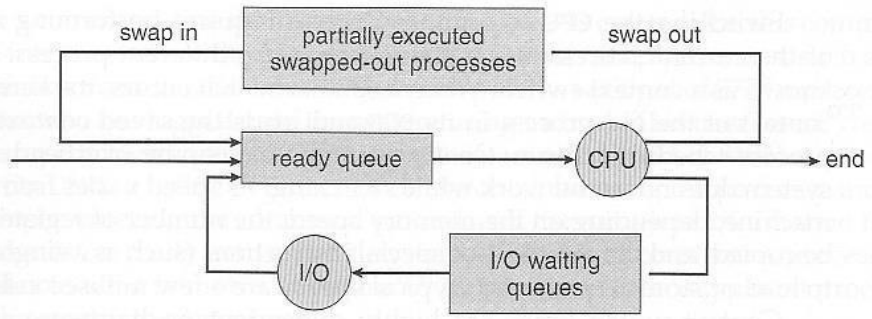

Figure 3.8 Addition of medium-term scheduling to the queueing diagram.

Context switching

- The task of switching a CPU from one process to another process is called context switching. Context-switch times are highly dependent on hardware support (Number of CPU registers).
- Whenever an interrupt occurs (hardware or software interrupt), the state of the currently running process is saved into the PCB and the state of another process is restored from the PCB to the CPU.
- Context switch time is an overhead, as the system does not do useful work while switching.

Operations on Processes

Q) Demonstrate the operations of process creation and process termination in UNIX

Process Creation

- 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. Every process has a unique process ID.
- On typical Solaris systems, the process at the top of the tree is the '**sched'** process with PID of 0. The **'sched'** process creates several children processes – **init**, **pageout** and **fsflush**. Pageout and fsflush are responsible for managing memory and file systems. The init process with a PID of 1, serves as a parent process for all user processes.

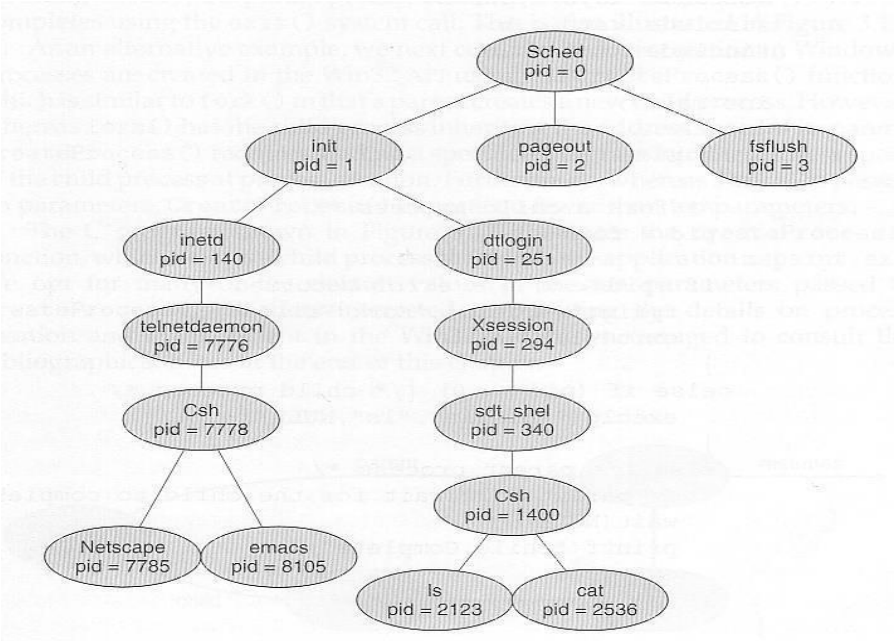

Figure 3.9 A tree of processes on a typical Solaris system.

A process will need certain resources (CPU time, memory, files, I/O devices) to accomplish its task. When a process creates a subprocess, the subprocess may be able to obtain its resources in two ways:

- directly from the operating system
- Subprocess may take the resources of the parent process.

  The resource can be taken from parent in two ways –
  - The parent may have to partition its resources among its children
  - Share the resources among several children.

There are two options for the parent process after creating the child:

- Wait for the child process to terminate and then continue execution. The parent makes a wait() system call.
- Run concurrently with the child, continuing to execute without waiting.

Two possibilities for the address space of the child relative to the parent:

- The child may be an exact duplicate of the parent, sharing the same program and data segments in memory. Each will have their own PCB, including program counter, registers, and PID. This is the behaviour of the **fork** system call in UNIX.
- The child process may have a new program loaded into its address space, with all new code and data segments. This is the behaviour of the **spawn** system calls in Windows.

```
#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");
       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);
    }
}
```

Figure 3.10 C program forking a separate process.

In UNIX OS, a child process can be created by **fork()** system call. The **fork** system call, if successful, returns the PID of the child process to its parents and returns a zero to the child process. If failure, it returns -1 to the parent. Process IDs of current process or its direct parent can be accessed using the getpid( ) and getppid( ) system calls respectively.

The parent waits for the child process to complete with the wait() system call. When the child process completes, the parent process resumes and completes its execution.

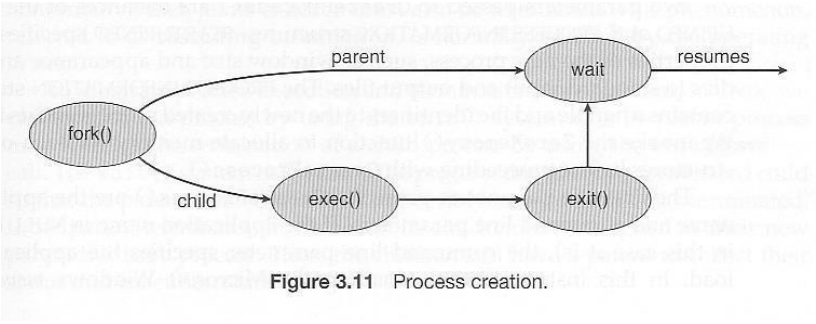

Figure 3.11 Process creation.

In windows the child process is created using the function **createprocess( )**. The createprocess( ) returns 1, if the child is created and returns 0, if the child is not created.

Process Termination

- A process terminates when it finishes executing its last statement and asks the operating system to delete it, by using the **exit** () system call. All of the resources assigned to the process like memory, open files, and I/O buffers, are deallocated by the operating system.
- A process can cause the termination of another process by using appropriate system call. The parent process can terminate its child processes by knowing of the PID of the child.
- A parent may terminate the execution of children for a variety of reasons, such as:
    - The child has exceeded its usage of the resources, it has been allocated.
    - The task assigned to the child is no longer required.
    - The parent is exiting, and the operating system terminates all the children. This is called **cascading termination**.

Interprocess Communication

Q) What is interprocess communication? Explain types of IPC.

Interprocess Communication- Processes executing may be either co-operative or independent processes.

- ***Independent Processes*** – processes that cannot affect other processes or be affected by other processes executing in the system.
- ***Cooperating Processes*** – processes that can affect other processes or be affected by other processes executing in the system.

Co-operation among processes are allowed for following reasons –

- **Information Sharing** - There may be several processes which need to access the same file. So the information must be accessible at the same time to all users.
- **Computation speedup** - Often a solution to a problem can be solved faster if the problem can be broken down into sub-tasks, which are solved simultaneously (particularly when multiple

processors are involved.)

- **Modularity** - A system can be divided into cooperating modules and executed by sending information among one another.
- **Convenience** - Even a single user can work on multiple tasks by information sharing.

Cooperating processes require some type of inter-process communication. This is allowed by two models:

1. Shared Memory systems
2. Message passing systems.

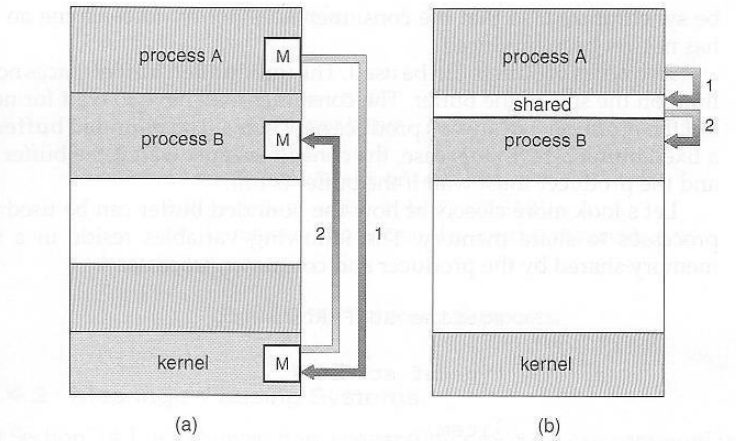

Figure 3.13 Communications models. (a) Message passing. (b) Shared memory.

Sl No	Shared Memory	Message passing
1.	A region of memory is shared by communicating processes, into which the information is written and read	Message exchange is done among the processes by using objects.
2.	Useful for sending large block of data	Useful for sending small data.
3.	System call is used only to create shared memory	System call is used during every read and write operation.
4.	Message is sent faster, as there are no system calls	Message is communicated slowly.

- **Shared Memory** is faster once it is set up, because no system calls are required and access occurs at normal memory speeds. Shared memory is generally preferable when large amounts of information must be shared quickly on the same computer.
- **Message Passing** requires system calls for every message transfer, and is therefore slower, but it is simpler to set up and works well across multiple computers. Message passing is generally preferable when the amount and/or frequency of data transfers is small.

Shared-Memory Systems

- A region of shared-memory is created within the address space of a process, which needs to communicate. Other process that needs to communicate uses this shared memory.
- The form of data and position of creating shared memory area is decided by the process. Generally, a few messages must be passed back and forth between the cooperating processes first in order to set up and coordinate the shared memory access.
- The process should take care that the two processes will not write the data to the shared memory at the same time.

Producer-Consumer Example Using Shared Memory

- This is a classic example, in which one process is producing data and another process is consuming the data.
- The data is passed via an intermediary buffer (shared memory). The producer puts the data to the buffer and the consumer takes out the data from the buffer. 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. In this situation, the consumer must wait until an item is produced.
- There are two types of buffers into which information can be put –
    - Unbounded buffer
    - Bounded buffer

- **With Unbounded buffer**, there is no limit on the size of the buffer, and so on the data produced by producer. But the consumer may have to wait for new items.

- **With bounded-buffer** – As the buffer size is fixed. The producer has to wait if the buffer is full and the consumer has to wait if the buffer is empty.

This example uses shared memory as a circular queue. The **in** and **out** are two pointers to the array. Note in the code below that only the producer changes "in", and only the consumer changes "out".

- First the following data is set up in the shared memory area:

```
#define BUFFER_SIZE 10

typedef struct {
    . . .
}item;

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

- The producer process –
  Note that the buffer is full when [(in+1) % BUFFER_SIZE == out]

```
item nextProduced;

while (true) {
     /* produce an item in nextProduced */
     while (((in + 1) % BUFFER-SIZE) == out)
        ; /* do nothing */
     buffer[in] = nextProduced;
     in = (in + 1) % BUFFER_SIZE;
}
```

Figure The producer process.

- The consumer process –
  Note that the buffer is empty when [in == out]

```
item nextConsumed;

while (true) {
     while (in == out)
        ; //do nothing

     nextConsumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;
     /* consume the item in nextConsumed */
}
```

Figure The consumer process.

Message-Passing Systems

A mechanism to allow process communication without sharing address space. It is used in distributed systems.

- Message passing systems uses system calls for "send message" and "receive message".
- A communication link must be established between the cooperating processes before messages can be sent.
- There are three methods of creating the link between the sender and the receiver-
  - Direct or indirect communication (naming)
  - Synchronous or asynchronous communication (Synchronization)
  - Automatic or explicit buffering.

1. Naming

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

a) Direct communication the sender and receiver must explicitly know each other's name. The syntax for send() and receive() functions are as follows-

- **send** (*P*, *message*) – send a message to process P
- **receive**(*Q*, *message*) – receive a message from process Q

Properties of communication link:

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

Types of addressing in direct communication –

- Symmetric addressing – the above-described communication is symmetric communication. Here both the sender and the receiver processes have to name each other to communicate.
- Asymmetric addressing – Here only the sender's name is mentioned, but the receiving data can be from any system.

  send (P, message) --- Send a message to process ***P***

  receive (id, message). Receive a message from any process

Disadvantages of direct communication – any changes in the identifier of a process, may have to change the identifier in the whole system (sender and receiver), where the messages are sent and received.

b) Indirect communication uses shared mailboxes, or ports.

A mailbox or port is used to send and receive messages. Mailbox is an object into which messages can be sent and received. It has a unique ID. Using this identifier messages are sent and received.

Two processes can communicate only if they have a shared mailbox. The send and receive functions are –

- **send** (*A*, *message*) – send a message to mailbox A
- **receive** (*A*, *message*) – receive a message from mailbox A

Properties of communication link:

- A link is established between a pair of processes only if they have a shared mailbox

- A link may be associated with more than two processes
- Between each pair of communicating processes, there may be any number of links, each link is associated with one mailbox.
- A mail box can be owned by the operating system. It must take steps to –
  - create a new mailbox
  - send and receive messages from mailbox
  - delete mailboxes.

2. Synchronization

The send and receive messages can be implemented as either **blocking** or **non-blocking**.

- **Blocking (synchronous) send -** sending process is blocked (waits) until the message is received by receiving process or the mailbox.
- **Non-blocking (asynchronous) send** - sends the message and continues (does not wait)

- **Blocking (synchronous) receive -** The receiving process is blocked until a message is available
- **Non-blocking (asynchronous) receive** - receives the message without block. The received message may be a valid message or null.

3. Buffering

When messages are passed, a temporary queue is created. Such queue can be of three capacities:

- **Zero capacity** – The buffer size is zero (buffer does not exist). Messages are not stored in the queue. The senders must block until receivers accept the messages.
- **Bounded capacity**- The queue is of fixed size(n). Senders must block if the queue is full. After sending 'n' bytes the sender is blocked.
- Unbounded capacity **- The queue is of infinite capacity. The sender never block**