# Concurrent Process. UNIT-2

## Process Concept :-

→ A process is an executing program, including the current values of the program counter, registers and variables

→ Difference between a process and a program is a group of instructions whereas the process is the activity.

→ Process can be described :

- I/O Bound Process - spends more time doing I/O then computation.
- CPU Bound Process :- spends more time doing computation.

## Cooperating Processes

→ Processes executing concurrently in the O.S. 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. Any process that does not share data with any other process is independent.

→ A 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.

→ Advantages of process cooperation :-

(i) Information sharing

(ii) Computation speedup

(iii) Modularity (dividing the system functions into separate processes or threads.

(iv) Convenience (an individual user may work on many tasks at the same time.

## Process Synchronization :

→ Concurrent access to shared data may result in data inconsistency.

→ Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.

→ Suppose that we wanted to provide a solution to the consumer-producer problem that fills all the buffers. We can do so by having an integer count that keeps track of the number of full buffers. Initially, count is set to 0. It is incremented by the producer after it produces a new buffer & is decremented by the consumer after it consumes the buffer.

### Critical Section Problem :

→ Consider a system consisting n processes { P0, P1 ... Pn }

→ Each process has a segment of code, called a critical section, in which the process may be changing common variables, updating a table, writing a file & so on.

→ The important feature of the system is that, when one process is executing in its C.S., no other process is to be allowed to execute in its C.S.

→ That is, no two processes are executing in their C.S.s at the same time (mutually exclusive)

→ The section of code implementing this request is the entry section. The C.S. may be followed by an exit section. The remaining code is the remainder section.

```
do {
    Entry Section
        Critical Section
    exit Section
        remainder section
} while(true);
```

## Solution to C.S. Problem

A. Mutual Exclusion :- If process $P_i$ is executing in its critical section, then no other processes can be executed in their critical sections.

B. Progress :- If no process is executing in its critical section and there exists some processes that wish to enter their C.S., then the selection of the processes that will enter the C.S. next cannot be postponed indefinitely.

C. Bounded Waiting :- A bound must exists on the number of times that other processes are allowed to enter their C.S, after a process has madey made a request to enter its C.S. and before that request is granted.

→ Assume that each process executes at a non zero speed.

→ No assumption concerning relative speed of the N processes.

Finally, we need four conditions to hold to have a good solution:

1. No two processes may be simultaneously inside their C.S.
2. No assumptions may be made about speeds or the number of processors.
3. No processes running outside its C.S. may block other processes.
4. No process should have to wait forever to enter its C.S.

Peterson's solution :-

```
do {
    flag[i] = TRUE
    turn = j;
    while (flag[j] && turn == j);

    critical section

    flag[i] = FALSE;

    remainder section
} while (TRUE);
```

## Dekker's Solution :-

```
Do {
flag[0] := true
  while flag[1] = true {
   if turn ≠ 0 {
    flag[0] := false
   while turn ≠ 0 {
   }
   flag[0] := true.
   }
 }
// critical section
....
turn := 1
flag[0] := false
// remainder section
} while (TRUE);
```

## Peterson's Solution

```
do {
  flag[i] = TRUE
   turn = j;
  while (flag[j] && turn == j);

  critical section

  flag[i] = FALSE

   remainder section

  } while (TRUE);
```

## Test & Set Operation :

```
do {
   waiting[i] = TRUE;
      key = TRUE;
   while (waiting[i] && key)
      key = TestandSet(&lock);
   waiting[i] = FALSE;
   // critical section
   j = (i+1) % n;
   while ((j != i) && !waiting[j])
      j = (j+1) % n
   if (j == i)
      lock = FALSE;
   else
      waiting[j] = FALSE;
   // remainder section
} while (TRUE);
```

## Semaphores :-

- Semaphores provide solution to critical section problem.
- Definition :- Semaphore S is an integer variable, apart from initialization is accessed only through two automatic operation wait() & signal().

| wait(S)<br>{<br>While(S<0)<br>;<br>S--; // no operation<br>} | Signal(S)<br>{<br>S++;<br>} |
|---|---|

## Mutex Implementation with semaphore with no busy time :-

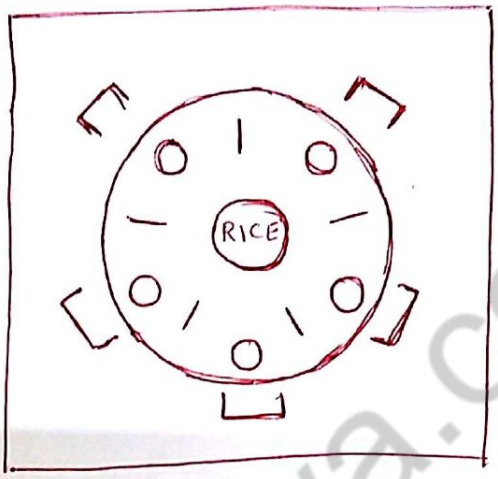
Must guarantee that not two processes can execute wait() & signal() on the same semaphore at the same time.

```
Do {
 wait(mutex)
   critical section
 signal(mutex)
    remainder section
} while(1);
```

| wait(S) {<br>value --;<br>if(value<0) {<br>add this process to waiting queue<br>block(); }<br>} | Signal(S) {<br>value++;<br>if(value <= 0) {<br>remove a process P from the waiting queue<br>wakeup(P);<br>} } |
|---|---|

## Classical Problem in Concurrency :-

### Dining Philosopher Problem :-




Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by 5 chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five single chopsticks.

Shared Data :-

- Bowl of rice (data set)
- Semaphore chopstick [5] initialized to 1.

The structure of Philosopher (i) :-

```
while (true) {
    wait (chopstick [i]);
    wait (chopstick [(i+1) % 5]);

    // eat
    signal (chopstick [i]);
    signal (chopstick [(i+1) % 5]);

    // think
}.
```

## Sleeping Barber's Problem

A Barber shop has a single room with one barber chair and n customer chairs. Customers (i.e. independent processes) enter the waiting room one at a time if empty chairs are available, otherwise they go elsewhere. Each time the barber (i.e. independent process) finishes a haircut the customer leaves to go elsewhere, & among the waiting customers, the one who has been waiting the longest (i.e. a FIFO waiting queue) move to the barber's chair to get a hair cut.

If the barber discovers the waiting room is empty, (s)he falls asleep in the waiting room. An arriving customer finding the barber asleep wakes the barber (i.e. a rendezvous takes place) and has a haircut; otherwise the arriving customer waits.

Using semaphores & shared variables only (but not critical region statements or monitors), give two algorithms, one for the barber, and another for an arbitrary customer, that implement the sleeping barber problem. Explain your algorithms, and explicitly specify any assumptions you make about the model.

# Interprocess Communication

## Cooperating Processes :-

⇒ An independent process is one that cannot affect or be affected by the execution of other processes.

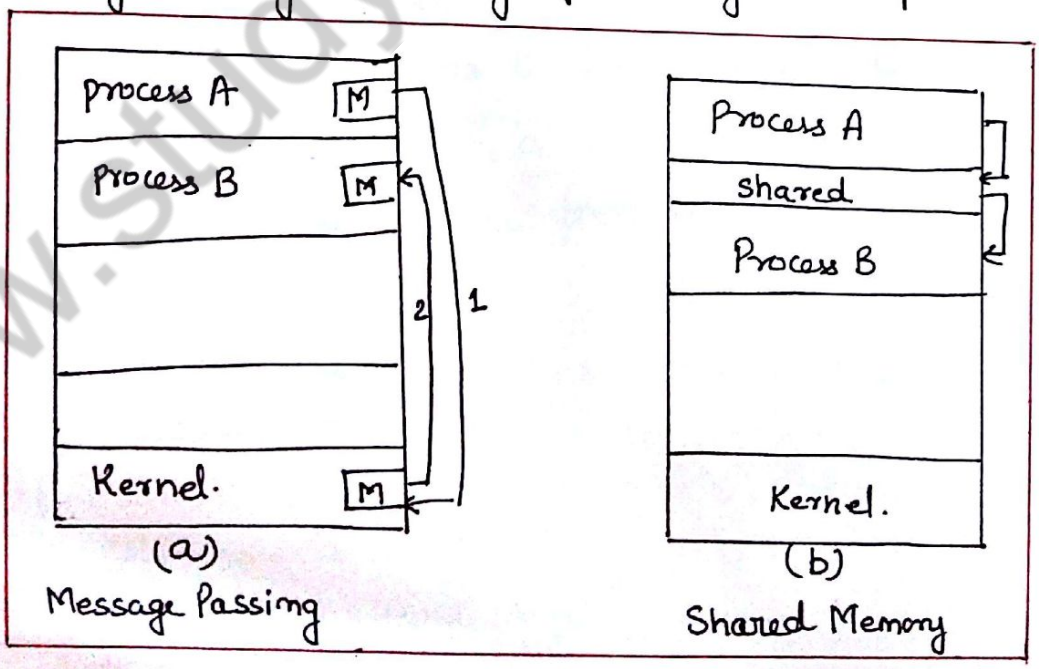
⇒ A cooperating process can affect or be affected by the execution of another process in the system.

### Advantages of cooperating processes

→ Information sharing (of the same piece of data)
→ Computation speed up (break a task into smaller subtasks)
→ Modularity (dividing up the system functions)
→ Convenience (to do multiple tasks simultaneously)

### Two fundamental models of interprocess communication

→ Shared Memory (a region of memory is shared)
→ Message Passing (exchange of messages b/w processes).



## Shared Memory Systems

→ Shared Memory requires communicating processes to establish a region of shared memory.

→ Information is exchanged by reading & writing data in the shared memory.

→ A common paradigm for cooperating processes is the producer-consumer problem.

→ A producer process produces inf$^{n}$ 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.

## Message Passing Systems :-

→ Mechanism to allow process to communicate & to synchronize their actions.

→ No address space needs to be shared; this is particularly useful in a distributed processing environment (eg. a chat program).

→ Message passing facility provides two operations :

- send (message) - msg size can be fixed or variable
- receive (message)

→ If P and Q wish to communicate, they need do :

- establish a communicat$^{n}$ link b/w them.
- exchange msg via send/receive.

## Logical Implementation of comm^n link.

→ Direct or indirect communicat^n
→ synchronous or asynchronous communication
→ Automatic or explicit buffering

### Direct Communication

→ Process must name each other explicitly:
- send (P, message) – send a msg to process P
- receive (Q, message) – receive a msg from process P

→ Properties of communication link
- Link are established automatically between every pair of processes that want to communicate.
- 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.

→ Disadvantages:-
- Limited modularity of the resulting process definitions
- Hard-coding of identifiers are less desirable than indirection techniques.

## 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
- send (A, message) - send message to mailbox A
- receive (A, message) - receive msg from mailbox A.

→ Properties of comm^n link
- Link is established between a pair of processes only if both have shared mailbox.
- A link may be associated with more than two processes.
- Between each pair of processes, there may be many different links, with each link corresponding to one mailbox.

→ For shared mailbox, messages are received based on the following methods:
- Allow a link to be established with atmost two processes.
- Allow at most one process at a time to execute a receive() operation.
- Allow the system to select arbitrary which process will receive the message. (eg round robin tech).

## Synchronization :-

→ Message passing may be either blocking or non-blocking.

→ Blocking is considered synchronous.

↪ • Blocking send has the sender block until the message is received.

° Blocking receive has the receiver block until a message is available.

→ Non blocking is considered asynchronous.

° Non-blocking send has the sender send the message & continue.

° Non-blocking receives has the receiver receive a valid message or null.

→ When the send() and receive() are blocking, we have a rendezvous btw the sender & receiver.

## Buffering

whether communication is direct or indirect, msg exchanged by communicating processes resides in a temporary queue. These queues can be implemented in 3 ways.

1 Zero capacity : the queue has a maximum length of zero. Sender must block until the recipient receives the msg.

2. Bounded capacity : the queue has the finite length of n. Sender must wait if queue is full.

3. Unbounded capacity : the queue length is unlimited Sender never blocks.

## Process Creation:-

Parent process create children processes, which in turn create other processes, forming a tree of processes.

### Resource sharing:-

1. Parent and children share all resources.
2. Children share subset of parent's resources.
3. Parent and child share no resources.

### Execution:-

1. Parent and children execute concurrently.
2. Parent waits until children terminate.

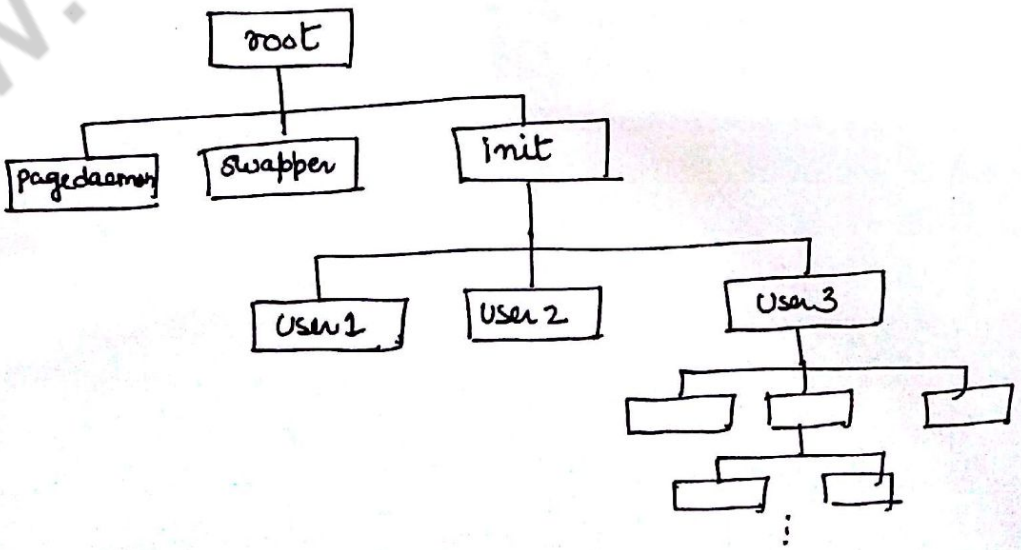
### Address Space

1. Child duplicate of parent.
2. Child has a program loaded to it.

### UNIX examples:-

1. "fork" system call creates new processes.
2. "exec" system call used after a fork to replace the process' memory space with a new program.

### Process tree on a Unix System:-



## Process Termination

Process executes last statement and asks the operating system to decide it (exit).

1. Output data from child to parent (via wait).
2. Process resources are de allocated by operating system. Parent may terminate execution of children processes (abort).
3. Child has exceeded allocated resources.
4. Task assigned to child is no longer required.
5. Parent is existing.
6. Operating system does not allow child to continue if its parent terminates.