UNIT 2

Process Management

A **process** is a program in execution. It is the unit of work in modern time-sharing system. When the program is loaded into the memory it becomes process and it is divided into four sections: stack, heap, text and data. Although its main concern is to execute user program, it also needs to take care of various system tasks. A system therefore consists of a collection of processes like operating system process executing system code and user processes executing user code. All the process can execute concurrently with the CPU multiplexed among them.

A process is more than a program code which is sometimes known as a **text section**. Text section also includes the current activity represented by the value of **program counter** and the content of processor's register. Process generally also includes the **process stack** which contains temporary data like function, parameter, return address and local variables. It also contains data section which contains global variable and includes heap also which is a memory that is dynamically allocated during process run time. The structure of process is shown below:

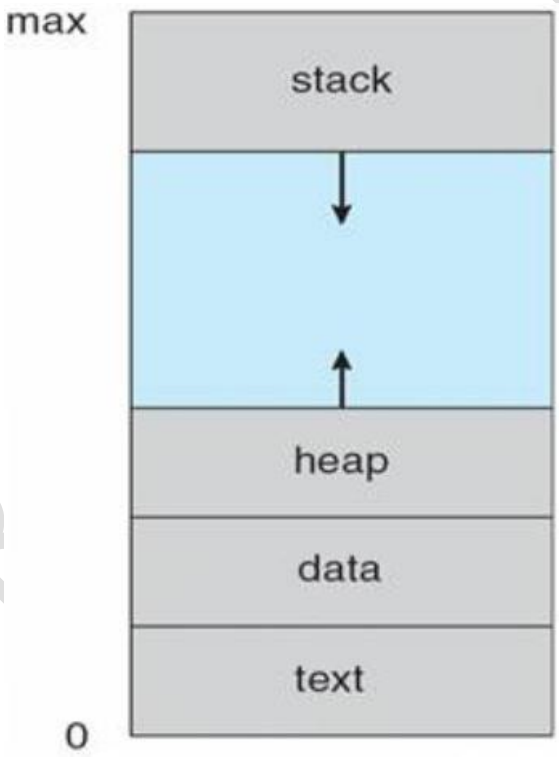


Figure: Process in memory

Difference between program and process:

Program by itself is not a process. Program is a passive entity such as file containing a set of instruction stored on the disk whereas process is an active entity with a program counter specifying the next instruction to execute and a set of associated resources. A program becomes process when an executable file is loaded into memory.

Process State:

As a process executes it changes state. The state of the process is defined in part by the current activity of that process. A process may be in one of the following states:

- **New**: this is the initial state in which the process is being created or started.
- **Running**: it is the state in which instruction are being executed.
- **Waiting**: the process is waiting for some event to occur such as I/O completion or reception of signal. Process moves to this state if it need to wait for some resources or waiting for a file to become available.
- **Ready**: here the process will be waiting to be assigned to a processor so that they can run. Process may come into this state after the start state or while running.
- **Terminated**: process moves to terminated state once it complete its execution or is terminated by the operating system. Here it waits to be remove from main memory.

It is important to realize that only one process can be running on any processor at any instant, other may be ready and waiting.

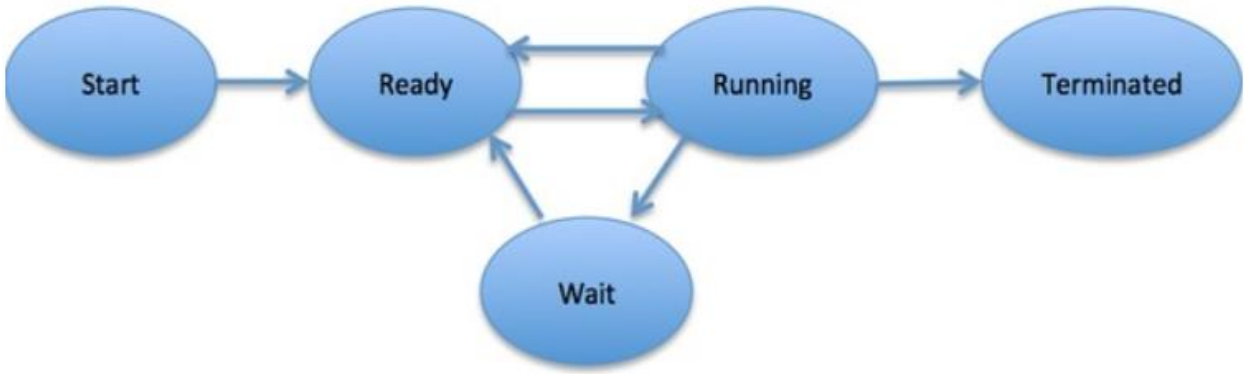


Figure: Process State

Process Control Block:

Each process is represented in the operating system by a process control block also known as task control block. It is the data structure maintained by the operating system for every process. PCB simply serves as the repository for an information that may vary from process to process. It contains many pieces of information associated with a specific process including:

- **Process state:** it refers to current state of the process i.e. the state may be new, ready, running and waiting, halted and so on.
- **Program counter:** the counter indicates the address of the next instruction to be executed for this process.
- **CPU register:** it includes various CPU registers where process needs to be stored for execution for running state. The register may vary in number and type depending on a computer architecture. They includes accumulator, index register, stack pointer, any condition code information. The state information must be saved when 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 information may include items like value of the base and limit register, page table, segment etc. depending upon the memory system used by the operating system.
- **Accounting information:** this information includes the amount of CPU time and real time used, time limits, account number, job or process number and so on.
- **I/O status information:** this information includes the list of I/O devices allocated to the process, a list of open file and so on.

The architecture of PCB completely depends on operating system used and may contains different information in different operating system. The structure of PCB is shown below:

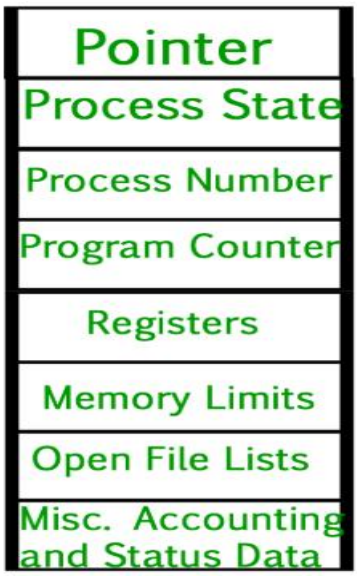


Figure: Process Control Block (PCB)

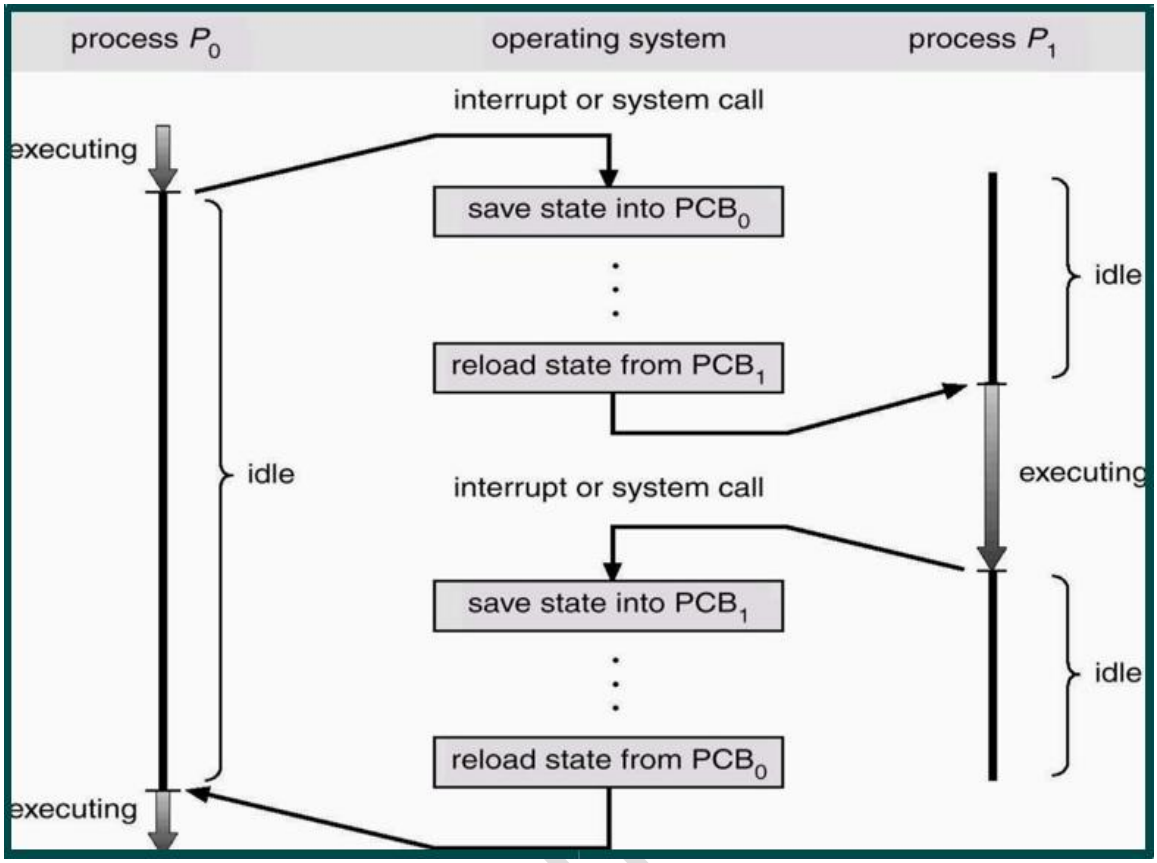


Figure: diagram showing CPU switch from process to process

Process scheduling:

The objective of multiprogramming is to have some process running at all times such that CPU utilization is maximized. Similarly, the objective of time sharing system is to switch CPU among the process so frequently that user can interact with each program while it is running. To meet these objective, the process scheduler selects an available process from a set of processes for program execution in CPU.

As process enters the system they are put into the job queues which consist of all the process in the system. The process that are residing in main memory and are ready to execute are kept on a list called ready queue. This queue is generally stored in linked list and a ready queue headers contains pointer to the first and last PCB in the list. Each PCB contains a pointer field that points to the next PCB in the ready queue.

When a process is allocated the CPU, it executes for a while then eventually quits or interrupted or wait for the occurrence of particular event like completion of I/O request. Thus the process may have to wait for disk. The list of process waiting for a particular I/O device is called device queue.

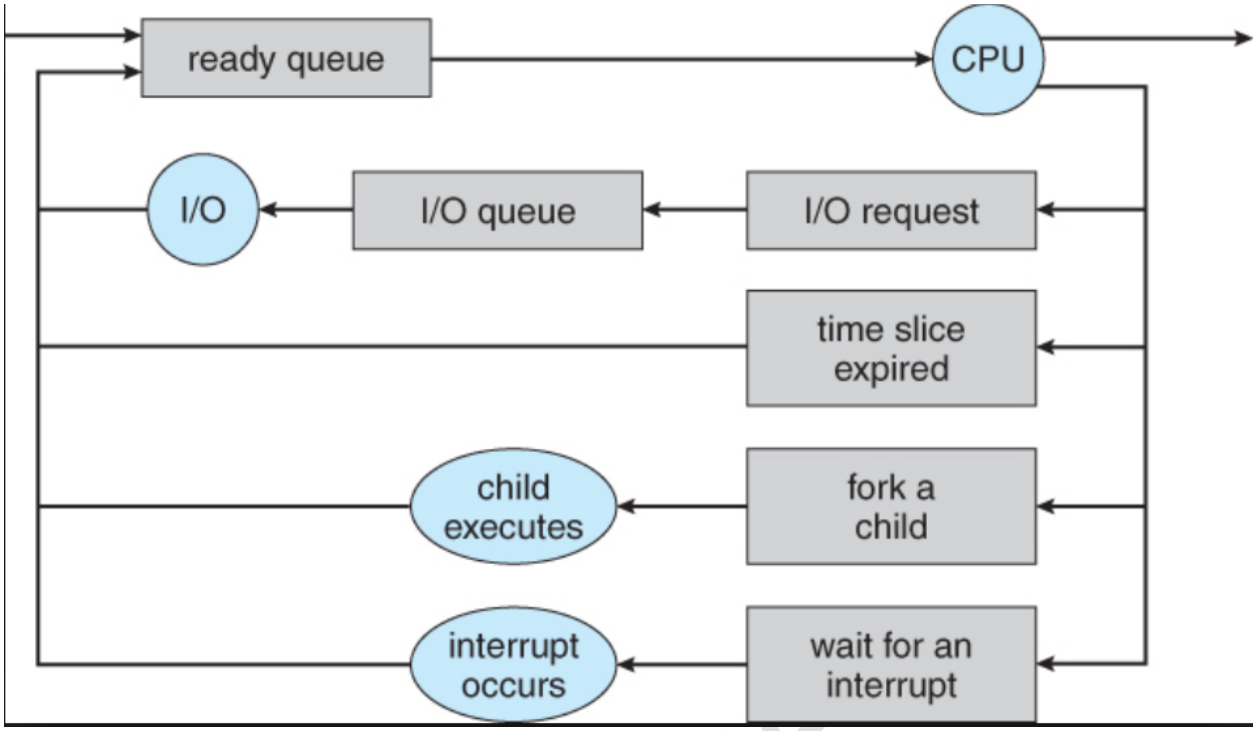


Figure: queuing diagram representing of process scheduling

The above figure shows the representation of process scheduling. Each rectangular box represents a queue. Here two types of queues are present: the ready queue and a set of device queue. The circle represents the resources that serve the queues and arrow represents flow of process in the system.

A new process is initially put into ready queue and waits there until it is selected for execution or dispatched. Once the process is allocated to CPU and is executing following events could occurs:

- A process could issues an I/O request and then be placed on I/O queue.
- A process could create a new child process and for child termination
- The process could be removed forcibly from CPU due to interrupt and be put back in the ready queue.

In the first two case process eventually switches from waiting state to the ready state and is then put back into ready queue and continues until it is terminated. The address space of the current process must be preserved as the space of the next task is prepared for use.

Process Context Switch:

Switching the CPU to another process requires performing a state save of the current process and a state restore of a different process. This task is known as context switch. Because of this CPU can resumes the process execution from the same point at which it has been stopped. When a context switch occur the kernel saves the context of old process in its PCB and load the saved context of the new process scheduled to run. Context-switching is pure overhead

because system does no useful work while switching. Switching speed varies from machine to machine depending upon memory speed, number of register and the existence of special instruction. Context switch times are highly dependent on hardware

Process Address Space:

An address space is the range of valid address in the memory that are available for the program or process i.e. the memory, process and program can access. It means a space that is allocated in memory for a process. One process cannot access another address space and same pointer address in different processes point to different memory. The process has at least three segment of usable address.

- A text segment that contains the executable image of the program
- A data segment contains the heap of dynamically allocated data
- A stack segment contains the function call stack.

Operation on Processes:

The process in most of the system can execute concurrently and they must be created and deleted dynamically. The system must provide the mechanism for process creation and termination. The mechanism involved in creating and terminating process in UNIX system is shown below:

1. **Process creation:**
   A process may create several new process during the course of execution. The creating process is called parent process and the new process are called the children of that process. Each of these process may in turn create other processes forming a tree of process known as **hierarchies** of process. Processes are identified according to the unique process identifier or pid, which is typically an integer value. The pid provides a unique value for each process in the system and it can be used as an index to access various attributes of a process within the kernel.
   The operating system can run many process at the same time but initially it directly starts the one process called init (initial) process. The init process which always has a pid 1 serve as the root parent process for all user process.
   When a process creates a new process, the two possibilities for execution exist:
   - The parent continue to execute concurrently with its children
   - The parent waits until some or all of its children have terminated.

   There are also two address space possibility for the new process:

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

   When a process created a child process, that child process will need certain resources (CPU time, memory etc.) to accomplish the task. Such resources can be obtain directly from the operating system or it may be constrained to the subset of resource of the parent process.

Following figure illustrate the typical process tree for Linux operating system, showing the name of the process and pid. Once the system has started, the init process can create various other process such as web or print server, ssh server etc.

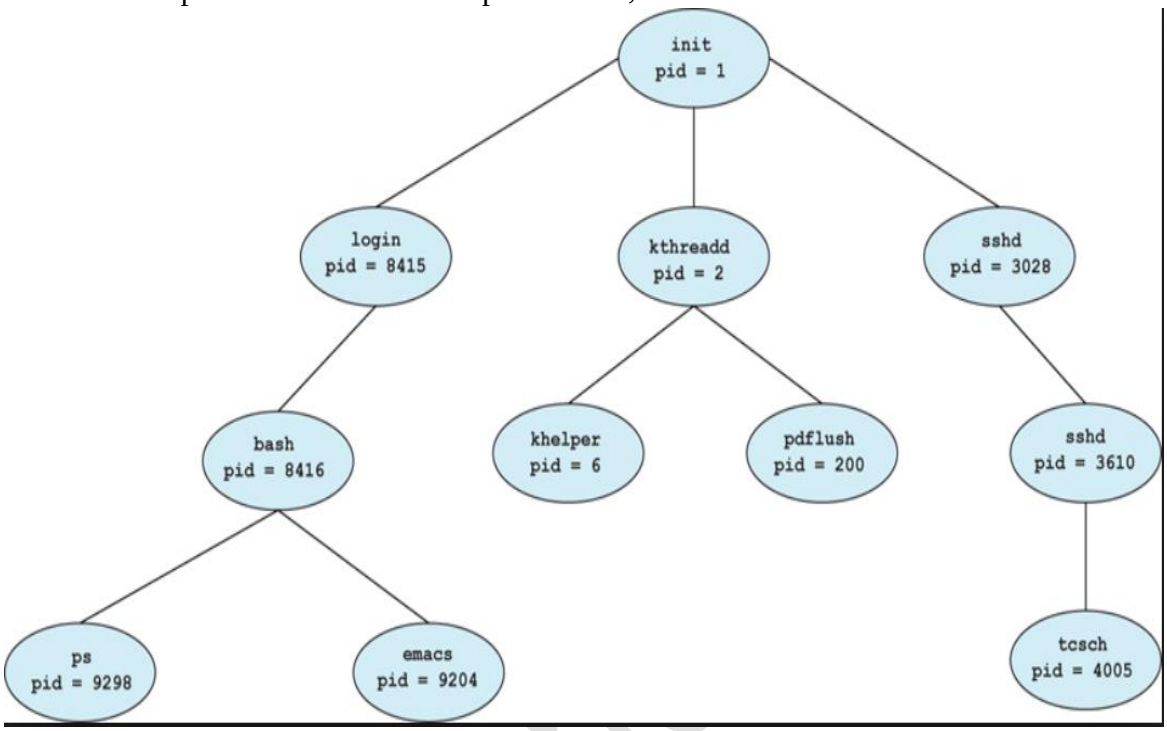


Figure: A tree of processes on a typical Linux system

In above figure, there are three children of init: kthreadd, sshd and login. The kthreaded process is responsible for creating additional process that perform the task in behalf of kernel. The sshd process is responsible for managing the client that connected to system by using ssh (secure shell).

The login process is responsible for the managing the client that directly logs on to the system. Here, a clients have logged in and is using a bash shell which has been assigned pid 8416. Using the bash command line user has created the process ps and emacs editor.

2. **Process Termination:**

A process terminates when it finishes executing its final statement and ask the operating system to delete it by using exit() system call. At this point process may return the status value to its parent process via the wait() system call. All the resource that process consumed are deallocated by the operating system.

A parent process may terminated the execution of one of its children for a variety of reasons such as:

- The child has exceeded its usage of some of the resources that it has been allocated.
- The task assigned to the children is no longer required.
- The parent is exiting and the OS does not allow a child to continue if its parent terminate.

In some system, if parent terminate then child process are not allowed to exist i.e. child process also have to terminate. This phenomenon is referred as cascading termination which is normally initiated by operating system.

In Linux and UNIX system, process are terminated by using exit() system call providing status as parameter. Exit may be called either directly as shown below or indirectly (by a return statement in main()).

exit(1)

A process that has been terminated but whose parent has not yet called wait() is known as **zombie process**. Once the process call wait(), the process identifier of the zombie process and its entry in the process table are released.

Inter-Process Communication:

Process executing concurrently in the operating system may require to exchange message between them. Such concurrently executing process may either be independent process or cooperating process. Any process that does not shares the data with any other process is **independent process**. Such process cannot affect or be affected by the other processes executing in the system.

Any process that shares data with other processes is a **cooperating process**. Such process can affect or be affected by other process executing in the system. There are several reason for providing inter process communication such as for information sharing, computation speedup, modularity, convenience.

Inter-process communication mechanism are required for cooperating process in order to exchange data and information. There are two fundamental model for inter-process communication:

- Shared memory
- Massage passing

1. **Shared Memory:**

   In shared memory, a region of the memory that is shared by the cooperating process is established. The process can then exchange information by reading and writing data to the shared region. A shared memory region resides in the address space of the process creating the shared memory segment. Other process that wish to communicate using this shared memory region must attach address space of the process that have created the shared memory region. Generally, operating system tries to prevent one process from accessing other process's space but for shared memory two process must have to remove this restriction. They can then exchange information by reading and writing data in the shared area.

   System calls are requires only to establish shared-memory region. Once shared memory is established, all access are treated as routine memory access and assistance from kernel is not required. The form of the data and location are determined by these process and are not under the operating system's control. Process are also responsible for ensuring that they are not writing to same location simultaneously.

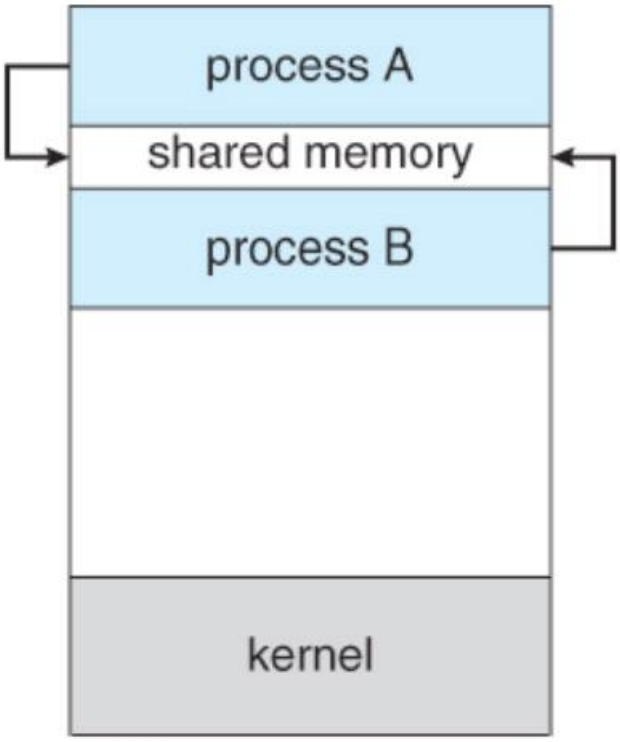


Figure: Shared Memory:

2. **Message passing system:**

In message passing system, communication takes place by means of message exchange between the cooperating processes. Message passing is useful for exchanging small amount of data. This mechanism allows processes to communicate and to synchronize their actions without sharing the same address space.

It is particularly useful in a distributed environment, where the communicating processes may reside on different computer connected by a network. A massage passing facility provides at least two operations: send (message) and receive (message). Message sent by a process can either be fixed or variable in size. This system are typically implemented using system calls and thus require more time-consuming task of kernel intervention.

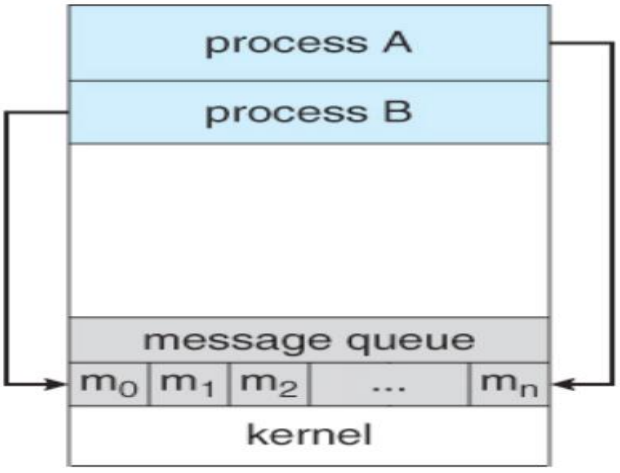


Figure: Message Passing System

For the two or more process to communicate, the communication link must exist between them. Here link are implemented logically and their methods are illustrated below:

- **Direct or indirect communication:**
  Under **direct communication**, each process that wants to communicate must explicitly name the recipient or sender of the communication. This scheme exhibits either symmetry in addressing where both the sender and the receiver process must name the other to communicate or asymmetry in address where only sender name the recipient but recipient is not required to name the sender. Both scheme are shown below:
  For symmetry:
  Send (P, message) – send a message to process P.
  Receiver (Q, message) – receive a message from process Q.
  For asymmetry:
  send (P, message) – send a message to process P.
  receiver (id, message) – receive a message from any process. Id specifies the name of the process with which communication has taken place.
  Here link is established automatically and associated with exactly two process with exactly one link.

  With the **indirect communication** the message are sent to receive from mail box or ports. Each mail-box have a unique identification and can be view abstractly as an object into which message can be placed and removed by process. Two process can communicate if they have shared mail-box. The send () and receive () primitive are shown below:
  send (A, message) – send a message to mailbox A
  receive (A, message) – receives a message form mailbox B.
  Here, a link may be associated with more than two processes but for two process to communicate they must share common mail box.

- **Synchronization:**
  Message passing may be either blocking or non-blocking also known as synchronous and asynchronous. In Blocking send: the sending process is blocked until the message is received by the receiving process or by the mailbox. In non-blocking send: the sending process sends the message and resumes operation. In Blocking receive: the receiver block until a message is available. In Non-blocking receive the receiver retrieves either a valid message or a null.
  When both send () and receive () are blocking there is a rendezvous point between the sender and the receiver.
- **Buffering:**
  Whether the communication is direct or indirect, message exchanged by communicating process reside in a temporary queue. Such queues can be implemented in three ways:
  - **Zero capacity:** the queue has maximum length of zero therefore, link cannot have any message waiting for it. The sender must block until

receiver receives the message. This case is sometimes refer as no buffering or automatic buffering.

- **Bounded capacity:** the queue has finite length n and at most n message can reside in it. If the queue is not full when a new message is sent then message are kept in queue and sender a can continue execution without waiting. If the link is full sender must block until space is available in the queue.
- **Unbounded capacity:** the queue's length is potentially infinite, thus any number of message can wait in it. The sender never block.

Process synchronization means sharing system resources by processes in such a way that, concurrent access to shared data is handled thereby minimizing the chance of inconsistent data. Maintaining data consistency demands mechanism to ensure synchronized execution of cooperative process. It handle the problem that arose while multiple process execute.

When multiple process are executing then a process may be interrupted at any point in its instruction stream and the processing core may be assigned to execute instruction of another process. In this case several process might be accessing and manipulating the same data resources (from common area) i.e. different parallel process may be reading and writing to same data resources. This makes the outcome of such resource inconsistent. Such case is known as race condition.

Race Condition: when several process access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place known as race condition.

To handle such case, we need process synchronization which will decide in which order process should execute i.e. which process will read at first and which will write at first from the common data resource. Synchronization is required For example:

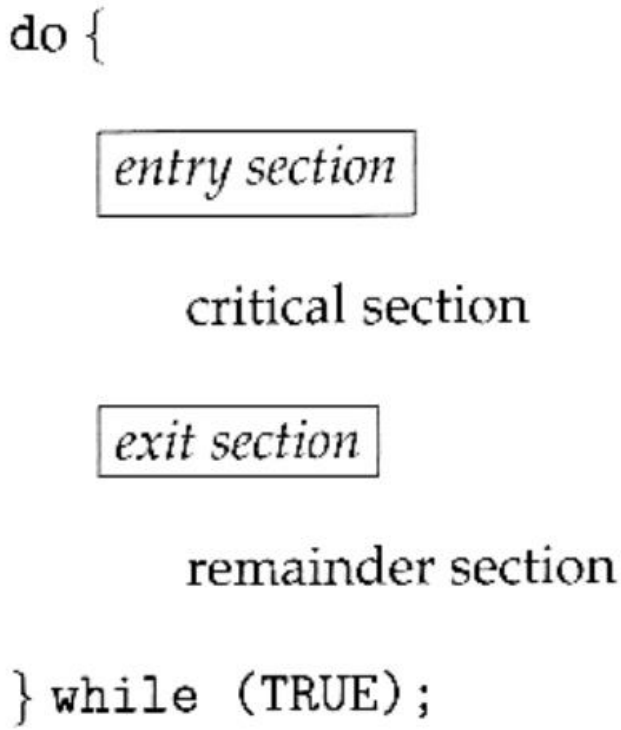
One process may be writing data to a certain main memory area while another process may be reading the data from that area and sending it to printer. The reader and writer must be synchronized so that writer does not overwrite.

Critical Section Problem:

Critical section is a piece of code that access a shared resources (data structure) that must not be concurrently accessed by more than one thread of execution. Each process have a segment of code called a critical section in which process may be changing common variable, accessing the common sharable variable, table, file etc. If a process at any point of time, wants to access common sharable variable, table or file then the process is trying to enter in its critical section.

When one process is executing in its critical section then no other process is allowed to execute in its critical section i.e. no two process are executing on critical section at the same time.

The general structure of typical process is shown below which contains three field:



- **Entry section:** section of code implementing or handling the number of request made by process to enter into critical section.
- Critical section: section of code in which only one process can execute at one time. If the OS grant the permission to process then process will enter in this section.
- Exit section: indicates the end of critical section, releasing the process from critical section.
- Remainder section: the remaining code after critical section.

A solution to the critical section problem must satisfy the following requirement:

1. **Mutual exclusion:** if the one process is executing in its critical section, then no other process are allowed to enter in its critical section. At a time only one process can enter into its critical section.
2. **Progress:** if no process is executing in its critical section and if some process wishes to enter into critical section then selection of such process cannot be postponed indefinitely. Only those process that are not executing in their remainder sections can participate in deciding which will enter its critical section.
3. **Bounded waiting**: there exist a bound or limit on the number of times the other processes are allowed to enter their critical section after a process has made a request to enter its critical section and before that the request is granted.

Solution of Critical Section Problem:

1. **Peterson's Solution**:
   Peterson's solution is a classical software based solution to the critical section problem which is restricted to two processes that alternate execution between their critical sections and remainder section. Let us consider two process be: Process (i) and Process (j).
   Following figure shows the structure of process Pi in Peterson's solution:

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

    Critical section

    flag[i] = false;

    Remainder section

} while (true);
```

Peterson solution requires the two process to share two data item:

```
int turn;
Boolean flag [i or j];
```

The variable turn indicates whose turn it is to enter its critical section i.e. if turn = = i then process i is allow to execute in its critical section. The flag array is used to indicate whether or not a process is ready to enter its critical section i.e. if flag [i] = = true then it indicates that the process i is ready to enter in its critical section.

To enter the critical section, process i first sets flag [i] to true and sets the turn to j indicating that if other process wishes to enter critical section it can do so. If both process try to enter at the same time then turn will be set to both i and j at roughly the same time but only one of these assignment will last, the other will occur but overwritten immediately.

To prove this solution is correct, following properties should be meet.

- Mutual exclusion is preserved.
- The progress requirement is satisfied.
- The bounded waiting requirement is met.

To prove the above property let us consider the following example:

For process i	For process j

do{
flag[i] = true
turn = j;
while (flag[j]==true && turn==j);
Critical section
flag[i] = false;
Remainder section;
}while(true);
	do{
	flag[j] = true
	turn = i;
	while (flag[i]==true && turn==i);
	Critical section
	flag[j] = false;
	Remainder section;
	}while(true);

Proving the mutual exclusion:

For the process i to enter on critical section it sets the flag [i] to true and turn = j. After that condition is checked of while loop. If both condition is matched process i will enter to while loop and if one condition is false it will goes to critical section.

Let us suppose, while process i is executing in its critical section, process j wishes to goes on critical section. In this condition we have flag[i]== true because process i haven't complete its critical section, process j sets its flag[j] to true (because it want to enter it its critical section) and sets turn = i. Now, the condition of while loop of process j is checked. Here, flag[i] is also true and turn is also i so, process j will enter into its while loop (not in critical section).

After the process i completes its critical section flag[i] is set to false. Now, the condition of while loop in process j will be false as flag[i] is set to false. Process j which was inside the while loop now, enters into critical section.

Therefore, it proves that if one process is executing in its critical section then no other process can enter in critical section at the same time. Hence, mutual exclusion is preserved.

Proving Progress:

We note that the process i can be prevented from entering the critical section only if it is stuck in its while loop with condition flag [j] = = true && turn = =j. If process j is not ready to enter in its critical section then flag [j] will be false and process i goes into critical section. If process j set flag [j] to true and is executing in its while loop then turn can be either i or j. If turn is i then process i will enter into critical section and if turn is j then process j will enter into critical section.

This proves that whenever the critical section is free then those process who makes the request at first will get chance to enter into critical section and if another process request, then it will not allow to enter at same time. This proves the progress requirement.

Proving Bounded Waiting:

When process i want to enter into critical section then it sets its flag to true and turn to j and enter into critical section if the condition flag[j] == true and turn == j does not matched. If process j makes request at this time then it will enter into its while loop. In this situation flag [i] is true, flag [j] is true and turn == i.

If process i completes its critical section then it sets its flag[i] to false. Now, we have flag[i] = false, flag [j] is true and turn == i. Now, if process i again wants to enter into its critical section then it sets flag[i] == true and turn to j. Here we have flag[i] = = true, flag[j] = = true (it is still in while loop) and turn = = j. Therefore, process i will not get chance to enter into critical section because condition of while loop is matched. So process j will enter into critical section.

It implies that process i's request to again entering into critical section is denied until process j complete its critical section. Hence bounded waiting requirement is met.

2. **Semaphore:**

Semaphore is a resource that contains an integer value and allows process to synchronize by testing and setting this value on a single atomic operation. It is an integer variable, apart from initialization is accessed through two standard atomic operation: wait () and signal (). It is a one kind of tool to prevent race condition. The wait () operation is used for testing whereas signal is used for increment. The process that test the value of semaphore and sets it to different value is guarantee no other process will interfere with the operation in the middle.

The definition of wait () is:

```
wait (S) {
while (S <= 0); // busy wait
    S--;
    }
```

The definition of signal () is:

```
signal (S) {
    S++;
    }
```

When one process modifies the value of semaphore then no other process can simultaneously modify that same semaphore value. In the case of wait() the testing of the integer value of S < = 0 as well as its possible modification S - - must be executed without interruption.

Work of wait () and signal ():

The **wait () operation** (it can be also called semWait() or down()) decrement the value of semaphore variable S if the condition is false in while loop and indicates that the process is interested to enter in critical section. The **signal () operation** (it can be also called semSignal() or up()) increment the value of semaphore variable S and indicates that process is terminated or has come out from its critical section.

Working Procedure of Semaphore:

Let us consider processes p1, p2…pn. Before any process request for critical section, the value of semaphore is initialize to one i.e. S=1. When process p1 request to enter into critical section, wait (S) operation is executed where condition S < = 0 is checked. Here, S = 1 so the condition did not matched. P1 will enter into critical section and value of S is decremented i.e. now S=0.

While P1 is executing in critical section and if process P2 makes request to enter into critical section then the wait (S) operation should be executed first where condition S <

= 0 is checked. In this time, S = 0 (P1 is still in critical section) so the condition is matched. P2 will not enter into critical section and falls in trap of while loop.
In some point of time, P1 will exit from critical section and execute signal () operation where the value of semaphore S is incremented by one. Now, S became one from 0 i.e. S=1. The condition S < = 0 is checked. Process P2 which was on trap of while loop now comes out as condition S<=0 becomes false. Now P2 will enter into critical section. Therefore, it implies that whenever one process is executing in its critical section no other process is allowed to enter in critical section at the same time. Hence, **mutual exclusion** is preserved.
Progress condition is automatically presents here because those process that want to enter into critical section will manipulate the value of S (increment and decrement) and no any process order is determined.

Types of Semaphore:

I. Counting semaphore:

Counting semaphore are used to control access to a given resource consisting of finite number of instances. The value of semaphore is initialized to the number of resource available. If any process wishes to use a resource perform wait () operation thereby decrementing the value of s. When a process release a resource it performs a signal () operation i.e. value of s is incremented. When the value of semaphore goes to zero then the process that wishes to use a resource is blocked until the S becomes greater than 0. For example if the resource is 10 then the value of semaphore is also 10 and 10 process can enter into critical section and after that all further process are blocked. The definition of counting semaphore is as follows:

wait (semaphore S){
S = S-1;
if (S<0){
Put process in suspend list, sleep()
}
else
return;
}
	Signal (S){
	S = S+1;
	If(S<=0){
	Select a process from suspend list, wakeup()
	}
	}

In this case, semaphore value is set to the number of resources available. The number of processes requesting to enter into critical section can enter until the condition S < 0 is false. When the condition S < 0 is true process requesting for critical section is blocked and put into suspend list.
Whenever the process want to exit from critical section signal operation is executed where the value of S is incremented and if the condition S<0 is true then the blocked process from suspend list are taken to ready queue using wakeup() operation and such process can now request for critical section.

II. Binary semaphore:

Binary semaphore are used to control access for single resources taking the value of either 0 indicating resource is in use or 1 indicating resource is available. It is use to prevent mutual exclusion. The definition of binary semaphore is:

Wait (semaphore S){	Signal(S){

If (S = = 1){
S = 0;
}
Else{
Block the process and place in suspend list, sleep();
}
}
	If (suspend list is empty){
	S = 1;
	}
	Else {
	Select the process from block list and wake up();

The value of semaphore S is 1 at first. Whenever a process makes request to enter in critical section condition S = = 1 is checked. If it is true then then value of S is changed to 0 and process goes into critical section. If condition does not matched then the process will be blocked.

Whenever the process want to exit from critical section signal operation is executed where suspend list is empty or not is checked and if it is empty value of S is changed to 1. If suspend list is not empty then the blocked process from suspend list is executed using wakeup () (brings to ready queue).

The disadvantage of semaphore is busy waiting that is when one process is in critical section then other process that request for critical section are loop indefinitely until the process exit from critical section.

3. **Mutex locks or lock variable:**

It is a simple tool that protect the critical region and thus prevent the race condition. Here, a process must acquire the lock before entering a critical section and releases the lock when it exist from the critical section. The acquire () function acquires the lock and release () function releases the lock. The definition of acquire () and release () is shown below:

acquire () {
while (!available); /* busy wait */
available = false;
}
	release (){
	available = true;
	}

Solution to critical section problem using mutex lock is shown below:

do {

acquire lock

Critical section

release lock

Remainder section

} while (true);

Whenever the process P1 request to enter into critical section first acquire () is executed. If available is false then process enter into while loop and if available is true P1 enter into critical section and the value of available is set to false. While executing in critical

section if another process P2 request to enter into critical section then it will enter into while loop (not in critical section) because available is false (P1 is still in critical section).

When P1 exit from critical section then release () is executed where the value of available is set to true. Now, P2 which was executing in while loop will exit and enter into critical section because available is true in this case (P1 is already exit from critical section). Therefore, it implies that when one process is executing in critical section then no other process are allowed to enter into critical section at the same time. Hence, mutual exclusion is preserved.

4. **Synchronization Hardware or test_and_set_lock:**

The critical section problem could be solved in a single processor environment if interrupt could be prevent from occurring while a shared variable was being modified. In this way no other instruction would be run so no unexpected modification could be made to the shared variable. But this solution is not as feasible in multiprocessor environment because disabling interrupts on a multiprocessor can be time consuming and system efficiency can be decreased.

Synchronization hardware mechanism is used to overcome the problem exist in mutex lock (mutual exclusion may no be hold in mutex lock).

To solve the critical section problem, many modern computer system provide special hardware instruction that allows either to test or modify the content of word or swap the content of two word atomically i.e. as one uninterruptible unit. Here test_and_set () instruction is used which is shown below:

```
Boolean test_and_set (boolean *target) {
    Boolean rv = *target;
    *target = true; //set to 1

    return rv;
    }
```

If the machine support test_and_set () instruction then mutual exclusion can be implemented by declaring a Boolean variable lock, initialize to false. The structure is shown below:

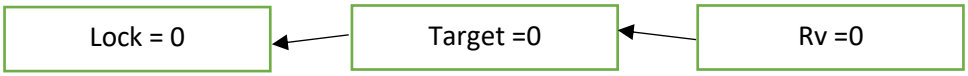
```
do {
while (test_and_set (&lock)); //do nothing
    critical section
    lock = false; //set to 0
    remainder section;
} while (true);
```

Note: value of lock = 0 (critical section empty)

Value of lock = 1 (critical section busy)

First the value of lock is set to 0 (critical section is empty). Suppose that process p1 request to go in critical section the it will execute while (test_and_set(&lock) function. Here, &lock is pointer and its initial value is 0 then Boolean test_and_set(Boolean * target) function is executed which have three code section:

- Boolean rv = * target (means that target contains lock so target will point to address of lock and value of target is assign to rv).
- *target = true; set the value of target to 1 (true).
- Return rv; return value of rv to function

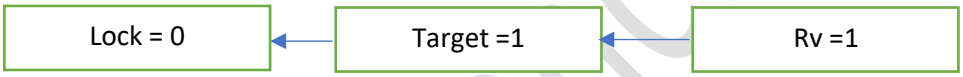


Address = 5 now target = 1 (true)

The value of rv is returned and while(test_and_set (&lock)) is now 0 and rv is returned to function. As condition in while loop is 0 (false) it will not get executed and process1 enter into critical section.

While p1 is in critical section and if p2 request to enter in critical section then it will execute while loop (test_and_set (&lock)) then control moves to Boolean test_and_set (boolean *target) which have three part

- Boolean rv = * target (means that target contains lock so target will point to address of lock and value of target is assign to rv). Here, value of target is 1 now so rv = 1;
- *target = true; set the value of target to 1 (true).
- Return rv; return value of rv (i.e. 1) to function



Now target = 1 again

The value of rv which is 1 is returned and while(test_and_set(&lock)) is now 1. Therefore, while loop get executed and p2 is not allowed to enter into critical section i.e. p2 will go on while loop. So while p1 is executing in critical section, p2 is not allowed to enter. Hence, mutual exclusion is preserved.

For progress: When critical section was empty, P1 or any process requested to enter will granted the access because at first, value of lock is 0 and value of target will also be 0 as target pointed to lock. So progress is preserved.

5. **Strict Alternation Approach or Turn Variable:**

It is the software mechanism which can be implemented only for two processes. Here, turn variable is used which is actually a lock. Let us consider two process: process I and process j.

For Process i	For process j

Non-critical section
//entry code
While (turn!=0); //busy wait
Critical section
Turn =1;//exit
	Non-critical section
	Entry code
	While(turn!=1);//busy wait
	Critical section
	Turn=0; //exit

At first, value of turn is 0 and if process 'I' request to enter into critical section then condition in while loop is checked. As, value of turn is 0 condition in while loop of process I is false so process I enter into critical section. While process I is executing in critical section if process j request to enter in critical section then while loop of process j is checked. Here, value of turn is still 0 as process I is in critical section so, condition of while loop of process j (turn!=1) i.e (0!=1) is true so process j will enter into while loop i.e. process j is not allowed to enter into critical section. After some time, if process I comes out of critical section then value of turn is changed into 1. Now, condition of while loop for process j is checked. Here, value of turn is 1 so condition turn!=1 i.e. 1!=1 is false so process j comes out of while loop and enter into critical section. Therefore, when one process is inside critical section then other process is not allowed to enter into critical section. So, **mutual exclusion** is preserved.

For progress: At first, value of turn is 0 and suppose process j request to enter into critical section then condition in its while loop (turn!=1) is checked i.e. (0!=1) which is true and j enter into while loop not in critical section. So progress is not satisfied here.

For bounded waiting: when process i comes out of critical section then value of turn is set to 1 and process j which was executing in its while loop checks the condition turn!=1. Now the value of turn is 1 so 1!=1 is false so process j enter into critical section. If process I again request to enter in critical section then condition of while loop (turn!=0) is checked i.e. (1!=0) which is true so process I cannot enter into critical section. Therefore, bounded waiting is preserved.