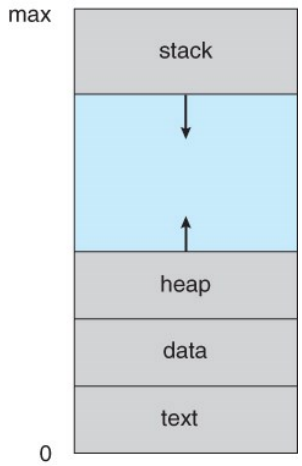
UNIT-2

Process: A process is a program in execution. The execution of a process must progress in a sequential fashion. Definition of process is following. A process is defined as an entity which represents the basic unit of work to be implemented in the system. Components of process are following. A program is passive entity whereas a process is an active entity. A program can be transformed to process in 2 ways: by double clicking the file and typing the file from command prompt.

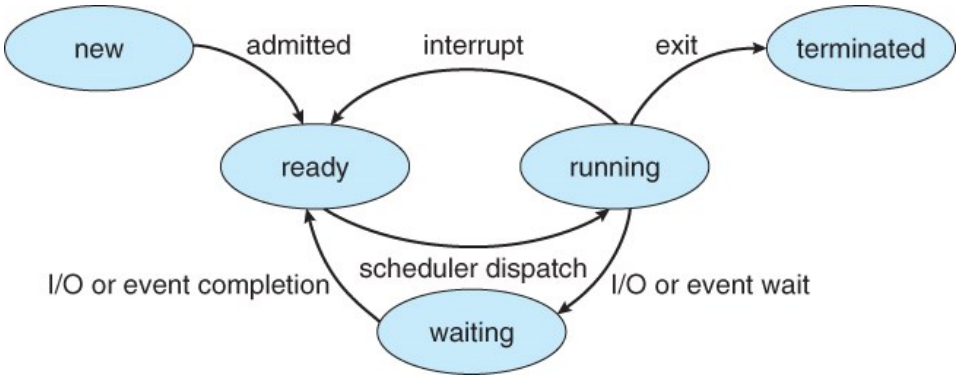
Process memory is divided into four sections as shown:

- The text section comprises the compiled program code, read in from non-volatile storage when the program is launched.
- The data section stores global and static variables, allocated and initialized prior to executing main.
- The heap is used for dynamic memory allocation, and is managed via calls to new, delete, malloc, free, etc.
- The stack is used for local variables. Space on the stack is reserved for local variables when they are declared (at function entrance or elsewhere, depending on the language), and the space is freed up when the variables go out of scope. Note that the stack is also used for function return values, and the exact mechanisms of stack management may be language specific.
- Note that the stack and the heap start at opposite ends of the process's free space and grow towards each other. If they should ever meet, then either a stack overflow error will occur, or else a call to new or malloc will fail due to insufficient memory available.
- When processes are swapped out of memory and later restored, additional information must also be stored and restored. Key among them are the program counter and the value of all program registers.



Process States: Processes may be in one of 5 states as shown:

- **New** - The process is in the stage of being created.
- **Ready** - The process has all the resources available that it needs to run, but the CPU is not currently working on this process's instructions.
- **Running** - The CPU is working on this process's instructions.
- **Waiting** - The process cannot run at the moment, because it is waiting for some resource to become available or 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.
- **Terminated -** The process has completed.



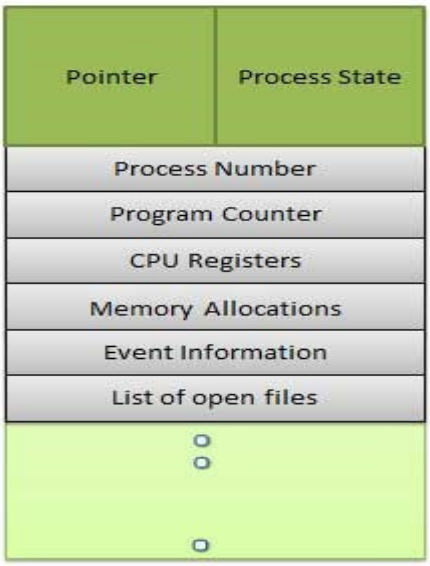
Process Control Block (PCB)

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

Each process is represented in the operating system by a process control block (PCB) also called a task control block. PCB is the data structure used by the operating system. Operating system groups all information that needs about particular process. PCB contains many pieces of information associated with a specific process which is described below.

S.N.	Information & Description
1	**Pointer** Pointer points to another process control block. Pointer is used for maintaining the scheduling list.
2	**Process State** Process state may be new, ready, running, waiting and so on.
3	**Program Counter** Program Counter indicates the address of the next instruction to be executed for this process.
4	**CPU registers** CPU registers include general purpose register, stack pointers, index registers and accumulators etc. number of register and type of register totally depends upon the computer

	architecture.
5	**Memory management information** This information may include the value of base and limit registers, the page tables, or the segment tables depending on the memory system used by the operating system. This information is useful for de-allocating the memory when the process terminates.
6	**Accounting information** This information includes the amount of CPU and real time used, time limits, job or process numbers, account numbers etc.



Process Control Block (**PCB**) includes CPU scheduling, I/O resource management, file management information etc.. The PCB serves as the repository for any information which can vary from process to process. Loader/linker sets flags and registers when a process is created. If that process gets suspended, the contents of the registers are saved on a stack and the pointer to the particular stack frame is stored in the PCB. By this technique, the hardware state can be restored so that the process can be scheduled to run again.

Fork System Call in Operating System

In many operating systems, the fork system call is an essential operation. The fork system call allows the creation of a new process. When a process calls the fork(), it duplicates itself, resulting in two processes running at the same time. The new process that is created is called a child process. It is a copy of the parent process. The fork system call is required for process creation and enables many important features such as parallel processing, multitasking, and the creation of complex process hierarchies.

Basic Terminologies Used in Fork System Call in Operating System

- **Process:** In an operating system, a process is an instance of a program that is currently running. It is a separate entity with its own memory, resources, CPU, I/O hardware, and files.
- **Parent Process:** The process that uses the fork system call to start a new child process is referred to as the parent process. It acts as the parent process's beginning point and can go on running after the fork.
- **Child Process:** The newly generated process as a consequence of the fork system call is referred to as the child process. It has its own distinct process ID (PID), and memory, and is a duplicate of the parent process.
- **Process ID:** A process ID (PID) is a special identification that the operating system assigns to each process.
- **Copy-on-Write:** The fork system call makes use of the memory management strategy known as copy-on-write. Until one of them makes changes to the shared memory, it enables the parent and child processes to share the same physical memory. To preserve data integrity, a second copy is then made.
- **Return Value:** The fork system call's return value gives both the parent and child process information. It assists in handling mistakes during process formation and determining the execution route.

Below are different values returned by fork().

- ***Negative Value***: The creation of a child process was unsuccessful.
- ***Zero***: Returned to the newly created child process.
- ***Positive value***: Returned to parent or caller. The value contains the process ID of the newly created child process.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{

    // make two process which run same
    // program after this instruction
    pid_t p = fork();
    if(p<0){
      perror("fork fail");
      exit(1);
    }
    printf("Hello world!, process_id(pid) = %d \n",getpid());
```

```
    return 0;
}
```

Output

```
Hello world!, process_id(pid) = 31
Hello world!, process_id(pid) = 32
```

vfork() :

Vfork() is also system call which is used to create new process. New process created by vfork() system call is called child process and process that invoked vfork() system call is called parent process. Code of child process is same as code of its parent process. Child process suspends execution of parent process until child process completes its execution as both processes share the same address space.

```
int main()
{ pid_t pid = vfork(); //creating the child process
 printf("parent process pid before if...else block: %d\n", getpid());
if (pid == 0)
{
 printf("This is the child process and pid is: %d\n\n", getpid());
 exit(0);
}
 else if (pid > 0)
 {
printf("This is the parent process and pid is: %d\n", getpid());
}
else { printf("Error while forking\n");
 exit(EXIT_FAILURE);
 }
return 0;
}
```

Process Scheduling

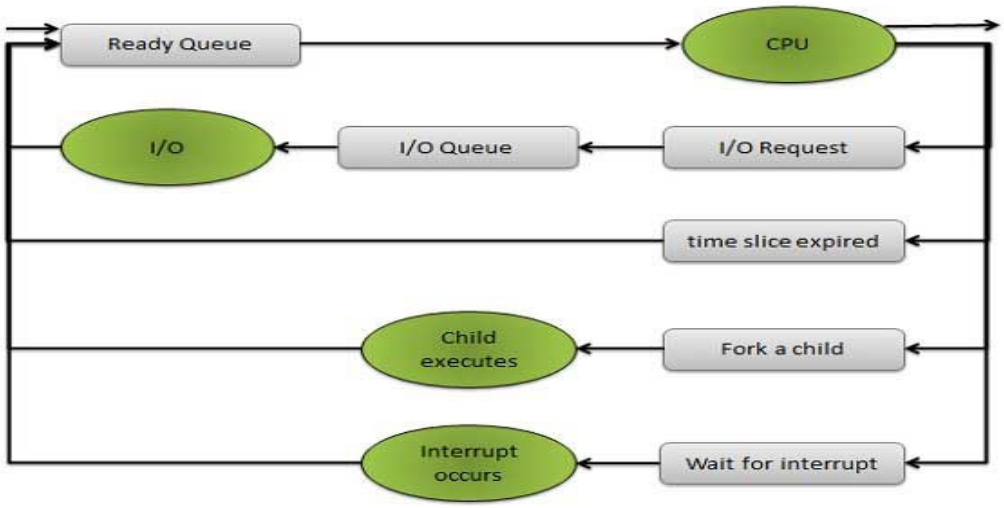
The process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy. It is an essential part of a Multiprogramming operating system. Such operating systems allow more than one process to be loaded into the executable memory at a time and loaded process shares the CPU using time multiplexing.

Scheduling Queues

- All processes are stored in the **job queue**.
- Processes in the Ready state are placed in the **ready queue**.
- Processes waiting for a device to become available or to deliver data are placed in **device queues**. There is generally a separate device queue for each device.
- Other queues may also be created and used as needed.

Queuing diagram:

- Queue is represented by rectangular box.
- The circles represent the resources that serve the queues.
- The arrows indicate the process flow in the system.



Schedulers

Schedulers are special system software's which handles process scheduling in various ways.Their main task is to select the jobs to be submitted into the system and to decide which process to run. Schedulers are of three types:

- Long Term Scheduler
- Short Term Scheduler
- Medium Term Scheduler

Long Term Scheduler

It is also called a job scheduler. Long term scheduler determines which programs are admitted to the system for processing. Job scheduler selects processes from the queue and loads them into memory for execution. The primary objective of the job scheduler is to provide a balanced mix of jobs, such as I/O bound and processor bound. It also controls the degree of multiprogramming. If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system.

On some systems, the long term scheduler may not be available or minimal. Time-sharing operating systems have no long term scheduler. When process changes the state from new to ready, then there is use of long term scheduler.

Short Term Scheduler

It is also called CPU scheduler. Main objective is increasing system performance in accordance with the chosen set of criteria. It is the change of ready state to running state of the process. CPU scheduler selects process among the processes that are ready to execute and allocates CPU to one of them.

Short term scheduler also known as dispatcher, executed most frequently and makes the fine grained decision of which process to execute next. Short term scheduler is faster than Long term scheduler.

Medium Term Scheduler

Medium term scheduling is part of the swapping. It removes the processes from the memory. It reduces the degree of multiprogramming. The medium term scheduler is in-charge of handling the swapped out-processes.

Running process may become suspended if it makes an I/O request. Suspended processes cannot make any progress towards completion. In this condition, to remove the process from memory and make space for other process, the suspended process is moved to the secondary storage. This process is called swapping, and the process is said to be swapped out or rolled out. Swapping may be necessary to improve the process mix.

Comparison between Schedulers

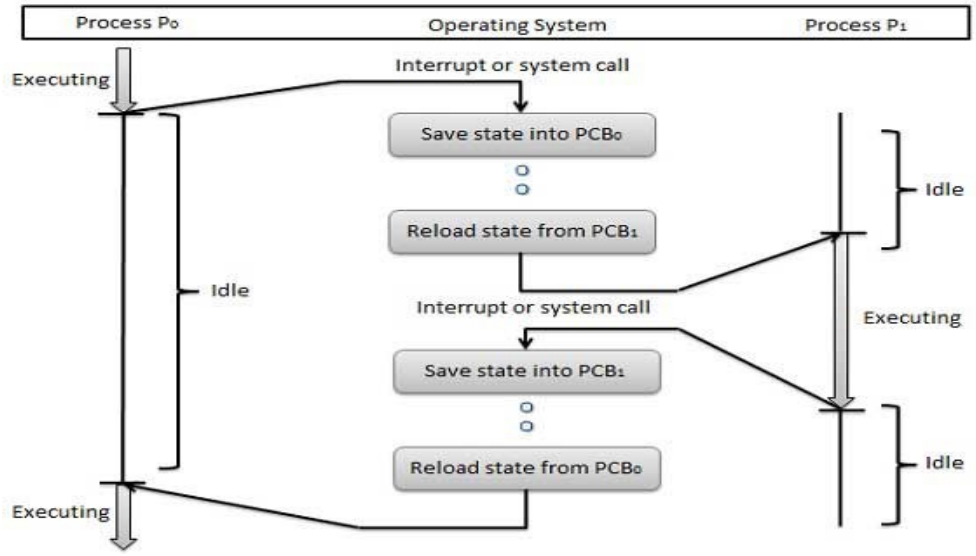
S.N.	Long Term Scheduler	Short Term Scheduler	Medium Term Scheduler
1	It is a job scheduler	It is a CPU scheduler	It is a process swapping scheduler.
2	Speed is lesser than short term scheduler	Speed is fastest among other two	Speed is in between both short and long term scheduler.
3	It controls the degree of multiprogramming	It provides lesser control over degree of multiprogramming	It reduces the degree of multiprogramming.

4	It is almost absent or minimal in time sharing system	It is also minimal in time sharing system	It is a part of Time sharing systems.
5	It selects processes from pool and loads them into memory for execution	It selects those processes which are ready to execute	It can re-introduce the process into memory and execution can be continued.

Context Switch

A context switch is the mechanism to store and restore the state or context of a CPU in Process Control block so that a process execution can be resumed from the same point at a later time. Using this technique a context switcher enables multiple processes to share a single CPU. Context switching is an essential part of a multitasking operating system features. When the scheduler switches the CPU from executing one process to execute another, the context switcher saves the content of all processor registers for the process being removed from the CPU, in its process descriptor. The context of a process is represented in the process control block of a process.

Context switch time is pure overhead. Context switching can significantly affect performance as modern computers have a lot of general and status registers to be saved. Content switching times are highly dependent on hardware support. Context switch requires (n + m) bxK time units to save the state of the processor with 'n' general registers, assuming 'b' are the store operations are required to save 'n' and 'm' registers of two process control blocks and each store instruction requires 'K' time units.



CPU Scheduler

Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them

CPU scheduling decisions may take place when a process:

1. Switches from running to waiting state

2. Switches from running to ready state

3. Switches from waiting to ready

4. Terminates

Scheduling under 1 and 4 is **non-preemptive and** all other scheduling is **preemptive**

Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
- switching context
- switching to user mode
- jumping to the proper location in the user program to restart that program
- **Dispatch latency** – time it takes for the dispatcher to stop one process and start another running

Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – # of processes that complete their execution per time unit
- **Turnaround time** – amount of time to execute a particular process
- **Waiting time** – amount of time a process has been waiting in the ready queue
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)
- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

Scheduling algorithms

Four major scheduling algorithms here which are following :

- First Come First Serve (FCFS) Scheduling
- Shortest-Job-First (SJF) Scheduling
- Priority Scheduling
- Round Robin(RR) Scheduling

First Come First Serve (FCFS)

- Jobs are executed on first come, first serve basis.
- Easy to understand and implement.

- Poor in performance as average wait time is high.

Process	Arrival Time	Execute Time	Service Time
P0	0	5	0
P1	1	3	5
P2	2	8	8
P3	3	6	16

Wait time of each process is following

Process	Wait Time : Service Time - Arrival Time
P0	0 - 0 = 0
P1	5 - 1 = 4
P2	8 - 2 = 6
P3	16 - 3 = 13

Average Wait Time: (0+4+6+13) / 4 = 5.55

Shortest Job First (SJF)

- Best approach to minimize waiting time.
- Impossible to implement
- Processer should know in advance how much time process will take.

Process	Arrival Time	Execute Time	Service Time
P0	0	5	0
P1	1	3	3
P2	2	8	8
P3	3	6	16

Wait time of each process is following

Process	Wait Time : Service Time - Arrival Time
P0	3 - 0 = 3
P1	0 - 0 = 0
P2	16 - 2 = 14

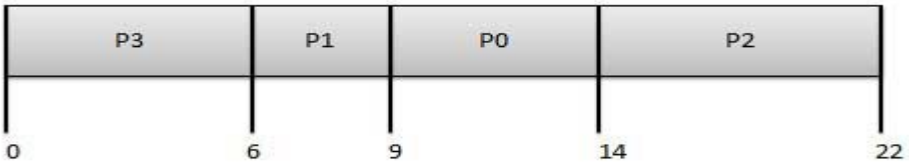
P3	8 - 3 = 5

Average Wait Time: (3+0+14+5) / 4 = 5.50

Priority Based Scheduling

- Each process is assigned a priority. Process with highest priority is to be executed first and so on.
- Processes with same priority are executed on first come first serve basis.
- Priority can be decided based on memory requirements, time requirements or any other resource requirement.

Process	Arrival Time	Execute Time	Priority	Service Time
P0	0	5	1	0
P1	1	3	2	3
P2	2	8	1	8
P3	3	6	3	16



Wait time of each process is following

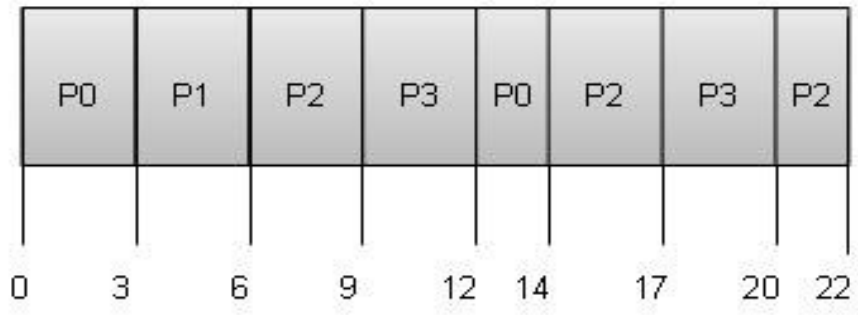
Process	Wait Time : Service Time - Arrival Time
P0	9 - 0 = 9
P1	6 - 1 = 5
P2	14 - 2 = 12
P3	0 - 0 = 0

Average Wait Time: (9+5+12+0) / 4 = 6.5

Round Robin Scheduling

- Each process is provided a fix time to execute called quantum.
- Once a process is executed for given time period. Process is preempted and other process executes for given time period.
- Context switching is used to save states of preempted processes.

Quantum = 3



Wait time of each process is following

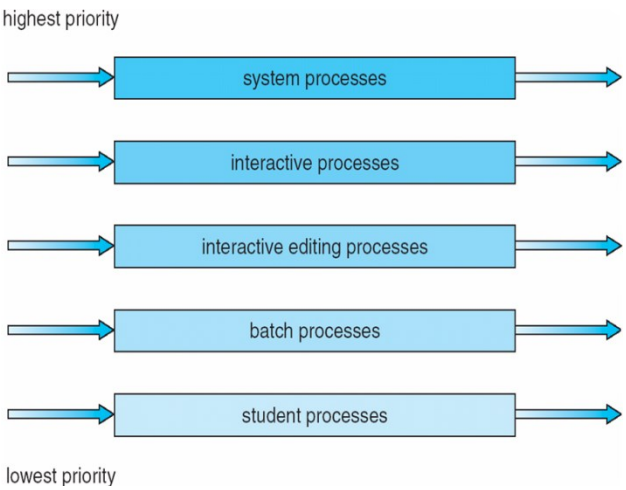
Process	Wait Time : Service Time - Arrival Time
P0	(0-0) + (12-3) = 9
P1	(3-1) = 2
P2	(6-2) + (14-9) + (20-17) = 12
P3	(9-3) + (17-12) = 11

Average Wait Time: (9+2+12+11) / 4 = 8.5

1. Multilevel Queue

- Ready queue is partitioned into separate queues: foreground (interactive) background (batch)
- Each queue has its own scheduling algorithm
- foreground – RR
- background – FCFS
- Scheduling must be done between the queues
- Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
- Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR

20% to background in FCFS



2.Multilevel Feedback Queue

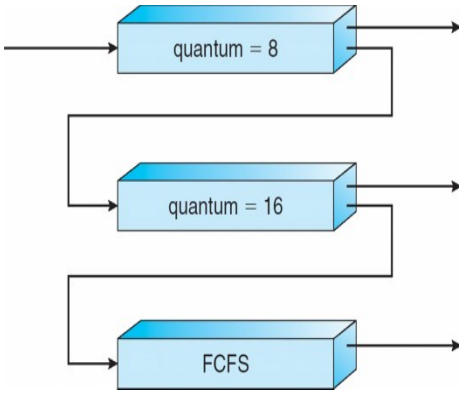
- A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
- number of queues
- scheduling algorithms for each queue
- method used to determine when to upgrade a process
- method used to determine when to demote a process
- method used to determine which queue a process will enter when that process needs service

Example of Multilevel Feedback Queue

Three queues:

- Q_0 – RR with time quantum 8 milliseconds
- Q_1 – RR time quantum 16 milliseconds
- Q_2 – FCFS
- Scheduling
- A new job enters queue Q_0which is servedFCFS. When it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds, job is moved to queue Q_1.
- At Q_1 job is again served FCFS and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue Q_2.

Multilevel Feedback Queues

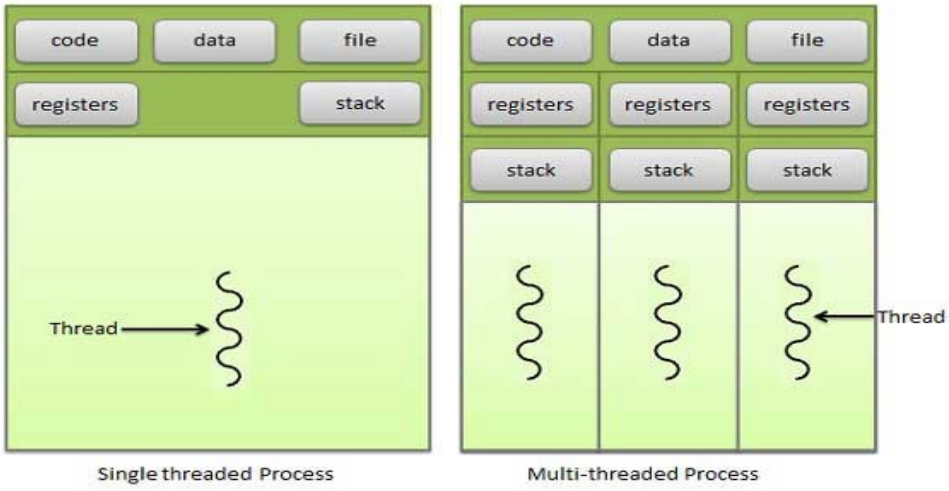


Thread Concepts:

A thread is a flow of execution through the process code, with its own program counter, system registers and stack. A thread is also called a light weight process. Threads provide a way to improve application performance through parallelism. Threads represent a software approach to improving performance of operating system by reducing the overhead thread is equivalent to a classical process.

Each thread belongs to exactly one process and no thread can exist outside a process. Each thread represents separate flows of control.Threads have been successfully used in implementing network servers and web server. They also provide a suitable foundation for

parallel execution of applications on shared memory multiprocessors. Following figure shows the working of the single and multithreaded processes.



Difference between Process and Thread

S.N.	Process	Thread
1	Process is heavy weight or resource intensive.	Thread is light weight taking lesser resources than a process.
2	Process switching needs interaction with operating system.	Thread switching does not need to interact with operating system.
3	In multiple processing environments each process executes the same code but has its own memory and file resources.	All threads can share same set of open files, child processes.
4	If one process is blocked then no other process can execute until the first process is unblocked.	While one thread is blocked and waiting, second thread in the same task can run.
5	Multiple processes without using threads use more resources.	Multiple threaded processes use fewer resources.
6	In multiple processes each process operates independently of the others.	One thread can read, write or change another thread's data.

Advantages of Threads

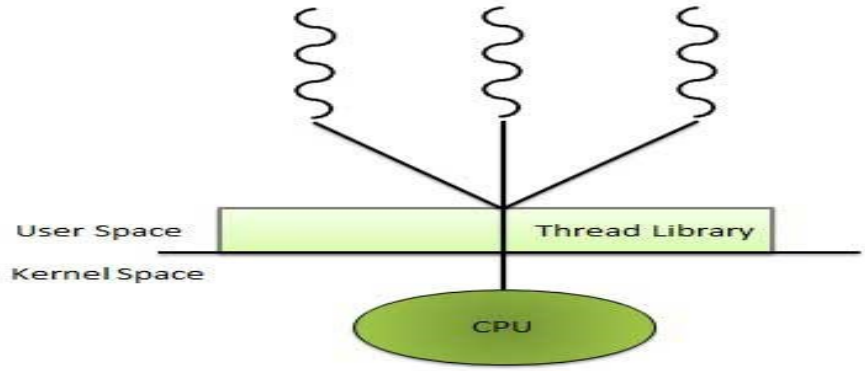
- Threads minimize context switching time.
- Use of threads provides concurrency within a process.
- Efficient communication.
- Economy- It is more economical to create and context switch threads.
- Utilization of multiprocessor architectures to a greater scale and efficiency.

Types of Threads

Threads are implemented in following two ways:

- **User Level Threads** -- User managed threads
- **Kernel Level Threads** -- Operating System managed threads acting on kernel, an operating system core.

User Level Threads: In this case, application manages thread management, kernel is not aware of the existence of threads. The thread library contains code for creating and destroying threads, for passing message and data between threads, for scheduling thread execution and for saving and restoring thread contexts. The application begins with a single thread and begins running in that thread.



Advantages

- Thread switching does not require Kernel mode privileges.
- User level thread can run on any operating system.
- Scheduling can be application specific in the user level thread.
- User level threads are fast to create and manage.

Disadvantages

- In a typical operating system, most system calls are blocking.
- Multithreaded application cannot take advantage of multiprocessing.

Kernel Level Threads

In this case, thread management is done by the Kernel. There is no thread management code in the application area. Kernel threads are supported directly by the operating system. Any application can be programmed to be multithreaded. All of the threads within an application are supported within a single process.

The Kernel maintains context information for the process as a whole and for individual threads within the process. Scheduling by the Kernel is done on a thread basis. The Kernel performs thread creation, scheduling and management in Kernel space. Kernel threads are generally slower to create and manage than the user threads.

Advantages

- Kernel can simultaneously schedule multiple threads from the same process on multiple processes.
- If one thread in a process is blocked, the Kernel can schedule another thread of the same process.
- Kernel routines themselves can multithreaded.

Disadvantages

- Kernel threads are generally slower to create and manage than the user threads.
- Transfer of control from one thread to another within same process requires a mode switch to the Kernel.

Multithreading Models

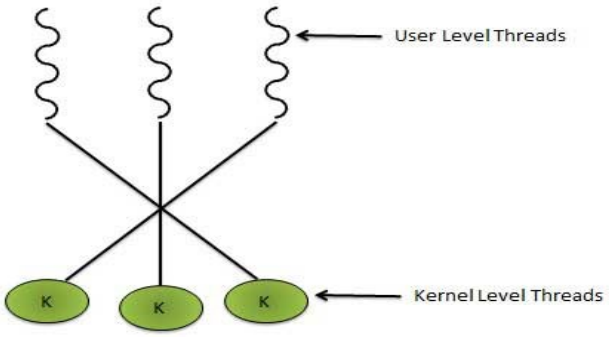
Some operating system provides a combined user level thread and Kernel level thread facility. Solaris is a good example of this combined approach. In a combined system, multiple threads within the same application can run in parallel on multiple processors and a blocking system call need not block the entire process. Multithreading models are three types :

- Many to many relationship.
- Many to one relationship.
- One to one relationship.

Many to Many Models

In this model, many user level threads multiplex to the Kernel thread of smaller or equal numbers. The number of Kernel threads may be specific to either a particular application or a particular machine.

Following diagram shows the many to many models. In this model, developers can create as many user threads as necessary and the corresponding Kernel threads can run in parallels on a multiprocessor.

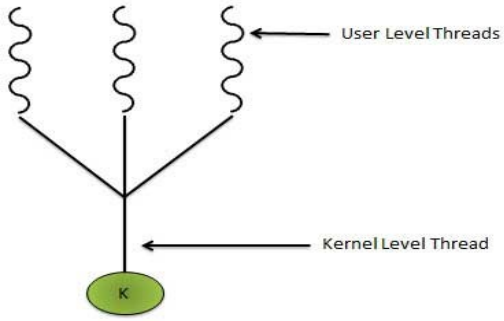


Many to One Model

Many to one model maps many user level threads to one Kernel level thread. Thread management is done in user space. When thread makes a blocking system call, the entire

process will be blocks. Only one thread can access the Kernel at a time,so multiple threads are unable to run in parallel on multiprocessors.

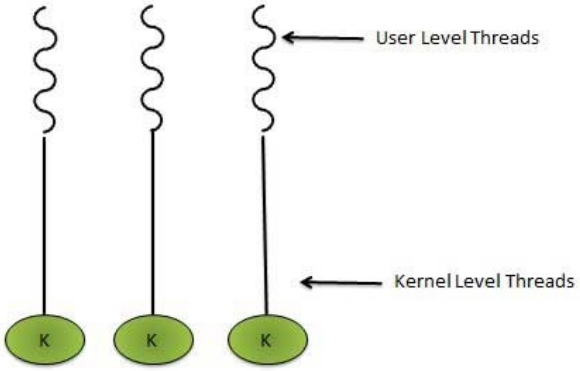
If the user level thread libraries are implemented in the operating system in such a way that system does not support them then Kernel threads use the many to one relationship modes.



One to One Model

There is one to one relationship of user level thread to the kernel level thread.This model provides more concurrency than the many to one model. It allows another thread to run when a thread makes a blocking system call. It supports multiple threads to execute in parallel on microprocessors.

Disadvantage of this model is that creating user thread requires the corresponding Kernel thread. OS/2 , Windows NT and Windows 2000 use one to one relationship model.



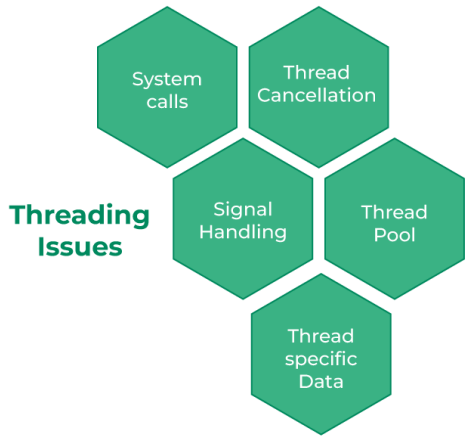
Difference between User Level & Kernel Level Thread

S.N.	User Level Threads	Kernel Level Thread
1	User level threads are faster to create and manage.	Kernel level threads are slower to create and manage.
2	Implementation is by a thread library at the user level.	Operating system supports creation of Kernel threads.
3	User level thread is generic and can run on any operating system.	Kernel level thread is specific to the operating system.
4	Multi-threaded application	Kernel routines themselves can

	cannot take advantage of multiprocessing.	be multithreaded.

Threading Issues in OS

- System Call
- Thread Cancellation
- Signal Handling
- Thread Pool
- Thread Specific Data



fork() and exec() System Calls

They are the system calls fork() and exec(). Fork() function gives rise to an identical copy of process which initiated fork call. The new duplication process is referred to as a child process, while the invoker process is identified by fork(). The instruction after fork continues the execution with both the parent process and the child process.

Discussing fork() system call, therefore. Let us assume that one of the threads belonging to a multi-threaded program has instigated a fork() call. Therefore, the new process is a duplication of fork(). Here, the question is as follows; will the new duplicate process made by fork() be multi-threaded like all threads of the old process or it will be a unique thread?

Now, certain UNIX systems have two variants of fork(). fork can either duplicate all threads of the parent process to a child process or just those that were invoked by the parent process. The application will determine which version of fork() to use.
When the next system call, namely exec() system call is issued, it replaces the whole programming with all its threads by the program specified in the exec() system call's parameters. Ordinarily, the exec() system call goes into queue after the fork() system call.
However, this implies that the exec() system call should not be queued immediately after the fork() system call because duplicating all the threads of the parent process into the child process will be superfluous. Since the exec() system call will overwrite the whole process with the one given in the arguments passed to exec().

Thread Cancellation

The process of prematurely aborting an active thread during its run is called 'thread cancellation'. So, let's take a look at an example to make sense of it. Suppose, there is a multithreaded program whose several threads have been given the right to scan a database for some information. The other threads however will get canceled once one of the threads happens to return with the necessary results.
The target thread is now the thread that we want to cancel. Thread cancellation can be done in two ways:

- Asynchronous Cancellation: The asynchronous cancellation involves only one thread that cancels the target thread immediately.
- Deferred Cancellation: In the case of deferred cancellation, the target thread checks itself repeatedly until it is able to cancel itself voluntarily or decide otherwise.

The issue related to the target thread is listed below:
How is it managed when resources are assigned to a canceled target thread?
Suppose the target thread exits when updating the information that is being shared with other threads.
However, in here, asynchronous threads cancellation whereas thread cancels its target thread irrespective of whether it owns any resource is a problem.
On the other hand, the target thread receives this message first and then checks its flag to see if it should cancel itself now or later. They are called the Cancellation Points of threads under which thread cancellation occurs safely.

Signal Handling

Signal is easily directed at the process in single threaded applications. However, in relation to multithreaded programs, the question is which thread of a program the signal should be sent.
Suppose the signal will be delivered to:

- Every line of this process.
- Some special thread of a process.
- thread to which it applies

Alternatively, you could give one thread the job of receiving all signals.
So, the way in which the signal shall be passed to the thread depends on how the signal has been created. The generated signals can be classified into two types: synchronous signal and asynchronous signal.
At this stage, the synchronous signals are routed just like any other signal was generated. Since these signals are triggered by events outside of the running process, they are received by the running process in an asynchronous manner, referred to as asynchronous signals.
Therefore, if the signal is synchronous, it will be sent to a thread that generated such a signal. The asynchronous signal cannot be determined into which thread in a multithreaded program delivery it should go. The asynchronous signal that is telling a process to stop, will result in all threads of the process receiving the signal.
Many UNIX UNITS have addressed, to some extent, the problem of asynchronous signals. Here, the thread is given an opportunity to identify the relevant or valid signals and those that it does not support. Windows OS on the other hand, has no idea about signals but does use ACP as equivalent for asynchronous signals adopted in Unix platforms.
In contrast with UNIX where a thread specifies that it can or cannot receive a thread, all control process instances (ACP) are sent to a particular thread.

Thread Pool

The server develops an independent thread every time an individual attempts to access a page on it. However, the server also has certain challenges. Bear in mind that no limit in the number of active threads in the system will lead to exhaustion of the available system resources because we will create a new thread for each request.
The establishment of a fresh thread is another thing that worries us. The creation of a new thread should not take more than the amount of time used up by the thread in dealing with the request and quitting after because this will be wasted CPU resources.
Hence, thread pool could be the remedy for this challenge. The notion is that as many fewer threads as possible are established during the beginning of the process. A group of threads that forms this collection of threads is referred as a thread pool. There are always threads that stay on the thread pool waiting for an assigned request to service.

Thread Specific Data

Of course, we all know that a thread belongs to data of one and the same process, right?. The challenge here will be when every thread in that process must have its own copy of the same data. Consequently, any data uniquely related to a particular thread is referred to as thread-specific data.

For example, a transaction processing system can process a transaction in individual threads for each one.I Each transaction we perform shall be assigned with a special identifier which in turn, shall uniquely identify that particular transaction to us. The system would then be in a position to distinguish every transaction among others.

Because we render each transaction separately on a thread. In this way, thread-specific datas will allow associating every thread with a definite transaction and some transaction ID. For example, libraries that support threads, namely Win32, Pthreads and Java, provide for thread-specific data (TSD).

Hence, these are the threading problems that arise in multithreaded programming environments. Additionally, we examine possible ways of addressing these concerns.

As a result, multithreading may be regarded as an integral operation within computer programming enhancing task concurrency and improved system performance. however, it is also associated with some problems and threading issues. There were numerous threading concerns such as system calls, thread cancellation, threads' pools and thread specific date.

Inter Process Communication (IPC)

A process can be of two types:

- Independent process.
- Co-operating process.

An independent process is not affected by the execution of other processes while a co-operating process can be affected by other executing processes. Though one can think that those processes, which are running independently, will execute very efficiently, in reality, there are many situations when co-operative nature can be utilized for increasing computational speed, convenience, and modularity. Inter-process communication (IPC) is a mechanism that allows processes to

communicate with each other and synchronize their actions. The communication between these processes can be seen as a method of co-operation between them. Processes can communicate with each other through both:

1. Shared Memory
2. Message passing

An operating system can implement both methods of communication. First, we will discuss the shared memory methods of communication and then message passing. Communication between processes using shared memory requires processes to share some variable, and it completely depends on how the programmer will implement it. One way of communication using shared memory can be imagined like this: Suppose process1 and process2 are executing simultaneously, and they share some resources or use some information from another process. Process1 generates information about certain computations or resources being used and keeps it as a record in shared memory. When process2 needs to use the shared information, it will check in the record stored in shared memory and take note of the information generated by process1 and act accordingly. Processes can use shared memory for extracting information as a record from another process as well as for delivering any specific information to other processes.

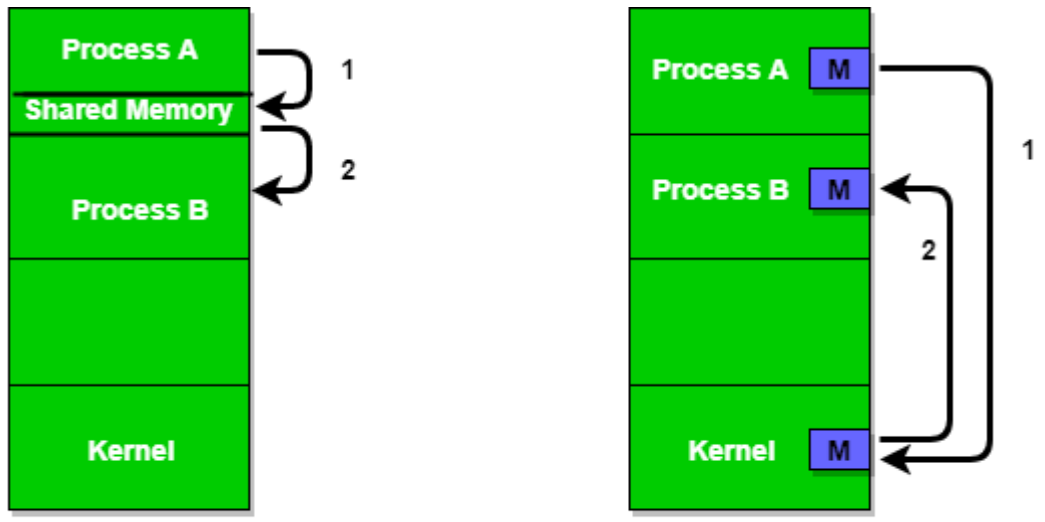


Figure 1 - Shared Memory and Message Passing

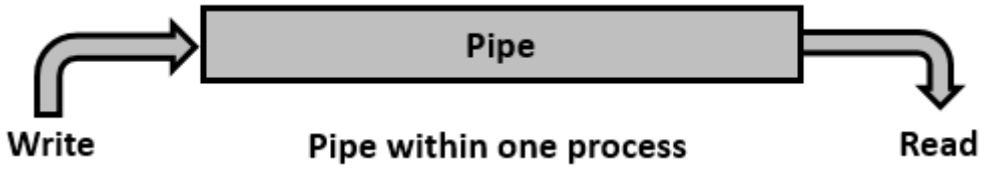
Approaches for Inter-Process Communication

Pipes

Pipe is widely used for communication between two related processes.

Pipe is a communication medium between two or more related or interrelated processes. It can be either within one process or a communication between the child and the parent processes. Communication can also be multi-level such as communication between the parent, the child and the grand-child, etc. Communication is achieved by one process writing into the pipe and other reading from the pipe. To achieve the pipe system call, create two files, one to write into the file and another to read from the file.

Pipe mechanism can be viewed with a real-time scenario such as filling water with the pipe into some container, say a bucket, and someone retrieving it, say with a mug. The filling process is nothing but writing into the pipe and the reading process is nothing but retrieving from the pipe. This implies that one output (water) is input for the other (bucket).



This system call would create a pipe for one-way communication i.e., it creates two descriptors, first one is connected to read from the pipe and other one is connected to write into the pipe.

Descriptor pipedes[0] is for reading and pipedes[1] is for writing. Whatever is written into pipedes[1] can be read from pipedes[0].

Message Passing

It is a mechanism for a process to communicate and synchronize. Using message passing, the process communicates with each other without resorting to shared variables.

IPC mechanism provides two operations:

- Send (message)- message size fixed or variable
- Received (message)

Message Queues

A message queue is a linked list of messages stored within the kernel. It is identified by a message queue identifier. This method offers communication between single or multiple processes with full-duplex capacity.

Direct Communication

In this type of inter-process communication process, should name each other explicitly. In this method, a link is established between one pair of communicating processes, and between each pair, only one link exists.

Indirect Communication

Indirect communication establishes like only when processes share a common mailbox each pair of processes sharing several

communication links. A link can communicate with many processes. The link may be bi-directional or unidirectional.

Shared Memory

Shared memory is a memory shared between two or more processes that are established using shared memory between all the processes. This type of memory requires to protected from each other by synchronizing access across all the processes.

FIFO

Communication between two unrelated processes.

Named pipes provide communication between related or unrelated processes that are subject to security checks. They can be used for communication between processes on the same computer or different computers.

What is Concurrency in OS?

Concurrency in operating systems refers to the ability of an OS to manage and execute multiple tasks or processes simultaneously. It allows multiple tasks to overlap in execution, giving the appearance of parallelism even on single-core processors. Concurrency is achieved through various techniques such as multitasking, multithreading, and multiprocessing.

Principles of Concurrency in Operating Systems

To effectively implement concurrency, OS designers adhere to several key principles:

- **Process Isolation:**
  Each process should have its own memory space and resources to prevent interference between processes. This isolation is critical to maintain system stability.
- **Synchronization:**
  Concurrency introduces the possibility of data races and conflicts. Synchronization mechanisms like locks, semaphores, and mutexes are used to coordinate access to shared resources and ensure data consistency.
- **Deadlock Avoidance:**
  OSs implement algorithms to detect and avoid deadlock situations where processes are stuck waiting for resources indefinitely. Deadlocks can halt the entire system.
- **Fairness:**
  The OS should allocate CPU time fairly among processes to prevent any single process from monopolizing system resources.

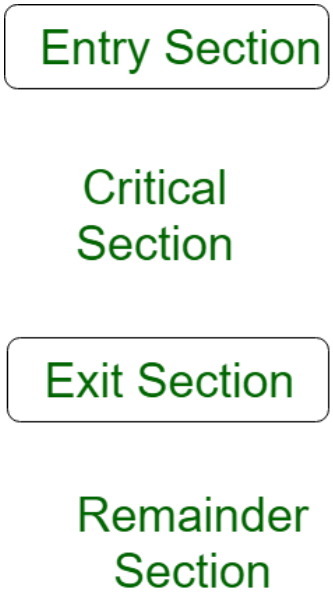
Process Synchronization

What is Race Condition?

Race Condition occurs when more than one process tries to access and modify the same shared data or resources because many processes try to modify the shared data or resources there are huge chances of a process getting the wrong result or data. Therefore, every process race to say that it has correct data or resources and this is called a race condition.

What is the Critical Section Problem?

What the critical section do is, make sure that only one process at a time has access to shared data or resources and only that process can modify that data. Thus when many problems try to modify the shared data or resources critical section allows only a single process to access and modify the shared data or resources.



What are the Rules of Critical Sections?

here are basically three rules which need to be followed to solve critical section problems.

- **Mutual Exclusion:-** Make sure one process is running is the critical section means one process is accessing the shared data or resources then no other process enters 3the critical section at the same time.
- **Progress:-** If there is no process in the critical section and some other processes are waiting to enter into the critical section. Now which process will enter into the critical section is taken by these processes.
- **Bounding waiting:-** When a new process makes a request to enter into the critical section there should be some waiting time or bound. This bound time is equal to the number of processes that are allowed to access critical sections before it.

Solutions to the Critical Section Problem:-

Software solutions:peterson's solution

Semaphores

Monitors

Hardware solutions : test and set

Synchronization hardware

Peterson's Solution

- Two process solution
- Assume that the LOAD and STORE instructions are atomic; that is, can't be interrupted.
- The two processes share two variables:
- int turn;
- Boolean flag[2]
- The variable turn indicates whose turn it is to enter the critical section.
- The flag array is used to indicate if a process is ready to enter the critical section. flag[i] = true implies that process P_i is ready!

Algorithm for Process P

```
do {
        flag[i] = TRUE;
```

```
        turn = j;
        while (flag[j] && turn == j);
                critical section
        flag[i] = FALSE;
                remainder section
} while (TRUE);
```

Mutual exclusion[edit]

P0 and P1 can never be in the critical section at the same time. If P0 is in its critical section, then `flag[0]` is true. In addition, either `flag[1]` is `false` (meaning that P1 has left its critical section), or `turn` is `0` (meaning that P1 is just now trying to enter the critical section, but graciously waiting), or P1 is at label `P1_gate` (trying to enter its critical section, after setting `flag[1]` to `true` but before setting `turn` to `0` and busy waiting). So if both processes are in their critical sections, then we conclude that the state must satisfy `flag[0]` and `flag[1]` and `turn = 0` and `turn = 1`. No state can satisfy both `turn = 0` and `turn = 1`, so there can be no state where both processes are in their critical sections.

Progress[edit]

Progress is defined as the following: if no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in making the decision as to which process will enter its critical section next. Note that for a process or thread, the remainder sections are parts of the code that are not related to the critical section. This selection cannot be postponed indefinitely.[3] A process cannot immediately re-enter the critical section if the other process has set its flag to say that it would like to enter its critical section.

Bounded waiting[edit]

Bounded waiting, or bounded bypass, means that the number of times a process is bypassed by another process after it has indicated its desire to enter the critical section is bounded by a function of the number of processes in the system.[3][4]:11 In Peterson's algorithm, a process will never wait longer than one turn for entrance to the critical section.

Semaphore

Semaphore as General Synchronization Tool

- Counting semaphore – integer value can range over an unrestricted domain
- Binary semaphore – integer value can range between 0 & 1; simpler to implement

- Also known as mutex locks. Can implement a counting semaphore S as a binary semaphore
- Provides mutual exclusionSemaphoremutex; // initialized to do {

wait (mutex);

// Critical Section

signal (mutex);

// remainder section

} while (TRUE);

Monitors:

Monitor in an operating system is one method for achieving process synchronization. Programming languages help the monitor to accomplish mutual exclusion between different activities in a system

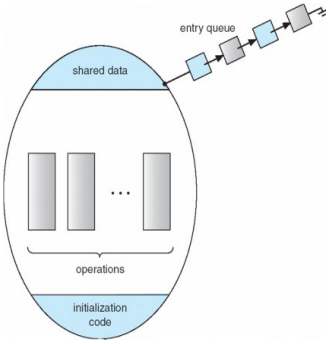
Syntax of Monitor in OS

```
Monitor monitorName{
    variables_declaration;
    condition_variables;

    procedure p1{ ... };
    procedure p2{ ... };
    ...
    procedure pn{ ... };

    {
        initializing_code;
    }

}
```



TestAndSet:

- Test and Set Lock (TSL) is a synchronization mechanism.
- It uses a test and set instruction to provide the synchronization among the processes executing concurrently.

Function testandset(var i:integer):boolean;

Begin

If i=0 then

Begin

i=1;

Testandset=true

Critical section

End

Else

Testandset=false

End

Synchronization Hardware:

- To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data
- To present both software and hardware solutions of the critical-section problem
- To introduce the concept of an atomic transaction and describe mechanisms to ensure atomicity
- 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 and is decremented by the consumer after it consumes a buffer

Producer

```
while (true) {


        /*  produce an item and put in nextProduced  */
            while (count == BUFFER_SIZE)
                                    ; // do nothing
                        buffer [in] = nextProduced;
                        in = (in + 1) % BUFFER_SIZE;
                        count++;
}
```

Consumer

```
while (true)  {
            while (count == 0)
                            ; // do nothing
                        nextConsumed =  buffer[out];
                        out = (out + 1) % BUFFER_SIZE;
            count--;
```

/* consume the item in nextConsumed

Synchronization Hardware

A *lock* is one form of hardware support for mutual exclusion.
If a shared resource has a locked hardware lock, it is already in use by another process.
If it is not locked, a process may freely
Lock it for itself;
Use it;
Unlock it when it finishes.

```
do
{
        acquire lock
            critical section
        release lock
            remainder section
    //  remainder section
} while (TRUE);
```

Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

Bounded-Buffer Problem/producer-consumer problem

The Producer-Consumer problem is a classical multi-process synchronization problem, that is we are trying to achieve synchronization between more than one process.

There is one Producer in the producer-consumer problem, Producer is producing some items, whereas there is one Consumer that is consuming the items produced by the Producer. The same memory buffer is shared by both producers and consumers which is of fixed-size.

The task of the Producer is to produce the item, put it into the memory buffer, and again start producing items. Whereas the task of the Consumer is to consume the item from the memory buffer.

- The producer should produce data only when the buffer is not full. In case it is found that the buffer is full, the producer is not allowed to store any data into the memory buffer.
- Data can only be consumed by the consumer if and only if the memory buffer is not empty. In case it is found that the buffer is empty, the consumer is not allowed to use any data from the memory buffer.
- Accessing memory buffer should not be allowed to producer and consumer at the same time.

The structure of the producer process

```
While(true)
{
While(count==buffersize)
buffer[i]=producer;
i=i+1;
Count++;
}
```

The structure of the consumer process

```
While(true)
{
While(count==0)
Consumer=buffer[i];
i=i+1;
Count--;
}
```

Readers-Writers Problem A data set is shared among a number of concurrent processes

- Readers – only read the data set; they do **not** perform any updates
- Writers – can both read and writenProblem – allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time

- Shared Data
- Data set
- Semaphore mutex initialized to 1
- Semaphore wrt initialized to 1
- Integer readcount initialized to 0

The structure of a writer process

```
do {
wait (wrt) ;
                    //    writing is performed
signal (wrt) ;
        } while (TRUE);
```

The structure of a reader process

```
            do {
wait (mutex) ;
readcount ++ ;
if (readcount == 1)
            wait (wrt) ;
signal (mutex)
       // reading is performed
wait (mutex) ;
readcount  - - ;
if (readcount  == 0)
                                    signal (wrt) ;
signal (mutex) ;
           } while (TRUE);
```

Dining-Philosophers Problem

The Dining Philosopher Problem states that 5 philosophers are seated around a circular table with one chopstick between each pair of philosophers. There is one chopstick between each philosopher(4 chopsticks). A philosopher may eat if he can pick up the two chopsticks adjacent to him. One chopstick may be picked up by any one of its adjacent followers but not both.

1.The maximum number of philosophers at the table should not exceed four.

2.Allow even/odd number of philosophers at a time.

3.A philosopher should only be permitted to choose their chopsticks if both of the available

```
do  {
wait ( chopstick[i] );
wait ( chopStick[ (i + 1) % 5] );
//  eat
signal ( chopstick[i] );
signal (chopstick[ (i + 1) % 5] );
//  think
} while (TRUE);
```