OS MODULE 2

PROCESS MANAGEMENT

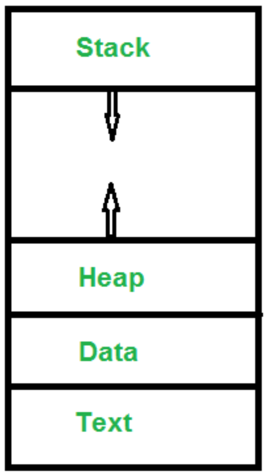
PROCESS MANAGEMENT

Process management is a key part of an operating system. It controls how processes are carried out, and controls how your computer runs by handling the active processes. This includes stopping processes, setting which processes should get more attention, and many more. You can manage processes on your own computer too.

The OS is responsible for managing the start, stop, and scheduling of processes, which are programs running on the system. The operating system uses a number of methods to prevent deadlocks, facilitate inter-process communication, and synchronize processes. Efficient resource allocation, conflict-free process execution, and optimal system performance are all guaranteed by competent process management. This essential component of an operating system enables the execution of numerous applications at once, enhancing system utilization and responsiveness.

PROCESS

A process in memory is divided into several distinct sections, each serving a different purpose. Here’s how a process typically looks in memory:



- **Text Section**: A Process, sometimes known as the Text Section, also includes the current activity represented by the value of the Program Counter.
- **Stack**: The stack contains temporary data, such as function parameters, returns addresses, and local variables.
- **Data Section**: Contains the global variable.
- **Heap Section**: Dynamically memory allocated_to process during its run time.

ADVANTAGES OF PROCESS MANAGEMENT

- **Running Multiple Programs:** Process management lets you run multiple applications at the same time, for example, listen to music while browsing the web.
- **Process Isolation:** It ensures that different programs don’t interfere with each other, so a problem in one program won’t crash another.

- **Fair Resource Use:** It makes sure resources like CPU time and memory are shared fairly among programs, so even lower-priority programs get a chance to run.
- **Smooth Switching:** It efficiently handles switching between programs, saving and loading their states quickly to keep the system responsive and minimize delays.

DISADVANTAGES OF PROCESS MANAGEMENT

- **Overhead:** Process management uses system resources because the OS needs to keep track of various data structures and scheduling queues. This requires CPU time and memory, which can affect the system's performance.
- **Complexity:** Designing and maintaining an OS is complicated due to the need for complex scheduling algorithms and resource allocation methods.
- **Deadlocks:** To keep processes running smoothly together, the OS uses mechanisms like semaphores and mutex locks. However, these can lead to deadlocks, where processes get stuck waiting for each other indefinitely.
- **Increased Context Switching:** In multitasking systems, the OS frequently switches between processes. Storing and loading the state of each process (context switching) takes time and computing power, which can slow down the system.

PROCESS CREATION

1. When a new process is created, the operating system assigns a unique Process Identifier (PID) to it and inserts a new entry in the primary process table.

2. Then required memory space for all the elements of the process such as program, data, and stack is allocated including space for its Process Control Block (PCB).

3. Next, the various values in PCB are initialized such as,

1. The process identification part is filled with PID assigned to it in step (1) and also its parent's PID.
2. The processor register values are mostly filled with zeroes, except for the stack pointer and program counter. The stack pointer is filled with the address of the stack-allocated to it in step (ii) and the program counter is filled with the address of its program entry point.
3. The process state information would be set to 'New'.
4. Priority would be lowest by default, but the user can specify any priority during creation.

4. Then the operating system will link this process to the scheduling queue and the process state would be changed from 'New' to 'Ready'. Now the process is competing for the CPU.

5. Additionally, the operating system will create some other data structures such as log files or accounting files to keep track of processes activity.

PROCESS DELETION

Processes are terminated by themselves when they finish executing their last statement, then operating system USES exit( ) system call to delete its context. Then all the resources held by that process like physical and virtual memory, 10 buffers, open files, etc., are taken back by the operating system. A process P can be terminated either by the operating system or by the parent process of P.

A parent may terminate a process due to one of the following reasons:

1. When task given to the child is not required now.
2. When the child has taken more resources than its limit.
3. The parent of the process is exiting, as a result, all its children are deleted. This is called cascaded termination.

TWO STATE MODEL

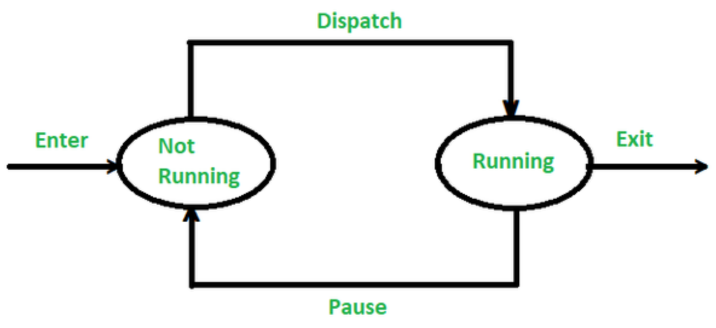
The simplest model in the process state will be a two-state model as it consists of only two states that are given below:

- **Running State-** A state in which the process is currently being executed.
- **Not Running State-** A state in which the process is waiting for execution.

Execution of Process in Two-state Model

A two-state can be created anytime no matter whether a process is being executed or not.

- Firstly, when the OS creates a new process, it also creates a process control block for the process so that the process can enter into the system in a non-running state. If any process exit/leaves the system, then it is known to the OS.
- Once in a while, the currently running process will be interrupted or break-in and the dispatcher (a program that switches the processor from one process to another) of the OS will run any other process.
- Now, the former process(interrupted process) moves from the running state to the non-running state and one of the other processes moves to the running state after which it exits from the system.



FIVE STATE MODEL

In five- state model the states have been split into two non-running states: *ready* and *blocked*, and along with this, two more states are added for the purpose: ***New*** and ***Exit/Terminate***. These two states are used because, in the previous models, the main memory is capable of storing all programs but that is not true because the programs are now very large, and loading those processes in the main memory is very tough/ or even not possible and also if there's a requirement for using the previous resources that are released by the process, then that is not possible here.

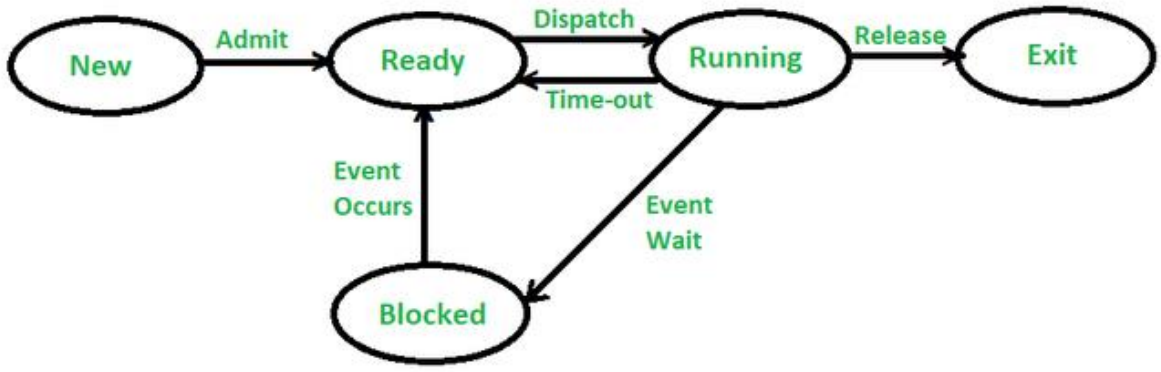
The five states that are being used in this process model are:

- **Running:** It means a process that is currently being executed. Assuming that there is only a single processor in the below execution process, so there will be at most one processor at a time that can be running in the state.
- **Ready:** It means a process that is prepared to execute when given the opportunity by the OS.
- **Blocked/Waiting:** It means that a process cannot continue executing until some event occurs like for example, the completion of an input-output operation.
- **New:** It means a new process that has been created but has not yet been admitted by the OS for its execution. A new process is not loaded into the main memory, but its process control block (PCB) has been created.
- **Exit/Terminate:** A process or job that has been released by the OS, either because it is completed or is aborted for some issue.

Execution of Process in Five-state Model

This model consists of five states i.e, running, ready, blocked, new, and exit. The model works when any new job/process occurs in the queue, it is first admitted in the queue after that it goes in the ready state. Now in the Ready state, the process goes in the running state. In the running state, a process has two conditions i.e., either the process goes to the event wait or the process gets a time-out.

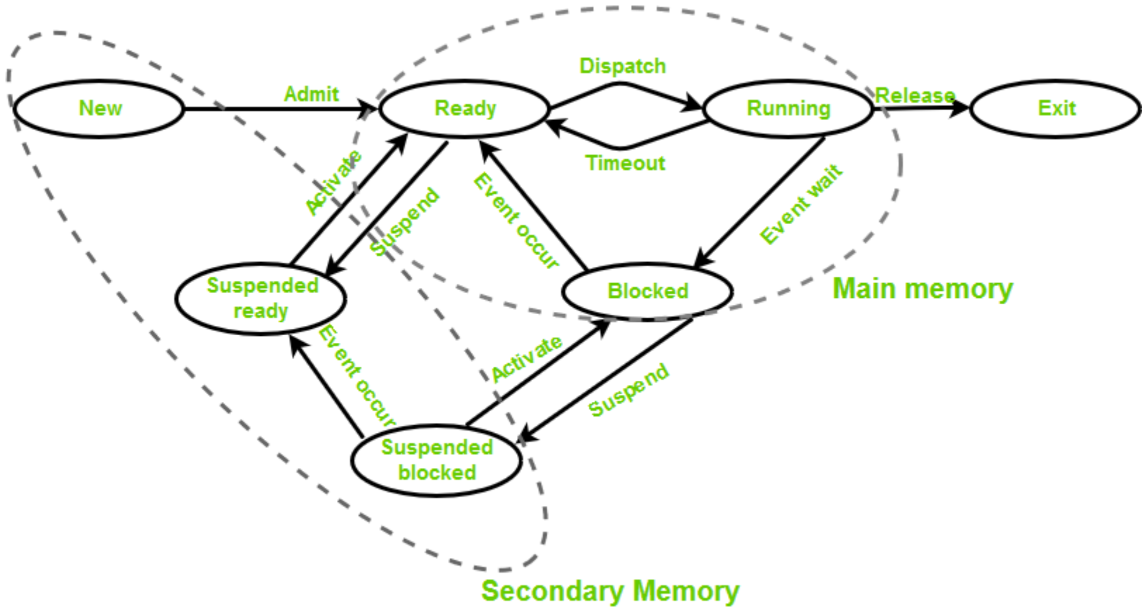
If the process has timed out, then the process again goes to the ready state as the process has not completed its execution. If a process has an event wait condition then the process goes to the blocked state and after that to the ready state. If both conditions are true, then the process goes to running state after dispatching after which the process gets released and at last it is terminated.



<u>SEVEN STATE MODEL</u>

The states of a process are as follows:

- **New State:** In this step, the process is about to be created but not yet created. It is the program that is present in secondary memory that will be picked up by the OS to create the process.
- **Ready State:** New -> Ready to run. After the creation of a process, the process enters the ready state i.e. the process is loaded into the main memory. The process here is ready to run and is waiting to get the CPU time for its execution. Processes that are ready for execution by the CPU are maintained in a queue called a ready queue for ready processes.
- **Run State:** The process is chosen from the ready queue by the OS for execution and the instructions within the process are executed by any one of the available CPU cores.
- **Blocked or Wait State:** Whenever the process requests access to I/O or needs input from the user or needs access to a critical region(the lock for which is already acquired) it enters the blocked or waits state. The process continues to wait in the main memory and does not require CPU. Once the I/O operation is completed the process goes to the ready state.
- **Terminated or Completed State:** Process is killed as well as PCB is deleted. The resources allocated to the process will be released or deallocated.
- **Suspend Ready:** Process that was initially in the ready state but was swapped out of main memory(refer to Virtual Memory topic) and placed onto external storage by the scheduler is said to be in suspend ready state. The process will transition back to a ready state whenever the process is again brought onto the main memory.
- **Suspend Wait or Suspend Blocked:** Similar to suspend ready but uses the process which was performing I/O operation and lack of main memory caused them to move to secondary memory. When work is finished it may go to suspend ready.



- **CPU and I/O Bound Processes:** If the process is intensive in terms of CPU operations, then it is called CPU bound process. Similarly, If the process is intensive in terms of I/O operations then it is called I/O bound process.

How Does a Process Move From One State to Other State?

A process can move between different states in an operating system based on its execution status and resource availability. Here are some examples of how a process can move between different states:

- **New to Ready:** When a process is created, it is in a new state. It moves to the ready state when the operating system has allocated resources to it and it is ready to be executed.
- **Ready to Running:** When the CPU becomes available, the operating system selects a process from the ready queue depending on various scheduling algorithms and moves it to the running state.
- **Running to Blocked:** When a process needs to wait for an event to occur (I/O operation or system call), it moves to the blocked state. For example, if a process needs to wait for user input, it moves to the blocked state until the user provides the input.
- **Running to Ready:** When a running process is preempted by the operating system, it moves to the ready state. For example, if a higher-priority process becomes ready, the operating system may preempt the running process and move it to the ready state.
- **Blocked to Ready:** When the event a blocked process was waiting for occurs, the process moves to the ready state. For example, if a process was waiting for user input and the input is provided, it moves to the ready state.
- **Running to Terminated:** When a process completes its execution or is terminated by the operating system, it moves to the terminated state.

Types of Schedulers

- **Long-Term Scheduler:** Decides how many processes should be made to stay in the ready state. This decides the degree of multiprogramming. Once a decision is taken it lasts for a long time which also indicates that it runs infrequently. Hence it is called a long-term scheduler.
- **Short-Term Scheduler:** Short-term scheduler will decide which process is to be executed next and then it will call the dispatcher. A dispatcher is a software that moves the process from ready to run and vice versa. In other words, it is context switching. It runs frequently. Short-term scheduler is also called CPU scheduler.
- **Medium Scheduler:** Suspension decision is taken by the medium-term scheduler. The medium-term scheduler is used for swapping which is moving the process from main memory to secondary and vice versa. The swapping is done to reduce degree of multiprogramming.
- **Preemption –** Process is forcefully removed from CPU. Pre-emption is also called time sharing or multitasking.
- **Non-Preemption –** Processes are not removed until they complete the execution. Once control is given to the CPU for a process execution, till the CPU releases the control by itself, control cannot be taken back forcibly from the CPU.

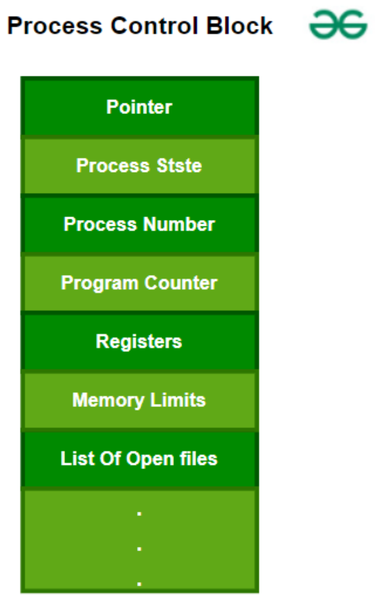
PROCESS CONTROL BLOCK

While creating a process, the operating system performs several operations. To identify the processes, it assigns a process identification number (PID) to each process. As the operating system supports multi-programming, it needs to keep track of all the processes. For this task, the process control block (PCB) is used to track the process's execution status. Each block of memory contains information about the process state, program counter, stack pointer, status of opened files, scheduling algorithms, etc.

All this information is required and must be saved when the process is switched from one state to another. When the process makes a transition from one state to another, the operating system must update information in the process's PCB. A process control block (PCB) contains information about the process, i.e. registers, quantum, priority, etc. The process table is an array of PCBs, which logically contains a PCB for all of the current processes in the system.

Structure of the Process Control Block

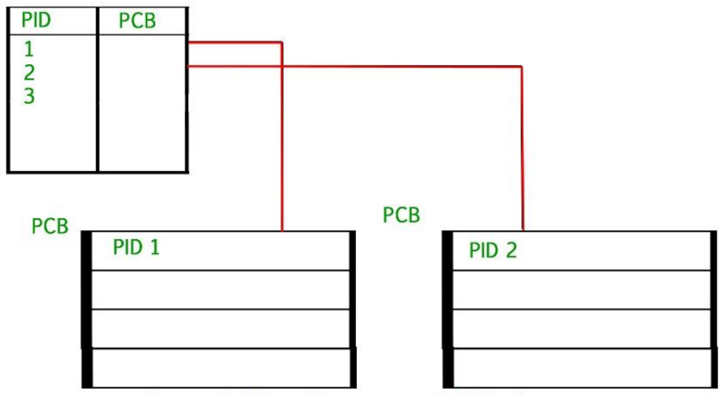
A Process Control Block (PCB) is a data structure used by the operating system to manage information about a process. The process control keeps track of many important pieces of information needed to manage processes efficiently. The diagram helps explain some of these key data items.



Process Control Block

- **Pointer:** It is a stack pointer that is required to be saved when the process is switched from one state to another to retain the current position of the process.
- **Process state:** It stores the respective state of the process.
- **Process number:** Every process is assigned a unique id known as process ID or PID which stores the process identifier.

- **Program counter:** Program Counter stores the counter, which contains the address of the next instruction that is to be executed for the process.
- **Register:** Registers in the PCB, it is a data structure. When a processes is running and it's time slice expires, the current value of process specific registers would be stored in the PCB and the process would be swapped out. When the process is scheduled to be run, the register values is read from the PCB and written to the CPU registers. This is the main purpose of the registers in the PCB.
- **Memory limits:** This field contains the information about memory management system used by the operating system. This may include page tables, segment tables, etc.
- **List of Open files:** This information includes the list of files opened for a process.



Process table and process control block

<u>MULTITHREADING MODELS</u>

Multi threading- It is a process of multiple threads executes at same time.

There are two main threading models in process management: user-level threads and kernel-level threads.

User-level threads: In this model, the operating system does not directly support threads. Instead, threads are managed by a user-level thread library, which is part of the application. The library manages the threads and schedules them on available processors. The advantages of user-level threads include greater flexibility and portability, as the application has more control over thread management. However, the disadvantage is that user-level threads are not as efficient as kernel-level threads, as they rely on the application to manage thread scheduling.

Kernel-level threads: In this model, the operating system directly supports threads as part of the kernel. Each thread is a separate entity that can be scheduled and executed independently by the operating system. The advantages of kernel-level threads include better performance and scalability, as the operating system can schedule threads more efficiently. However, the disadvantage is that kernel-level threads are less flexible and portable than user-level threads, as they are managed by the operating system.

There are also hybrid models that combine elements of both user-level and kernel-level threads. For example, some operating systems use a hybrid model called the "two-level model", where each process has one or more user-level threads, which are mapped to kernel-level threads by the operating system.

Overall, the choice of threading model depends on the requirements of the application and the capabilities of the underlying operating system.

Here are some advantages and disadvantages of each threading model:

User-level threads:

Advantages:

Greater flexibility and control: User-level threads provide more control over thread management, as the thread library is part of the application. This allows for more customization and control over thread scheduling.

Portability: User-level threads can be more easily ported to different operating systems, as the thread library is part of the application.

Disadvantages:

Lower performance: User-level threads rely on the application to manage thread scheduling, which can be less efficient than kernel-level thread scheduling. This can result in lower performance for multithreaded applications.

Limited parallelism: User-level threads are limited to a single processor, as the application has no control over thread scheduling on other processors.

Kernel-level threads:

Advantages:

Better performance: Kernel-level threads are managed by the operating system, which can schedule threads more efficiently. This can result in better performance for multithreaded applications.

Greater parallelism: Kernel-level threads can be scheduled on multiple processors, which allows for greater parallelism and better use of available resources.

Disadvantages:

Less flexibility and control: Kernel-level threads are managed by the operating system, which provides less flexibility and control over thread management compared to user-level threads.

Less portability: Kernel-level threads are more tightly coupled to the operating system, which can make them less portable to different operating systems.

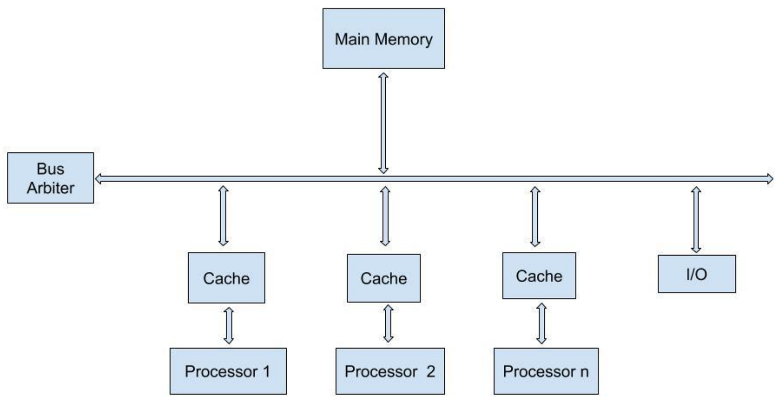
Parameters	**User Level Thread**	**Kernel Level Thread**
Implemented by	User threads are implemented by user-level libraries.	Kernel threads are implemented by Operating System (OS).

Parameters	User Level Thread	Kernel Level Thread
Recognize	The operating System doesn't recognize user-level threads directly.	Kernel threads are recognized by Operating System.
Implementation	Implementation of User threads is easy.	Implementation of Kernel-Level thread is complicated.
Context switch time	Context switch time is less.	Context switch time is more.
Hardware support	No hardware support is required for context switching.	Hardware support is needed.
Blocking operation	If one user-level thread performs a blocking operation then the entire process will be blocked.	If one kernel thread performs a blocking operation then another thread can continue execution.
Multithreading	Multithreaded applications cannot take full advantage of multiprocessing.	Kernels can be multithreaded.
Creation and Management	User-level threads can be created and managed more quickly.	Kernel-level level threads take more time to create and manage.
Operating System	Any operating system can support user-level threads.	Kernel-level threads are operating system-specific.
Thread Management	Managed by a thread library at the user level.	The application code doesn't contain thread management code; it's an API to the kernel mode.

Parameters	User Level Thread	Kernel Level Thread
Example	POSIX threads, Mach C-Threads.	Java threads, POSIX threads on Linux.
Advantages	Simple and quick to create, more portable, does not require kernel mode privileges for context switching.	Allows for true parallelism, multithreading in kernel routines, and can continue execution if one thread is blocked.
Disadvantages	Cannot fully utilize multiprocessing, entire process blocked if one thread blocks.	Requires more time to create/manage, involves mode switching to kernel mode.
Memory management	In user-level threads, each thread has its own stack, but they share the same address space.	In kernel-level threads have their own stacks and their own separate address spaces, so they are better isolated from each other.
Fault tolerance	User-level threads are less fault-tolerant than kernel-level threads. If a user-level thread crashes, it can bring down the entire process.	Kernel-level threads can be managed independently, so if one thread crashes, it doesn’t necessarily affect the others.
Resource utilization	Limited access to system resources, cannot directly perform I/O operations.	It can access to the system-level features like I/O operations.
Portability	User-level threads are more portable than kernel-level threads.	Less portable due to dependence on OS-specific kernel implementations.

SYMMETRIC MULTIPROCESSING

SMP i.e. symmetric multiprocessing, refers to the computer architecture where multiple identical processors are interconnected to a single shared main memory, with full accessibility to all the I/O devices, unlike asymmetric MP. In other words, all the processors have common shared(common) memory and same data path or I/O bus as shown in the figure.



Main memory and data bus or I/O bus being shared among multiple processors in SMP

Characteristics of SMP

- **Identical**: All the processors are treated equally i.e. all are identical.
- **Communication**: Shared memory is the mode of communication among processors.
- **Complexity**: Are complex in design, as all units share same memory and data bus.
- **Expensive**: They are costlier in nature.
- Unlike asymmetric where a task is done only by Master processor, here tasks of the operating system are handled individually by processors.

Applications

This concept finds its application in **parallel processing**, where time-sharing systems(TSS) have assigned tasks to different processors running in parallel to each other, also in TSS that uses **multithreading** i.e. multiple threads running simultaneously.

Advantages

- **Throughput:** Since tasks can be run by all the processors unlike in asymmetric, hence increased degree of throughput(processes executed in unit time).
- **Reliability:** Failing a processor doesn't fail whole system, as all are equally capable processors, though throughput do fail a little.

Disadvantages

- **Complex design:** Since all the processors are treated equally by OS, so designing and management of such OS become difficult.
- **Costlier:** As all the processors share the common main memory, on account of which size of memory required is larger implying more expensive.

SCHEDULING CRITERIA

CPU Scheduling has several criteria. Some of them are mentioned below.

1. CPU utilization

The main objective of any CPU scheduling algorithm is to keep the CPU as busy as possible. Theoretically, CPU utilization can range from 0 to 100 but in a real-time system, it varies from 40 to 90 percent depending on the load upon the system.

2. Throughput

A measure of the work done by the CPU is the number of processes being executed and completed per unit of time. This is called throughput. The throughput may vary depending on the length or duration of the processes.

3. Turnaround Time

For a particular process, an important criterion is how long it takes to execute that process. The time elapsed from the time of submission of a process to the time of completion is known as the turnaround time. Turn-around time is the sum of times spent waiting to get into memory, waiting in the ready queue, executing in CPU, and waiting for I/O.

Turn Around Time = Completion Time – Arrival Time.

4. Waiting Time

A scheduling algorithm does not affect the time required to complete the process once it starts execution. It only affects the waiting time of a process i.e. time spent by a process waiting in the ready queue.

Waiting Time = Turnaround Time – Burst Time.

5. Response Time

In an interactive system, turn-around time is not the best criterion. A process may produce some output fairly early and continue computing new results while previous results are being output to the user. Thus another criterion is the time taken from submission of the process of the request until the first response is produced. This measure is called response time.

Response Time = CPU Allocation Time(when the CPU was allocated for the first) – Arrival Time

6. Completion Time

The completion time is the time when the process stops executing, which means that the process has completed its burst time and is completely executed.

7. Priority

If the operating system assigns priorities to processes, the scheduling mechanism should favor the higher-priority processes.

8. Predictability

A given process always should run in about the same amount of time under a similar system load.

MULTI PROCESSOR SCHEDULING

Multiple processor scheduling or multiprocessor scheduling focuses on designing the system's scheduling function, which consists of more than one processor. Multiple CPUs share the load (load sharing) in multiprocessor scheduling so that various processes run simultaneously. In general, multiprocessor scheduling is complex as compared to single processor scheduling. In the multiprocessor scheduling, there are many processors, and they are identical, and we can run any process at any time.

The multiple CPUs in the system are in close communication, which shares a common bus, memory, and other peripheral devices. So we can say that the system is tightly coupled. These systems are used when we want to process a bulk amount of data, and these systems are mainly used in satellite, weather forecasting, etc.

There are cases when the processors are identical, i.e., homogenous, in terms of their functionality in multiple-processor scheduling. We can use any processor available to run any process in the queue.

Multiprocessor systems may be ***heterogeneous*** (different kinds of CPUs) or ***homogenous*** (the same CPU). There may be special scheduling constraints, such as devices connected via a private bus to only one CPU.

Approaches to Multiple Processor Scheduling

There are two approaches to multiple processor scheduling in the operating system: Symmetric Multiprocessing and Asymmetric Multiprocessing.

1. **Symmetric Multiprocessing:** It is used where each processor is ***self-scheduling***. All processes may be in a common ready queue, or each processor may have its private queue for ready processes. The scheduling proceeds further by having the scheduler for each processor examine the ready queue and select a process to execute.
2. **Asymmetric Multiprocessing:** It is used when all the scheduling decisions and I/O processing are handled by a single processor called the ***Master Server***. The other processors execute only the ***user code***. This is simple and reduces the need for data sharing, and this entire scenario is called Asymmetric Multiprocessing.

Processor Affinity

Processor Affinity means a process has an ***affinity*** for the processor on which it is currently running. When a process runs on a specific processor, there are certain effects on the cache memory. The data most recently accessed by the process populate the cache for the processor. As a result, successive memory access by the process is often satisfied in the cache memory.

Now, suppose the process migrates to another processor. In that case, the contents of the cache memory must be invalidated for the first processor, and the cache for the second processor must be repopulated. Because of the high cost of invalidating and repopulating caches, most SMP(symmetric multiprocessing) systems try to avoid migrating processes from one processor to another and keep a process running on the same processor. This is known as processor affinity. There are two types of processor affinity, such as:

1. **Soft Affinity:** When an operating system has a policy of keeping a process running on the same processor but not guaranteeing it will do so, this situation is called soft affinity.
2. **Hard Affinity:** Hard Affinity allows a process to specify a subset of processors on which it may run. Some Linux systems implement soft affinity and provide system calls like ***sched_setaffinity()*** that also support hard affinity.

Load Balancing

Load Balancing is the phenomenon that keeps the workload evenly distributed across all processors in an SMP system. Load balancing is necessary only on systems where each processor has its own private queue of a process that is eligible to execute.

Load balancing is unnecessary because it immediately extracts a runnable process from the common run queue once a processor becomes idle. On SMP (symmetric multiprocessing), it is important to keep the workload balanced among all processors to utilize the benefits of having more than one processor fully. One or more processors will sit idle while other processors have high workloads along with lists of processors awaiting the CPU. There are two general approaches to load balancing:

1. **Push Migration:** In push migration, a task routinely checks the load on each processor. If it finds an imbalance, it evenly distributes the load on each processor by moving the processes from overloaded to idle or less busy processors.
2. **Pull Migration:**Pull Migration occurs when an idle processor pulls a waiting task from a busy processor for its execution.

Multi-core Processors

In multi-core processors, multiple processor cores are placed on the same physical chip. Each core has a register set to maintain its architectural state and thus appears to the operating system as a separate physical processor. ***SMP systems*** that use multi-core processors are faster and consume less power than systems in which each processor has its own physical chip.

However, multi-core processors may complicate the scheduling problems. When the processor accesses memory, it spends a significant amount of time waiting for the data to become available. This situation is called a ***Memory stall***. It occurs for various reasons, such as cache miss, which is accessing the data that is not in the cache memory.

In such cases, the processor can spend upto 50% of its time waiting for data to become available from memory. To solve this problem, recent hardware designs have implemented multithreaded processor cores in which two or more hardware threads are assigned to each core. Therefore if one thread stalls while waiting for the memory, the core can switch to another thread. There are two ways to multithread a processor:

1. **Coarse-Grained Multithreading:** A thread executes on a processor until a long latency event such as a memory stall occurs in coarse-grained multithreading. Because of the delay caused by the long latency event, the processor must switch to another thread to begin execution.

The cost of switching between threads is high as the instruction pipeline must be terminated before the other thread can begin execution on the processor core. Once this new thread begins execution, it begins filling the pipeline with its instructions.

2. **Fine-Grained Multithreading:** This multithreading switches between threads at a much finer level, mainly at the boundary of an instruction cycle. The architectural design of fine-grained systems includes logic for thread switching, and as a result, the cost of switching between threads is small.

Symmetric Multiprocessor

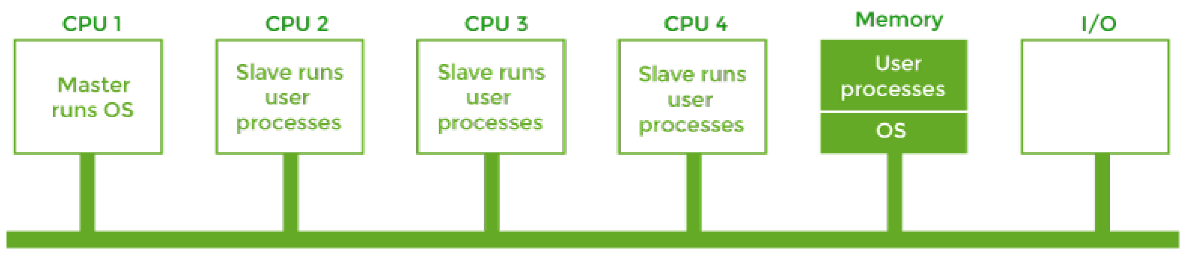
Symmetric Multiprocessors (SMP) is the third model. There is one copy of the OS in memory in this model, but any central processing unit can run it. Now, when a system call is made, the central processing unit on which the system call was made traps the kernel and processed that system call. This model balances processes and memory dynamically. This approach uses Symmetric Multiprocessing, where each processor is self-scheduling.

The scheduling proceeds further by having the scheduler for each processor examine the ready queue and select a process to execute. In this system, this is possible that all the process may be in a common ready queue or each processor may have its private queue for the ready process. There are mainly three sources of contention that can be found in a multiprocessor operating system.

- **Locking system:** As we know that the resources are shared in the multiprocessor system, there is a need to protect these resources for safe access among the multiple processors. The main purpose of the locking scheme is to serialize access of the resources by the multiple processors.
- **Shared data:** When the multiple processors access the same data at the same time, then there may be a chance of inconsistency of data, so to protect this, we have to use some protocols or locking schemes.
- **Cache coherence:** It is the shared resource data that is stored in multiple local caches. Suppose two clients have a cached copy of memory and one client change the memory block. The other client could be left with an invalid cache without notification of the change, so this conflict can be resolved by maintaining a coherent view of the data.

Master-Slave Multiprocessor

In this multiprocessor model, there is a single data structure that keeps track of the ready processes. In this model, one central processing unit works as a master and another as a slave. All the processors are handled by a single processor, which is called the master server.


CPU 1
Master runs OS
CPU 2
Slave runs user processes
CPU 3
Slave runs user processes
CPU 4
Slave runs user processes
Memory
User processes
OS
I/O

The master server runs the operating system process, and the slave server runs the user processes. The memory and input-output devices are shared among all the processors, and all the processors are connected to a common bus. This system is simple and reduces data sharing, so this system is called ***Asymmetric multiprocessing***.

Virtualization and Threading

In this type of ***multiple processor*** scheduling, even a single CPU system acts as a multiple processor system. In a system with virtualization, the virtualization presents one or more virtual CPUs to each of the virtual machines running on the system. It then schedules the use of physical CPUs among the virtual machines.

- Most virtualized environments have one host operating system and many guest operating systems, and the host operating system creates and manages the virtual machines.
- Each virtual machine has a guest operating system installed, and applications run within that guest.
- Each guest operating system may be assigned for specific use cases, applications, or users, including time-sharing or real-time operation.
- Any guest operating-system scheduling algorithm that assumes a certain amount of progress in a given amount of time will be negatively impacted by the virtualization.
- A time-sharing operating system tries to allot 100 milliseconds to each time slice to give users a reasonable response time. A given 100 millisecond time slice may take much more than 100 milliseconds of virtual CPU time. Depending on how busy the system is, the time slice may take a second or more, which results in a very poor response time for users logged into that virtual machine.
- The net effect of such scheduling layering is that individual virtualized operating systems receive only a portion of the available CPU cycles, even though they believe they are receiving all cycles and scheduling all of those cycles. The time-of-day clocks in virtual machines are often incorrect because timers take no longer to trigger than they would on dedicated CPUs.
- Virtualizations can thus undo the good scheduling algorithm efforts of the operating systems within virtual machines.

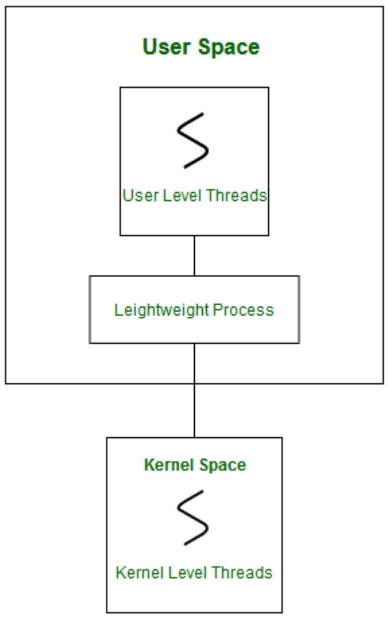
<u>THREAD SCHEDULING</u>

Scheduling of threads involves two boundary scheduling.

1. Scheduling of user-level threads (ULT) to kernel-level threads (KLT) via lightweight process (LWP) by the application developer.
2. Scheduling of kernel-level threads by the system scheduler to perform different unique OS functions.

Lightweight Process (LWP)

Light-weight process are threads in the user space that acts as an interface for the ULT to access the physical CPU resources. Thread library schedules which thread of a process to run on which LWP and how long. The number of LWPs created by the thread library depends on the type of application. In the case of an I/O bound application, the number of LWPs depends on the number of user-level threads. This is because when an LWP is blocked on an I/O operation, then to invoke the other ULT the thread library needs to create and schedule another LWP. Thus, in an I/O bound application, the number of LWP is equal to the number of the ULT. In the case of a CPU-bound application, it depends only on the application. Each LWP is attached to a separate kernel-level thread.



In real-time, the first boundary of thread scheduling is beyond specifying the scheduling policy and the priority. It requires two controls to be specified for the User level threads: Contention scope, and Allocation domain. These are explained as following below.

Contention Scope

The word contention here refers to the competition or fight among the User level threads to access the kernel resources. Thus, this control defines the extent to which contention takes place. It is defined by the application developer using the thread library.

Depending upon the extent of contention it is classified as-

- **Process Contention Scope (PCS) :**
  The contention takes place among threads **within a same process**. The thread library schedules the high-prioritized PCS thread to access the resources via available LWPs (priority as specified by the application developer during thread creation).

- **System Contention Scope (SCS) :**
  The contention takes place among **all threads in the system**. In this case, every SCS thread is associated to each LWP by the thread library and are scheduled by the system scheduler to access the kernel resources.

In LINUX and UNIX operating systems, the POSIX Pthread library provides a function *Pthread_attr_setscope* to define the type of contention scope for a thread during its creation.

int Pthread_attr_setscope(pthread_attr_t *attr, int scope)

The first parameter denotes to which thread within the process the scope is defined.
The second parameter defines the scope of contention for the thread pointed. It takes two values.

PTHREAD_SCOPE_SYSTEM

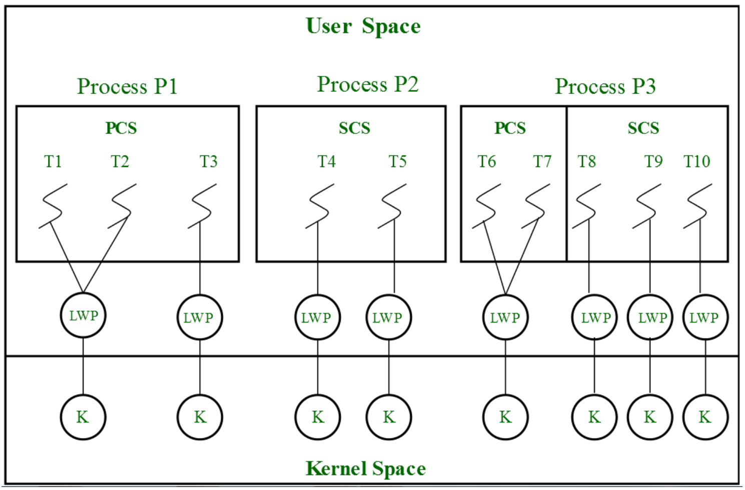
PTHREAD_SCOPE_PROCESS

If the scope value specified is not supported by the system, then the function returns *ENOTSUP*.

Allocation Domain

The allocation domain is **a set of one or more resources** for which a thread is competing. In a multicore system, there may be one or more allocation domains where each consists of one or more cores. One ULT can be a part of one or more allocation domain. Due to this high complexity in dealing with hardware and software architectural interfaces, this control is not specified. But by default, the multicore system will have an interface that affects the allocation domain of a thread.

Consider a scenario, an operating system with three process P1, P2, P3 and 10 user level threads (T1 to T10) with a single allocation domain. 100% of CPU resources will be distributed among all the three processes. The amount of CPU resources allocated to each process and to each thread depends on the contention scope, scheduling policy and priority of each thread defined by the application developer using thread library and also depends on the system scheduler. These User level threads are of a different contention scope.



In this case, the contention for allocation domain takes place as follows:

Process P1

All PCS threads T1, T2, T3 of Process P1 will compete among themselves. The PCS threads of the same process can share one or more LWP. T1 and T2 share an LWP and T3 are allocated to a separate LWP. Between T1 and T2 allocation of kernel resources via LWP is based on preemptive priority scheduling by the thread library. A Thread with a high priority will preempt low priority threads. Whereas, thread T1 of process p1 cannot preempt thread T3 of process p3 even if the priority of T1 is greater than the priority of T3. If the priority is equal, then the allocation of ULT to available LWPs is based on the scheduling policy of threads by the system scheduler(not by thread library, in this case).

Process P2

Both SCS threads T4 and T5 of process P2 will compete with processes P1 as a whole and with SCS threads T8, T9, T10 of process P3. The system scheduler will schedule the kernel resources among P1, T4, T5, T8, T9, T10, and PCS threads (T6, T7) of process P3 considering each as a separate process. Here, the Thread library has no control of scheduling the ULT to the kernel resources.

Process P3

Combination of PCS and SCS threads. Consider if the system scheduler allocates 50% of CPU resources to process P3, then 25% of resources is for process scoped threads and the remaining 25% for system scoped threads. The PCS threads T6 and T7 will be allocated to access the 25% resources based on the priority by the thread library. The SCS threads T8, T9, T10 will divide the 25% resources among themselves and access the kernel resources via separate LWP and KLT. The SCS scheduling is by the system scheduler.

SCHEDULING IN REAL TIME SYSTEMS

Real-time systems are systems that carry real-time tasks. These tasks need to be performed immediately with a certain degree of urgency. In particular, these tasks are related to control of certain events (or) reacting to them. Real-time tasks can be classified as hard real-time tasks and soft real-time tasks.

A hard real-time task must be performed at a specified time which could otherwise lead to huge losses. In soft real-time tasks, a specified deadline can be missed. This is because the task can be rescheduled (or) can be completed after the specified time,

In real-time systems, the scheduler is considered as the most important component which is typically a short-term task scheduler. The main focus of this scheduler is to reduce the response time associated with each of the associated processes instead of handling the deadline.

If a preemptive scheduler is used, the real-time task needs to wait until its corresponding tasks time slice completes. In the case of a non-preemptive scheduler, even if the highest priority is allocated to the task, it needs to wait until the completion of the current task. This task can be slow (or) of the lower priority and can lead to a longer wait.

A better approach is designed by combining both preemptive and non-preemptive scheduling. This can be done by introducing time-based interrupts in priority based systems which means the currently running process is interrupted on a time-based interval and if a higher priority process is present in a ready queue, it is executed by preempting the current process.

Based on schedulability, implementation (static or dynamic), and the result (self or dependent) of analysis, the scheduling algorithm are classified as follows.

1. **Static table-driven approaches:**
   These algorithms usually perform a static analysis associated with scheduling and capture the schedules that are advantageous. This helps in providing a schedule that can point out a task with which the execution must be started at run time.

2. **Static priority-driven preemptive approaches:**
   Similar to the first approach, these type of algorithms also uses static analysis of scheduling. The difference is that instead of selecting a particular schedule, it provides a useful way of assigning priorities among various tasks in preemptive scheduling.

3. **Dynamic planning-based approaches:**
   Here, the feasible schedules are identified dynamically (at run time). It carries a certain fixed time interval and a process is executed if and only if satisfies the time constraint.

4. **Dynamic best effort approaches:**
   These types of approaches consider deadlines instead of feasible schedules. Therefore the task is aborted if its deadline is reached. This approach is used widely is most of the real-time systems.

PROCESS SECURITY

Measures to prevent a person from illegally using resources in a computer system, or interfering with them in any manner. These measures ensure that data and programs are used only by authorized users and only in a desired manner, and that they are neither modified nor denied to authorized users. Security measures deal with threats to resources that come from outside a computer system, while protection measures deal with internal threats. Passwords are the principal security tool.

A password requirement thwarts attempts by unauthorized persons to masquerade as legitimate users of a system. The confidentiality of passwords is upheld by encryption. Computer users need to share data and programs stored in files with collaborators, and here is where an operating system's protection measures come in.

The owner of a file informs the OS of the specific access privileges other users are to have—whether and how others may access the file. The operating system's protection function then ensures that all accesses to the file are strictly in accordance with the specified access privileges. We begin by discussing how different kinds of security breaches are carried out: Trojan horses, viruses, worms, and buffer overflows. Their description is followed by a discussion of encryption techniques. We then describe three popular protection structures called access control lists, capability lists, and protection domains, and examine the degree of control provided by them over sharing of files. In the end, we discuss how security classifications of computer systems reflect the degree to which a system can withstand security and protection threats

Security measures guard a user's data and programs against interference from persons or programs outside the operating system; we broadly refer to such persons and their programs as nonusers.

Threats

Malware

Malware is short for malicious software and refers to any software that is designed to cause harm to computer systems, networks, or users. Malware can take many forms. Malware is a program designed to gain access to computer systems, generally for the benefit of some third party, without the user's permission.

Network Intrusion

A system called an intrusion detection system (IDS) observes network traffic for malicious transactions and sends immediate alerts when it is observed. It is software that checks a network or system for malicious activities or policy violations. Each illegal activity or violation is often recorded either centrally using a SIEM system or notified to an administration.

Buffer Overflow Technique

The buffer overflow technique can be employed to force a server program to execute an intruder-supplied code to breach the host computer system's security. It has been used to a devastating effect in mail servers and other Web servers. The basic idea in this technique is simple. Most systems contain a fundamental vulnerability—some programs do not validate the lengths of inputs they receive from users or other programs.

Because of this vulnerability, a buffer area in which such input is received may overflow and overwrite contents of adjoining areas of memory. On hardware platforms that use stacks that grow downward in memory e.g., the Intel 80×86 architecture, such overflows provide an opportunity to execute a piece of code that is disguised as data put in the buffer. This code could launch a variety of security attacks

Types of Threats

Below are tow types of threats.

1. Program threats

Below are some program threats.

- **Virus:** A virus is a malicious executable code attached to another executable file. The virus spreads when an infected file is passed from system to system. Viruses can be harmless or they can modify or delete data. Opening a file can trigger a virus.
- **Trojan Horse:** A Trojan horse is malware that carries out malicious operations under the appearance of a desired operation such as playing an online game.
- **Logic Bomb:** A logic bomb is a malicious program that uses a trigger to activate the malicious code. The logic bomb remains non-functioning until that trigger event happens.

2. System Threats

Below are some system threats.

- **Worm:** Worms replicate themselves on the system, attaching themselves to different files and looking for pathways between computers, such as computer network that shares common file storage areas.
- **Denial of Service:** Denial of Service (DoS) is a cyber-attack on an individual Computer or Website with the intent to deny services to intended users. Their purpose is to disrupt an organization's network operations by denying access to its users.

How to Ensure Process Security?

- **Authorization:** It means verification of access to the system resources. Intruders may guess or steal password and use it. Intruder may use a vendor-supplied password, which is expected to use by system administrator. It may find password by trial and error method. If the user logs on and goes for a break then the intruder may use the terminal. An intruder can write a dummy login program to fool user and that program collects information for its use later on.

- **Authentication:** Authentication is verification of a user's identity. Operating systems most often perform authentication by knowledge. That is, a person claiming to be some user X is called upon to exhibit some knowledge shared only between the OS and user X, such as a password
- **Browsing:** Files are very permissive so one can easily browse system files. Due to that it may access database and confidential information can be read.
- **Trap doors:** Sometimes Software designers want to modify their programs after installation. for that there are some secret entry points which programmers keep and it does not require and permission . These are called trap doors. Intrudes can use these trap doors.
- **Invalid Parameters:** Due to invalid parameters some security violation can take place.
- **Line Tapping:** Tapings in the communication line can access or modify confidential data.
- **Electronic data capture:** Using wiretaps or mechanism to pick up screen radiation and recognize what is displayed on screen is termed electronic data capture.
- **Lost Line:** In networking, the line way gets lost. In such case some o/s log out and allow access only after correct identify of user. some o/s cannot do this. So process will be floating and allow intruder to access data.
- **Improper Access Controls:** Some administrators may not plan about all rights. So some users may have more access and some users have very less access.
- **Waste Recovery:** If the block is deleted its information will be as it is. until it is allocated to another file. Intruder may use some mechanism to scan these blocks.
- **Rogue Software:** Programs are written to create mischief .