**Chapter 3**

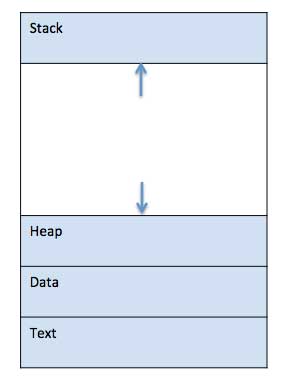
**Process Management**

**Define process**.

A process is a program in execution. Process is also called as job, task or unit of work. The execution of a process must progress in a sequential fashion. Process is an active entity.

To put it in simple terms, we write our computer programs in a text file and when we execute this program, it becomes a process which performs all the tasks mentioned in the program.

When a program is loaded into the memory and it becomes a process, it can be divided into four sections ─ stack, heap, text and data. The following image shows a simplified layout of a process inside main memory



Process In Memory

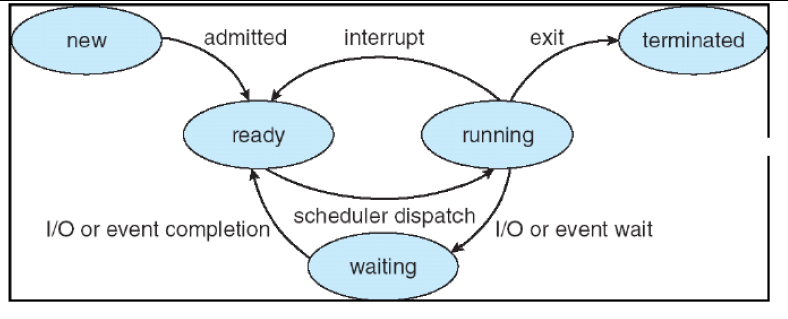
**Process memory** is divided into four sections for efficient working :

* The **Text section** is made up of the compiled program code, read in from non-volatile storage when the program is launched.
* The **Data section** is made up the global and static variables, allocated and initialized prior to executing the main.
* The **Heap** is used for the 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.

**Difference between program and process.**

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Process** | **Program** |
| Definition | An executing part of a program is called a process. | A program is a group of ordered operations to achieve a programming goal. |
| Nature | The process is an instance of the program being executing. | The nature of the program is passive, so it's unlikely to do to anything until it gets executed. |
| Resource management | The resource requirement is quite high in case of a process. | The program only needs memory for storage. |
| Overheads | Processes have considerable overhead. | No significant overhead cost. |
| Lifespan | The process has a shorter and very limited lifespan as it gets terminated after the completion of the task. | A program has a longer lifespan as it is stored in the memory until it is not manually deleted. |
| Creation | New processes require duplication of the parent process. | No such duplication is needed. |
| Required Process | Process holds resources like CPU, memory address, disk, I/O, etc. | The program is stored on disk in some file and does not require any other resources. |
| Entity type | A process is a dynamic or active entity. | A program is a passive or static entity. |
| Contain | A process contains many resources like a memory address, disk, printer, etc. | A program needs memory space on disk to store all instructions. |

**Draw process state diagram and describe each state.**

****

**New:** The process being created is available in the new state. It is the new state because the system is not

permitted it to enter the ready state due to limited memory available in the ready queue. If some memory

becomes available, then the process from the new state will go to ready state.

**Ready State:** The process which is not waiting for any external event such as I/O operation and which is

not running is said to be in ready state. It is not in the running state because some other process is already

running. It is waiting for its turn to go to the running state.

**Running State:** The process which is currently running and has control of the CPU is known as the

process in running state. In single user system, there is only one process which is in the running state. In

multiuser system, there are multiple processes which are in the running state.

**Blocked State:** The process that is currently waiting for external event such as an I/O operation is said to

be in blocked state. After the completion of I/O operation, the process from blocked state enters in the

ready state and from the ready state when the process turn will come it will again go to running state.

**Terminated State:** The process whose operation is completed, it will go the terminated state from the

running state. In halted state, the memory occupied by the process is released.

**Draw and explain process control block in detail**

Each process is represented as a process control block (PCB) in the operating system. It contains

Information associated with specific process.

In general, a PCB may contain information regarding:

1. **Process Number:** Each process is identified by its process number, called process identification

number (PID).

2. **Priority:** Each process is assigned a certain level of priority that corresponds to the relative importance of the event that it services.

3. **Process State:** This information is about the current state of the process. I.e. whether process is in

new, ready, running, waiting or terminated state.

4. **Program Counter:** This contains the address of the next instruction to be executed for this process.

5. **CPU Registers:** CPU registers vary in number and type, depending upon the computer architectures. These include index registers, stack pointers and general purpose registers etc. When an interrupt occurred, information about the current status of the old process is saved in registers along with the program counters. This information is necessary to allow the process to be continued correctly after the completion of an interrupted process.

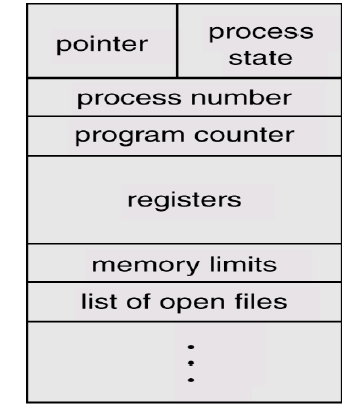
6. **CPU Scheduling Information:** This information includes a process priority, pointers to scheduling queues and any other scheduling parameters.

7. **Memory Management Information:** This information may include such information as the value of base and limit registers, the page table or the segment table depending upon the memory system used by operating system.

8. **Accounting:** This includes actual CPU time used in executing a process in order to charge individual user for processor time.

9. **I/O Status:** It includes outstanding I/O request, allocated devices information, pending operation and so on.

10. **File Management:** It includes information about all open files, access rights etc.

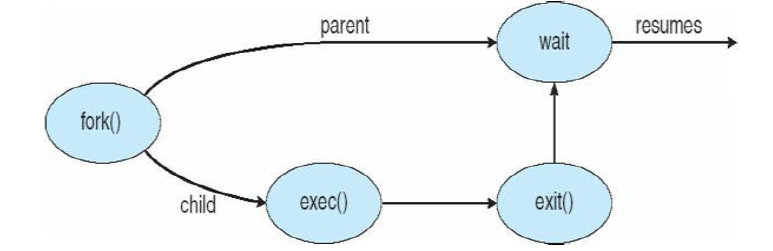


**Why is process creation necessary? State the role of fork process in the context**

Process creation: When a new process is to be added to those currently being managed, the operating system builds the data structures that are used to manage the process and allocates address space in main memory to the process. This is the creation of a new process. Create Process Operating system creates a new process with the specified or default attributes and identifier. A process may create several new sub processes.

**Syntax for creating new process is: CREATE (processed, attributes)**

Process creation in UNIX and Linux are done through fork() system calls. When the operating system creates a process at the explicit request of another process, the action is referred to as process spawning. When one process spawns another, the former is referred to as the parent process, and the spawned process is referred to as the child process. The parent may have to partition its resources among its children, or it may be able to share some resources among several of its children. A sub-process may be able to obtain its resources directly from the operating system. exec system call used after a fork to replace the process’ memory space with a new program



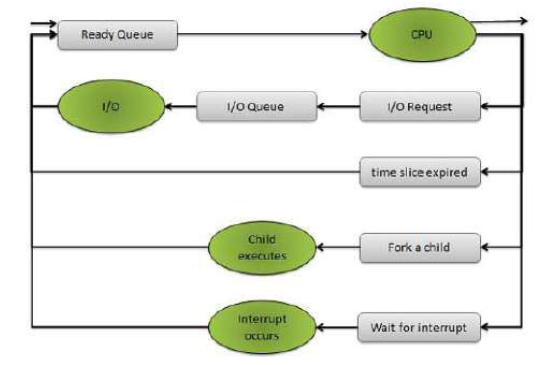
**Process Termination in operating systems**

Process termination is a technique in which a process is terminated and release the CPU after completing the execution.

Most of the OS use exit( ) system call to terminate a process. Main causes of process termination Normal Completion: The process completes all task and releases the CPU.

**Scheduling Queues**

Scheduling queues refers to queues of processes or devices. When the process enters into the system, then this process is put into a job queue. This queue consists of all processes in the system. The operating system also maintains other queues such as device queue

****

**The above figure shows the queuing diagram of process scheduling.**

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. A process migrates between the various scheduling queues throughout its life time. The operating system must select, for scheduling purposes, processes from these queues in some fashion. The selection process is carried out by the appropriate scheduler. Scheduler is the system program which schedules processes from the scheduling queues. Their main task is to select the jobs to be submitted into the system and to decide which process to run.

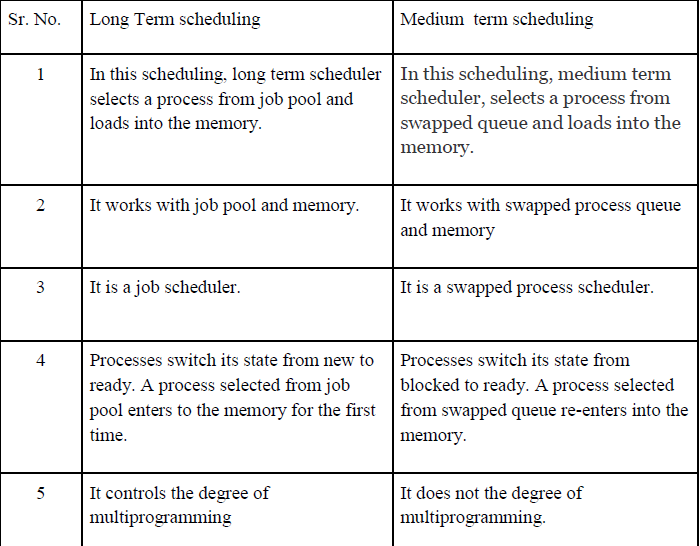
**Schedulers are of three types:**

Long Term Scheduler

Short Term Scheduler

Medium Term Scheduler

**Differentiate between long term scheduling and medium term scheduling**

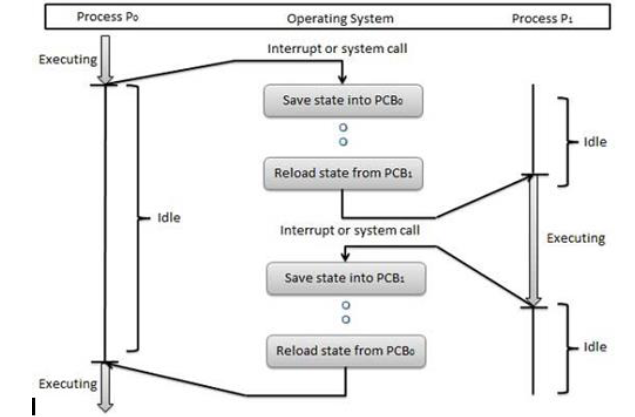
****

**Context switch.**

**Explain working of CPU switch from process to process with diagram.**

A CPU switch from process to process is referred as context switch. A context switch is a mechanism that 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. When the scheduler switches the CPU from one process to another process, the context switch saves the contents of all process registers for the process being removed from the CPU, in its process control block.

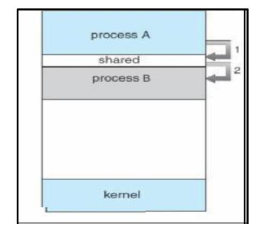
Context switch includes two operations such as state save and state restore. State save operation stores the current information of running process into its PCB. State restore operation restores the information of process to be executed from its PCB. Switching the CPU from one process to another process requires performing state save operation for the currently executing process (blocked) and a state restore operation for the process ready for execution. This task is known as context switch.



**Draw and explain inter-process communication model**

Inter-process communication: Cooperating processes require an Inter-process communication (IPC) mechanism that will allow them to exchange data and information. There are two models of IPC:

1. **Shared memory**

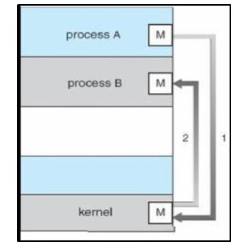


In this a region of the memory residing in an address space of a process creating a shared memory segment can be accessed by all processes who want to communicate with other processes. All the processes using the shared memory segment should attach to the address space of the shared memory. All the processes can exchange information by reading and/or writing data in shared memory segment. The form of data and location are determined by these processes who want to communicate with each other. These processes are not under the control of the operating system. The processes are also responsible for ensuring that

they are not writing to the same location simultaneously. After establishing shared memory segment, all accesses to the shared memory segment are treated as routine memory access and without assistance of kernel.

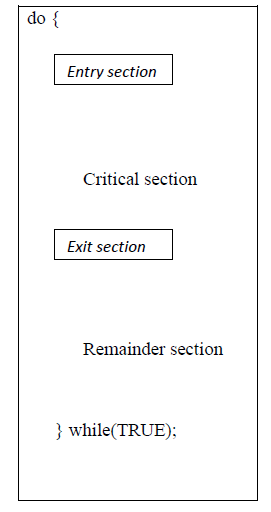
**2. Message Passing:**

In this model, communication takes place by exchanging messages between cooperating processes. It allows processes to communicate and synchronize their action without sharing the same address space. It is particularly useful in a distributed environment when communication process may reside on a different computer connected by a network. Communication requires sending and receiving messages through the kernel. The processes that want to communicate with each other must have a communication link between them. Between each pair of processes exactly one communication link.



**Describe the critical-section problem.**

Each process contains two sections. One is critical section where a process may need to access common variable or objects and other is remaining section containing instructions for processing of sharable objects or local objects of the process. Each process must request for permission to enter inside its critical section. The section of code implementing this request is the entry section. In entry section if a process gets permission to enter into the critical section then it works with common data. At this time all other processes are in waiting state for the same data. The critical section is followed by an exit section. Once the process completes its task, it releases the common data in exit section. Then the remaining code placed in the remainder section is executed by the process.



Two processes cannot execute their critical sections at the same time. The critical section problem is to design a protocol that the processes can use to cooperate i.e. allowing entry to only one process at a time inside the critical section. Before entering into the critical section each process must request for permission to entry inside critical section.

### **Solution to Critical Section Problem**

A solution to the critical section problem must satisfy the following three conditions:

#### 1. Mutual Exclusion

Out of a group of cooperating processes, only one process can be in its critical section at a given point of time.

#### 2. Progress

If no process is in its critical section, and if one or more threads want to execute their critical section then any one of these threads must be allowed to get into its critical section.

#### Bounded Waiting

Bounded waiting means that each process must have a limited waiting time. Itt should not wait endlessly to access the critical section.

**Example**

Problem Description In discussion of the critical section problem, we often assume that each thread is executing the following code. It is also assumed that

(1) After a thread enters a critical section, it will eventually exit the critical section;

(2) A thread may terminate in the non-critical section.

while (true)

{

entry section

critical section

exit section

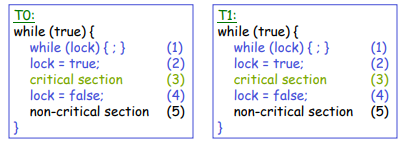
non-critical section

}

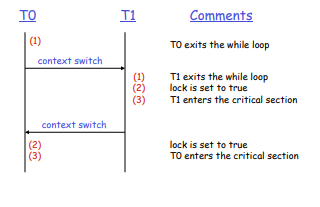
**Solution 1**

In this solution, lock is a global variable initialized to false. A thread sets lock to true to indicate that it is entering the critical section.

boolean lock = false;



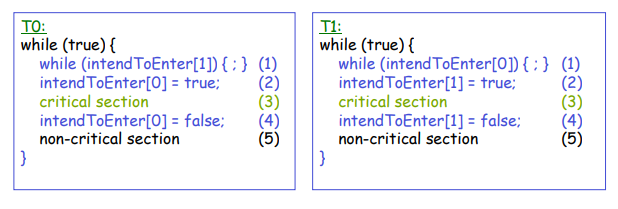
Solution 1 is incorrect!



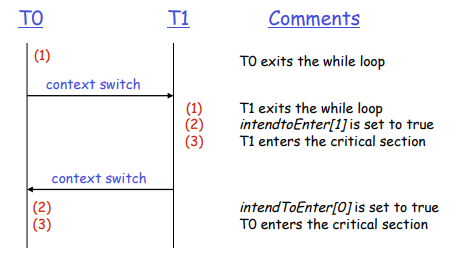
**Solution 2**

The threads use a global array intend To Enter to indicate their intention to enter the critical section.

boolean intendToEnter[] = {false, false};



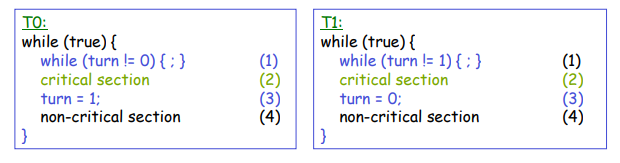
**Solution 2 is incorrect!**



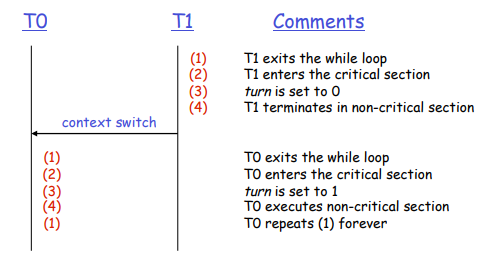
**Solution 3**

The global variable turn is used to indicate the next process to enter the critical section. The initial value of turn can be 0 or 1.

int turn = 1;



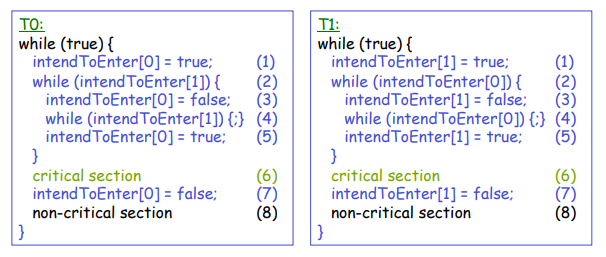
**Solution 3 is incorrect!**

****

**Solution 4**

When a thread finds that the other thread also intends to enter its critical section, it sets its own intendToEnter flag to false and waits until the other thread exits its critical section.

boolean intendToEnter[] = {false, false};



**Explain the working of semaphores.**

Semaphore is a synchronization tool. A semaphore S is an integer variable which is initialized and accessed by only two standard operations: wait () and signal ().All the modifications to the integer value of semaphore in wait () and signal () operations can be done only by one process at a time.

Working of semaphore to solve synchronization problem:- Consider two concurrently running processes P1 and P2.P1 contains statement S1 and P2 contains statement S2.When we want to execute statement S2 only after execution of statement S1, then we can implement it by sharing a common semaphore synch between two processes. Semaphore synch is initialized to 0.to execute the sequence modify code for process P1 and P2.

Process P1 contains:

S1;

signal (synch);

Process P2 contains:-

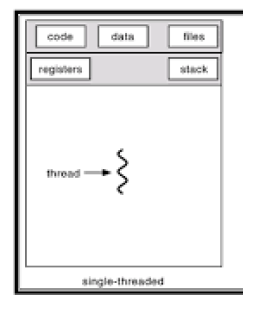
wait (synch);

S2;

As synch is initialized to 0, Process P2 will wait and process P1 will execute. Once process P1 completes execution of statement S1, it performs signal () operation that increments synch value. Then wait () operation checks the incremented value and starts execution of statement S2 from Process P2.

**Define thread. State any three benefits of thread.**

A thread, sometimes called a lightweight process, is a basic unit of CPU utilization. A traditional (or heavyweight) process has a single thread of control. If a process has multiple threads of control, it can do more than one task at a time. This is because there are situations in which it is desirable to have multiple threads of control in the same address space, running as though they were separate processes.



Threads benefits The benefits of multithreaded programming can be broken down into four major categories.

**1. Responsiveness:** Multithreading an interactive application may allow a program to

continue running even if part of it is blocked or is performing a lengthy operation, thereby

increasing responsiveness to the user.

**For example:** A multithreaded web browser could still allow user interaction in one thread while an image is being loaded in another thread. A multithreaded Web server with a front-end and (thread) processing modules.

**2. Resource sharing:** By default, threads share the memory and the resources of the process to which they belong. The benefit of code sharing is that it allows an application to have several different threads of activity all within the same address space. A word processor with three threads.

**For example:** A multithreaded word processor allows all threads to have access to the document being edited.

**3. Economy**: Because threads share resources of the process to which they belong, it is

more economical to create and switch threads, than create and context switch processes

(it is much more time consuming). For example: in Sun OS Solaris 2 creating a process

is about 30 times slower than is creating a thread (context switching is about five times

slower than threads switching).

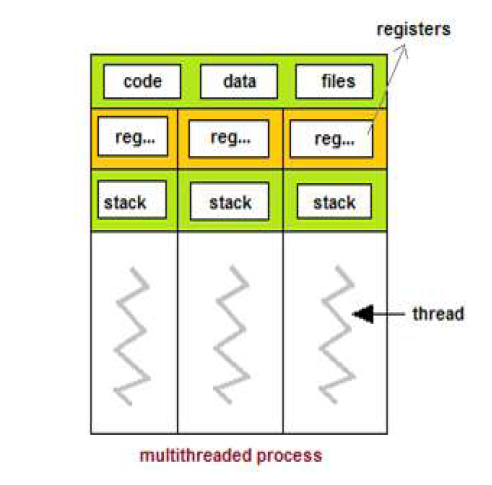
**4. Utilization of multiprocessor architectures**: The benefits of multithreading can be

greatly increased in a multiprocessor architecture (or even in a single-CPU architecture),

where each thread may be running in parallel on a different processor.

**Describe Multithreading:**

Multithreading is the ability of a central processing unit (CPU) or a single core in a multicore processor to execute multiple processes or threads concurrently, appropriately supported by the operating system.



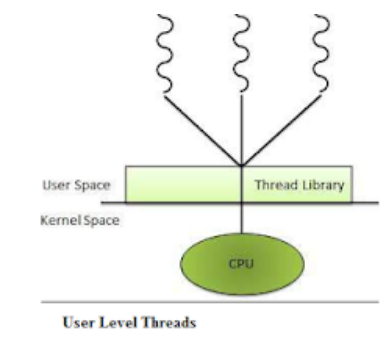
**Differences between a process and a thread**

|  |  |  |
| --- | --- | --- |
| **Comparison Basis** | **Process** | **Thread** |
| Definition | A process is a program under execution i.e an active program. | A thread is a lightweight process that can be managed independently by a scheduler. |
| Context switching time | Processes require more time for context switching as they are more heavy. | Threads require less time for context switching as they are lighter than processes. |
| Memory Sharing | Processes are totally independent and don’t share memory. | A thread may share some memory with its peer threads. |
| Communication | Communication between processes requires more time than between threads. | Communication between threads requires less time than between processes. |
| Blocked | If a process gets blocked, remaining processes can continue execution. | If a user level thread gets blocked, all of its peer threads also get blocked. |
| Resource Consumption | Processes require more resources than threads. | Threads generally need less resources than processes. |
| Dependency | Individual processes are independent of each other. | Threads are parts of a process and so are dependent. |
| Data and Code sharing | Processes have independent data and code segments. | A thread shares the data segment, code segment, files etc. with its peer threads. |
| Treatment by OS | All the different processes are treated separately by the operating system. | All user level peer threads are treated as a single task by the operating system. |
| Time for creation | Processes require more time for creation. | Threads require less time for creation. |
| Time for termination | Processes require more time for termination. | Threads require less time for termination. |

**Explain user thread and kernel threads**

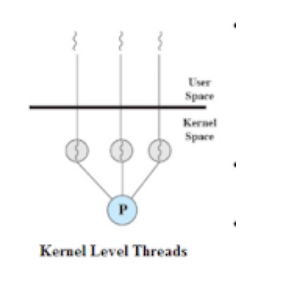
**User-Level Threads:**

* A user-level thread is a thread within a process which the OS does not know about.
* In a user-level thread approach the cost of a context switch between threads less since the operating system itself does not need to be involved–no extra system calls are required.
* A user-level thread is represented by a program counter; registers, stack, and small thread control block (TCB).
* Programmers typically use a thread library to simplify management of threads within a process.
* Creating a new thread, switching between threads, and synchronizing threads are done via function calls into the library. This provides an interface for creating and stopping threads, as well as control over how they are scheduled.

****

**Kernel Threads:**

* In systems that use kernel-level threads, the operating system itself is aware of each individual thread.
* Kernel threads are supported and managed directly by the operating system.
* A context switch between kernel threads belonging to the same process requires only the registers, program counter, and stack to be changed; the overall memory management information does not need to be switched since both of the threads share the same address space. Thus context switching between two kernel threads is slightly faster than switching between two processes.
* Kernel threads can be expensive because system calls are required to switch between threads. Also, since the operating system is responsible for scheduling the threads, the application does not have any control over how its threads are managed.

****

**Difference between user level thread and kernel level thread**

|  |  |
| --- | --- |
| **User level thread** | **Kernel level thread** |
| User threads are supported above the kernel and are implemented by a *thread library* at the user level. The library provides support for thread creation, scheduling, and management with no support from the kernel. | Kernel threads are supported directly by the operating system.  The kernel performs thread creation, scheduling, and management in kernel space. |
| When threads are managed in user space, each process needs its own private thread table to keep track of the threads in that process | No run-time system is needed in each. Also, there is no thread table in each process. Instead, the kernel has a thread table that keeps track of all the threads in the system. |
| User-level  *threads requires non-blocking systems call*, that means a multithreaded kernel. Otherwise, entire process will blocked in the kernel, even if there are runnable threads left in the processes. For example, if one thread causes a page fault, the process blocks. | Kernel threads do not require any new, non-blocking system calls. If one thread in a process causes a page fault, the kernel can easily check to see if the process has any other runnable threads, and if so, run one of them while waiting for the required page to be brought in from the disk. |
| User-level threads are generally fast to create and manage | The kernel-level threads are slow and inefficient. For instance, threads operations are hundreds of times slower than that of user-level threads. |
| User level thread is generic and can run on any operating system. | Kernel level thread is specific to the operating system. |
| *Example:* User-thread libraries include POSIX Pthreads, Mach C-threads, and Solaris 2 UI-threads. | *Example:* Windows NT, Windows 2000, Solaris 2, BeOS, and Tru64 UNIX (formerly Digital UNIX)-support kernel threads. |

**Explain multithreading model with diagram.**

Multithreading models:-

1. Many-to-One

2. One-to-One

3. Many-to-Many

**1. Many-to-One:** - This model maps many user level threads to one kernel level thread.

Thread management is done by thread library in user space.

**Advantages:-**

 It is an efficient model as threads are managed by thread library in user space.

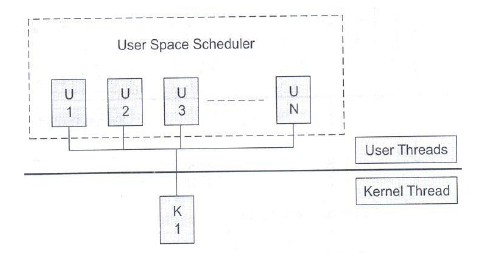
**Disadvantages:-**

 Only one thread can access the kernel at a time, so multiple threads are unable to run in

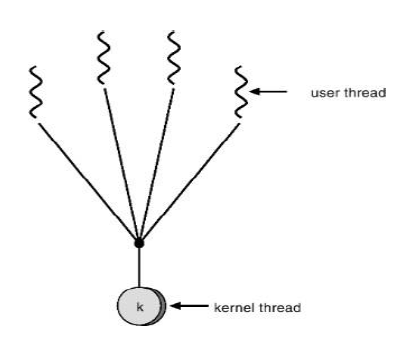
parallel on microprocessor.

 If a thread makes a blocking system call then the entire process will be block.

Example: - Green threads – a thread library available for Solaris use many-to-one model.

****

**OR**

****

**2. One-to-One:** It maps each user level thread to a kernel level thread. Even one thread makes a

blocking call; other thread can run with the kernel thread.

**Advantages:-**

 It allows multiple threads to run in parallel on multiprocessors.

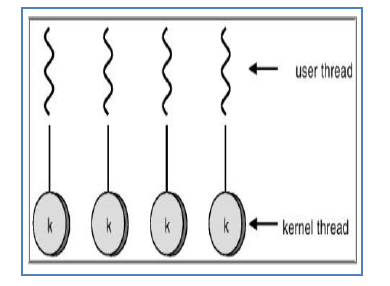
**Disadvantages:-**

 Creating a user thread requires creating the corresponding kernel thread. Creating kernel thread may

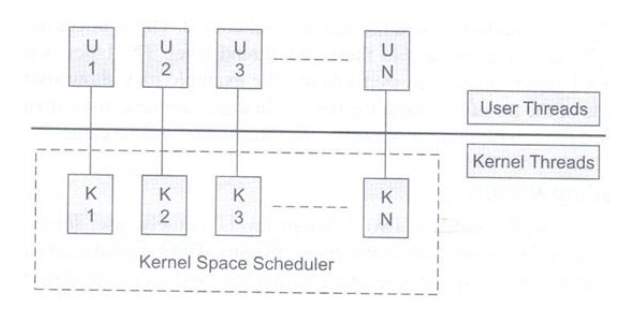
affect the performance of an application.

Example: - Linux, Windows OS including Win 95, 98, NT 2000, and XP implement the

one-to-one model.

****

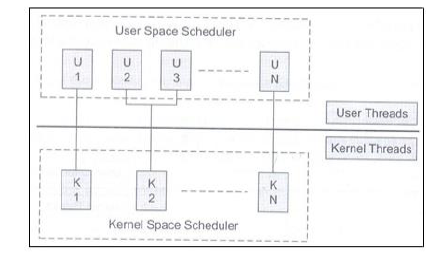
**OR**

****

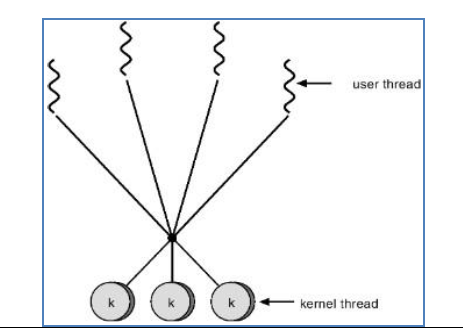
**3. Many-to-Many:**

This model maps many user level threads to a smaller or equal number of kernel threads. Number of

kernel threads may be specific to either a particular application or particular machine.

****

**OR**

****

# **ps command in Linux with Examples**

Linux provides us a utility called **ps** for viewing information related with the processes on a system which stands as abbreviation for **“Process Status”.**ps command is used to list the currently running processes and their PIDs along with some other information depends on different options.

**Syntax –**

**ps [options]**

**Options for ps Command :**

1. **Simple process selection :** Shows the processes for the current shell –

[root@rhel7 ~]# ps

PID TTY TIME CMD

12330 pts/0 00:00:00 bash

21621 pts/0 00:00:00 ps

Result contains four columns of information.  
Where,  
**PID –** the unique process ID  
**TTY –** terminal type that the user is logged into  
**TIME –** amount of CPU in minutes and seconds that the process has been running  
**CMD –** name of the command that launched the process.

**Note –** Sometimes when we execute **ps** command, it shows TIME as 00:00:00. It is nothing but the total accumulated CPU utilization time for any process and 00:00:00 indicates no CPU time has been given by the kernel till now. In above example we found that, for bash no CPU time has been given. This is because bash is just a parent process for different processes which needs bash for their execution and bash itself is not utilizing any CPU time till now.

1. **View Processes :**View all the running processes use either of the following option with ps –

[root@rhel7 ~]# ps -A

[root@rhel7 ~]# ps -e

1. **View Processes not associated with a terminal :** View all processes except both session leaders and processes not associated with a terminal.

[root@rhel7 ~]# ps -a

PID TTY TIME CMD

27011 pts/0 00:00:00 man

27016 pts/0 00:00:00 less

27499 pts/1 00:00:00 ps

1. **View all the processes except session leaders :**

[root@rhel7 ~]# ps -d

1. **View all processes except those that fulfill the specified conditions (negates the selection) :**  
   Example – If you want to see only session leader and processes not associated with a terminal. Then, run

[root@rhel7 ~]# ps -a -N

OR

[root@rhel7 ~]# ps -a --deselect

1. **View all processes associated with this terminal :**

[root@rhel7 ~]# ps -T

1. **View all the running processes :**

[root@rhel7 ~]# ps -r

1. **View all processes owned by you :**Processes i.e same EUID as ps which means runner of the ps command, root in this case –

[root@rhel7 ~]# ps -x

**Wait PID**

**Sleep 10**

**(Delay 10 sec)**

**Exit**

**Kill PID**