Name of Faculty: Vivek Kumar

Designation: Asst. Prof

Department: CSE

Subject: Operating System

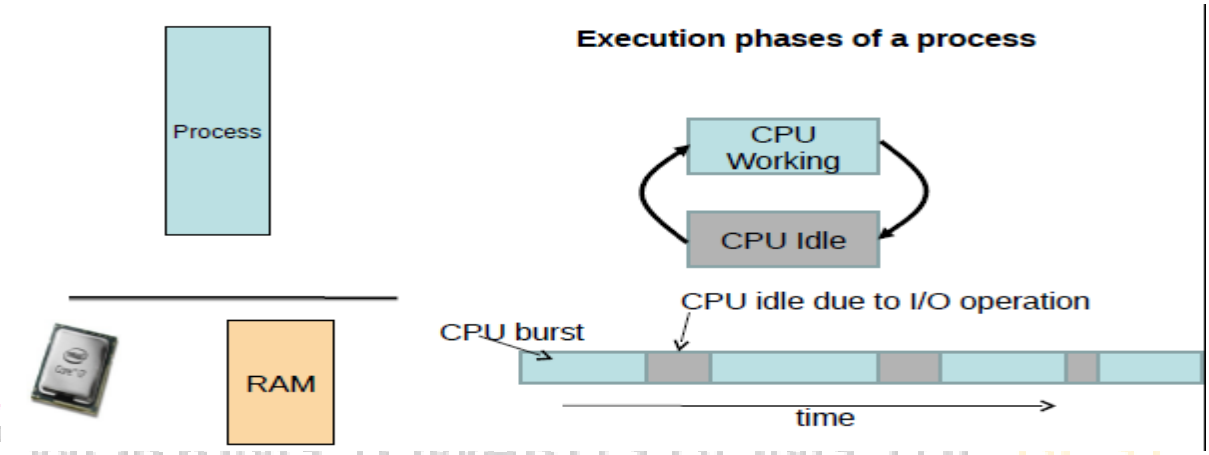
Unit: 03

Topic: CPU Scheduling

Arrival Time: Time at which the process arrives in the ready queue.
Completion Time: Time at which process completes its execution.
Burst Time: Time required by a process for CPU execution.
Turn Around Time: Time Difference between completion time and arrival time.
Turn Around Time = Completion Time – Arrival Time

Why do we need scheduling?
A typical process involves both I/O time and CPU time. In a uni programming system like MS-DOS, time spent waiting for I/O is wasted and CPU is free during this time. In multi programming systems, one process can use CPU while another is waiting for I/O. This is possible only with process scheduling.

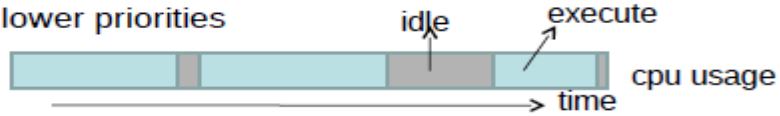
Execution phases of a process



Types of Processes

- I/O bound
  - Has small bursts of CPU activity and then waits for I/O
  - eg. Word processor
  - Affects user interaction (we want these processes to have highest priority)

- CPU bound
  - Hardly any I/O, mostly CPU activity (eg. gcc, scientific modeling, 3D rendering, etc)
    - Useful to have long CPU bursts
  - Could do with lower priorities



- Scheduler triggered to run when timer interrupt occurs or when running process is blocked on I/O Scheduler picks another process from the ready queue Performs a context switch .

Different Scheduling Algorithms

- First Come First Serve (FCFS)
- *Shortest Job First (SJF)*
- *Round Robin Scheduling:*
- *Priority Based scheduling*
- *Multilevel Queue Scheduling*
- *Multi level Feedback Queue Scheduling*

Schedulers

->Decides which process should run next.

-> Aims

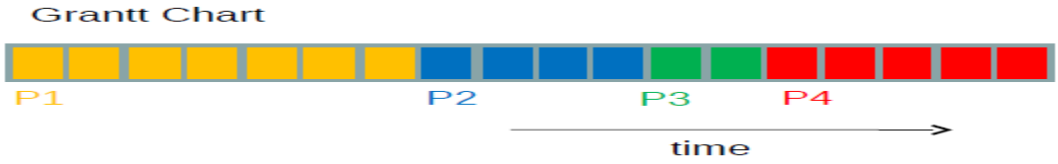
– Minimize waiting time
• Process should not wait long in the ready queue
– Maximize CPU utilization
• CPU should not be idle
– Maximize throughput
• Complete as many processes as possible per unit time
– Minimize response time
• CPU should respond immediately
– Fairness
• Give each process a fair share of CPU

FCFS Scheduling (First Come First Serve)

• First job that requests the CPU gets the CPU
• Non preemptive
– Process continues till the burst cycle ends

FCFS Example

Process	Arrival Time	Burst Time
P1	0	7
P2	0	4
P3	0	2
P4	0	5



Average Waiting Time
= (0 + 7 + 11 + 13) / 4
= 7.75
Average Response Time
= (0 + 7 + 11 + 13) / 4
= 7.75
(same as Average Waiting Time)

FCFS Pros and Cons

• Advantages
– Simple
– Fair (as long as no process hogs the CPU, every process will eventually run)
• Disadvantages
– Waiting time depends on arrival order
– short processes stuck waiting for long process to complete .

Shortest Job First (SJF) no pre-emption

• Schedule process with the shortest burst time
– FCFS if same
• Advantages
– Minimizes average wait time and average response time
• Disadvantages
– Not practical : difficult to predict burst time
• Learning to predict future
– May starve long jobs

Example of SJF (without pre emption)

Process	Arrival Time	Burst Time
P1	0	7
P2	2	4
P3	4	2
7	8	1

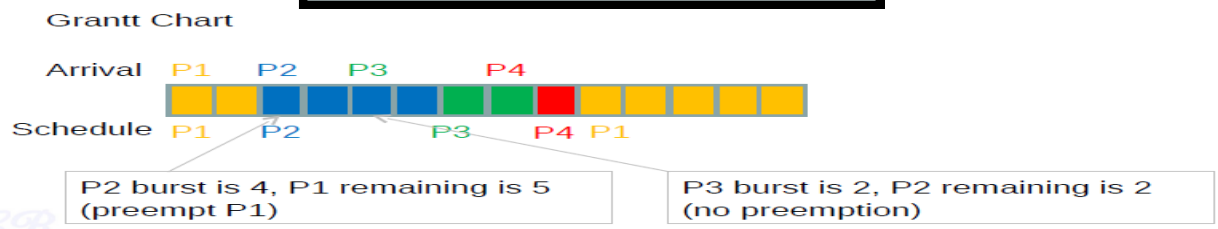
Average wait time

= (0 + 8 + 4 + 0) / 4
= 3
Average response time
= (Average wait time)

Shortest Remaining Time First – SRTF (SJF with pre emption)

- If a new process arrives with a shorter burst time than *remaining of current process* then schedule new process
- Further reduces average waiting time and average response time
- Not practical

Process	Arrival Time	Burst Time
P1	0	7
P2	2	4
P3	4	2
P4	7	1



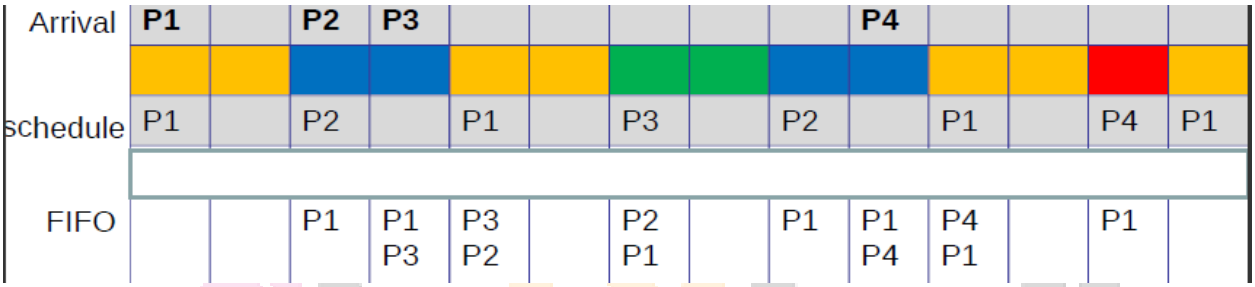
Average wait time
= (7 + 0 + 2 + 1) / 4
= 2.5
Average response time
= (0 + 0 + 2 + 1) / 4
= 0.75

Round Robin Scheduling

• Run process for a time slice then move to FIFO

Process	Arrival Time	Burst Time
P1	0	7
P2	2	4
P3	3	2
P4	9	1

Time Slice =2



Average Waiting time
= (7 + 4 + 3 + 3) / 4
= 4.25
Average Response Time
= (0 + 0 + 3 + 3) / 4
= 1.5

• Advantages

– Fair (Each process gets a fair chance to run on the CPU)
– Low average wait time, when burst times vary
– Faster response time

• Disadvantages

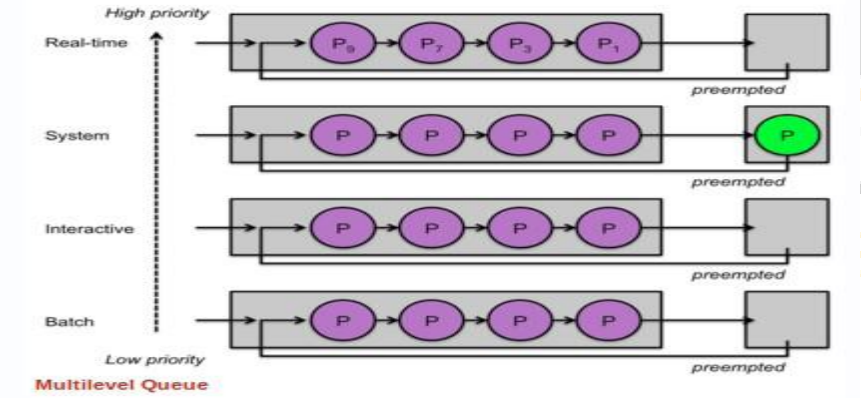
– Increased context switching
• Context switches are overheads!!!
– High average wait time, when burst times have equal lengths

Priority based Scheduling

• Not all processes are equal
– Lower priority for compute intensive processes
– Higher priority for interactive processes (can't keep the user waiting)
• Priority based Scheduling
– Each process is assigned a priority
– Scheduling policy : pick the process in the ready queue having the highest priority
– Advantage : mechanism to provide relative importance to processes
– Disadvantage : could lead to starvation of low priority processes

Multilevel Queues

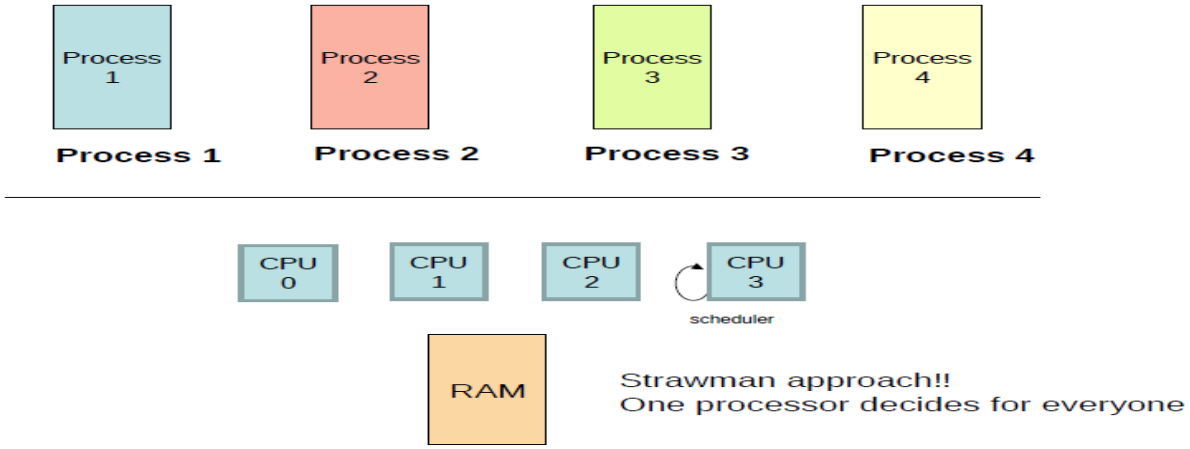
• Processes assigned to a priorityclasses
• Each class has its own readyqueue
• Scheduler picks the highestpriority queue (class) which has atleast one ready process
• Selection of a process within theclass could have its own policy
– Typically round robin (but can bechanged)
– High priority classes can implementfirst come first serve in order to ensurequick response time for critical tasks



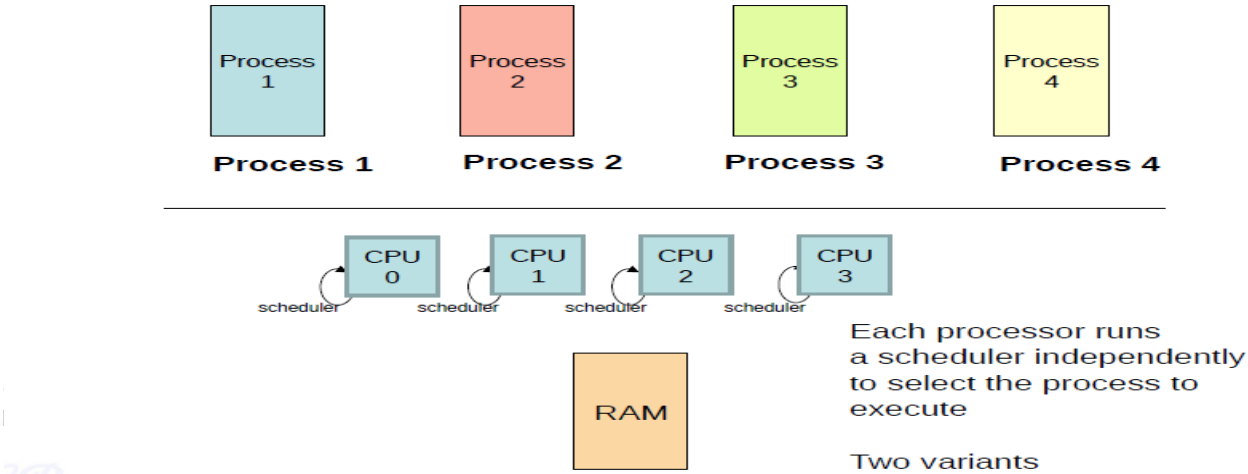
More on Multilevel Queues

• Scheduler can adjust time slice based on the queue class picked
– I/O bound process can be assigned to higher priority classes with larger time slice
– CPU bound processes can be assigned to lower priority classes with shorter time slices
• Disadvantage :
– Class of a process must be assigned apriori (not the most efficient way to do things!)

Multiprocessor Scheduling



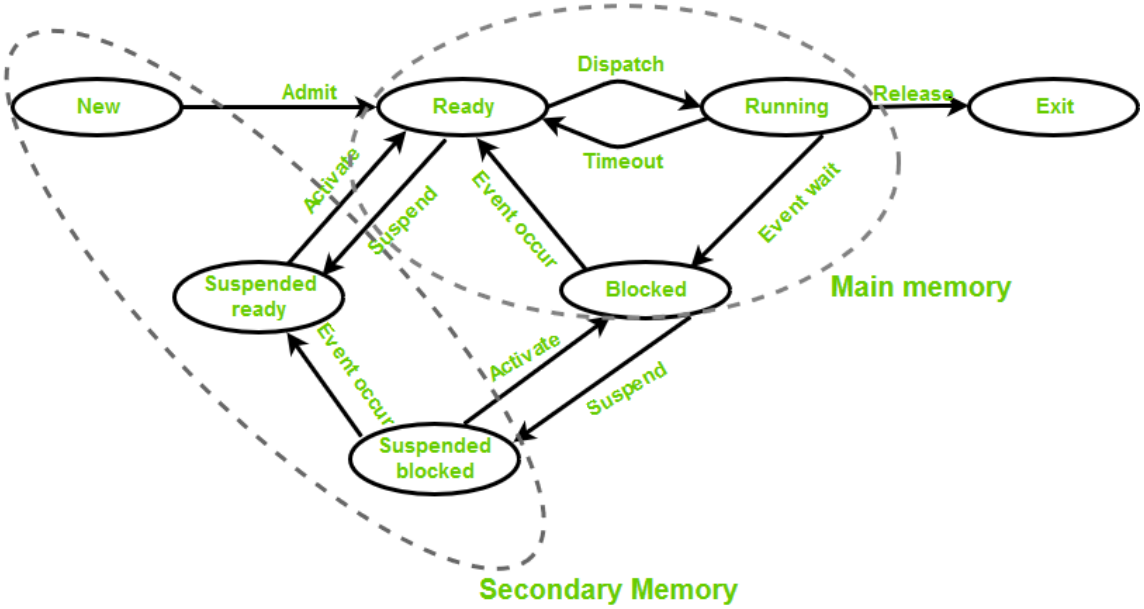
Multiprocessor Scheduling (Symmetical Scheduling)



Load Balancing

• On SMP systems, one processor may be overworked, while another underworked
• Load balancing attempts to keep the workload evenly distributed across all processors
• Two techniques
– Push Migration : A special task periodically monitors load of all processors, and redistributes work when it finds an imbalance
– Pull Migration : Idle processors pull a waiting task from a busy processor

States of a Process in Operating Systems



- **New (Create) –** In this step, the process is about to be created but not yet created, it is the program which is present in secondary memory that will be picked up by OS to create the process.
- **Ready –** 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 for ready processes.
- **Run –** The process is chosen by CPU for execution and the instructions within the process are executed by any one of the available CPU cores.
- **Blocked or wait –** 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 wait 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 –** Process is killed as well as PCB is deleted.
- **Suspend ready –** Process that was initially in the ready state but were swapped out of main memory(refer Virtual Memory topic) and placed onto external storage by scheduler are said to be in suspend ready state. The process will transition back to 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.

System calls for Process Management

In computing, a system call is the programmatic way in which a computer program requests a service from the kernel of the operating system it is executed on. A system call is a way for programs to interact with the operating system. A computer program makes a system call when it makes a request to the operating system's kernel. System call provides the services of the operating system to the user programs via Application Program Interface(API). It provides an interface between a process and operating system to allow user-level processes to request services of the operating system. System calls are the only entry points into the kernel system. All programs needing resources must use system calls.

Services Provided by System Calls :

1. Process creation and management
2. Main memory management
3. File Access, Directory and File system management
4. Device handling(I/O)
5. Protection
6. Networking, etc.

Types of System Calls : There are 5 different categories of system calls –

1. **Process control:** end, abort, create, terminate, allocate and free memory.
2. **File management:** create, open, close, delete, read file etc.
3. Device management
4. Information maintenance
5. Communication

	Windows	Linux
Process Control	CreateProcess()	fork()
	ExitProcess()	exit()
	WaitForSingleObject()	wait()
File Manipulation	CreateFile()	open()
	ReadFile()	read()
	WriteFile()	write()
	CloseHandle()	close()
Device Manipulation	SetConsoleMode()	ioctl()
	ReadConsole()	read()
	WriteConsole()	write()
Information Maintenance	GetCurrentProcessID()	getpid()
	SetTimer()	alarm()
	Sleep()	sleep()
Communication	CreatePipe()	pipe()
	CreateFileMapping()	shmget()
	MapViewOfFile()	mmap()
Protection	SetFileSecurity()	chmod()
	InitlializeSecurityDescriptor()	umask()
	SetSecurityDescriptorGroup()	chown()

Concept of Threads

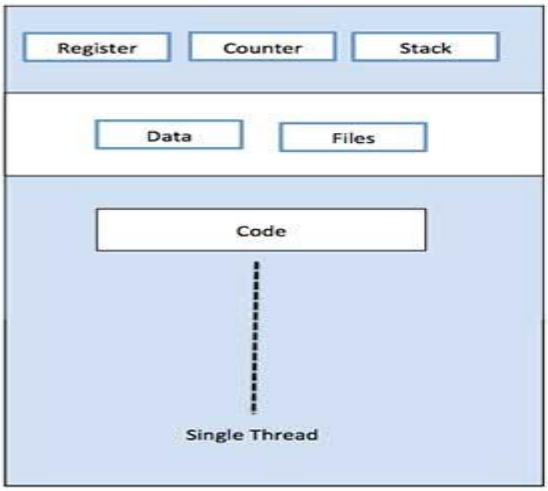
What is Thread

A thread is a flow of execution through the process code, with its own program counter that keeps track of which instruction to execute next, system registers which hold its current working variables, and a stack which contains the execution history.

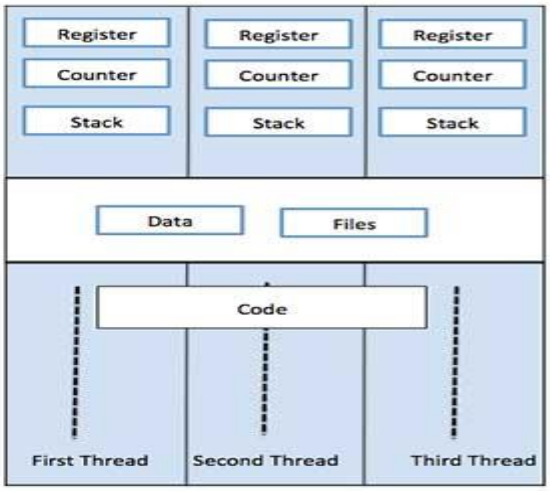
A thread shares with its peer threads few information like code segment, data segment and open files. When one thread alters a code segment memory item, all other threads see that.

A thread is also called a lightweight 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 a separate flow 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. The following figure shows the working of a single-threaded and a multithreaded process.



Single Process P with single thread



Single Process P with three threads

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, a 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 Thread

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

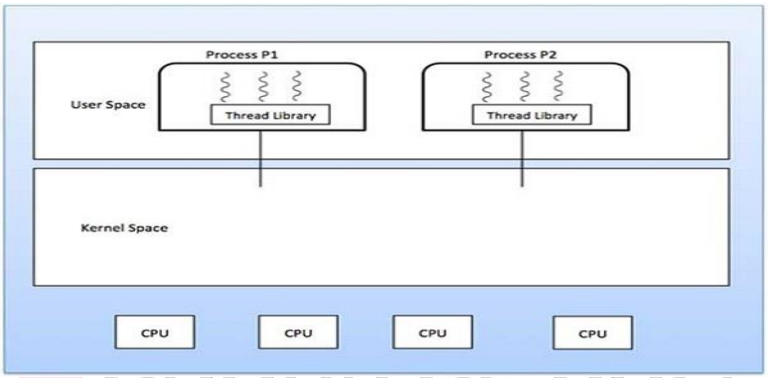
Types of Thread

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, the 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 starts with a single 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 individuals 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 be multithreaded.

Disadvantages

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

Multithreading Models

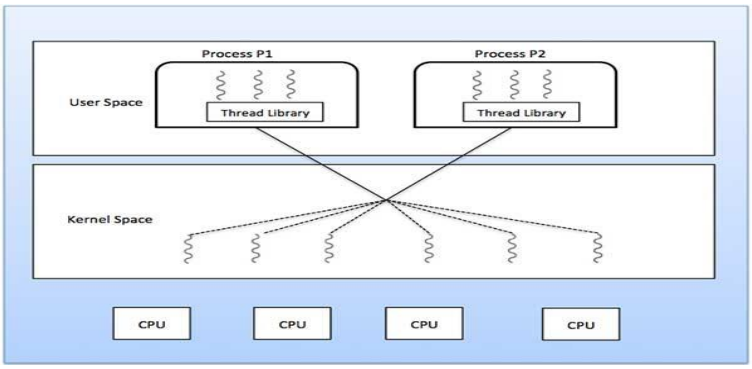
Some operating system provide a combined user level thread and Kernel level thread facility. Solaris is a good example of this combined approach. In a combined system, multiple threads within the same application can run in parallel on multiple processors and a blocking system call need not block the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

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

Many to Many Model

The many-to-many model multiplexes any number of user threads onto an equal or smaller number of kernel threads.

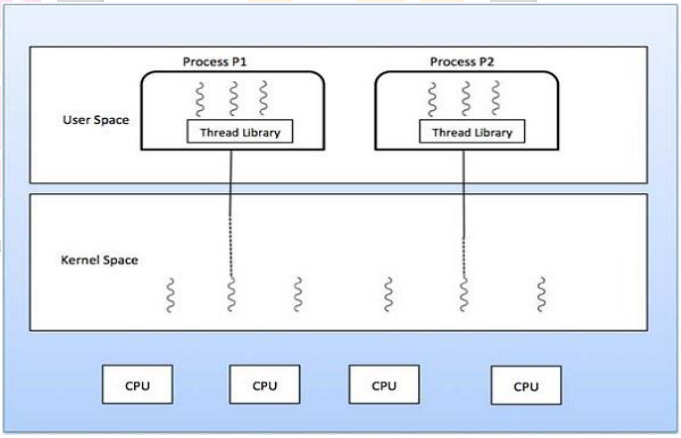
The following diagram shows the many-to-many threading model where 6 user level threads are multiplexing with 6 kernel level threads. In this model, developers can create as many user threads as necessary and the corresponding Kernel threads can run in parallel on a multiprocessor machine. This model provides the best accuracy on concurrency and when a thread performs a blocking system call, the kernel can schedule another thread for execution.



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 by the thread library. When thread makes a blocking system call, the entire process will be blocked. Only one thread can access the Kernel at a time, so multiple threads are unable to run in parallel on multiprocessors.

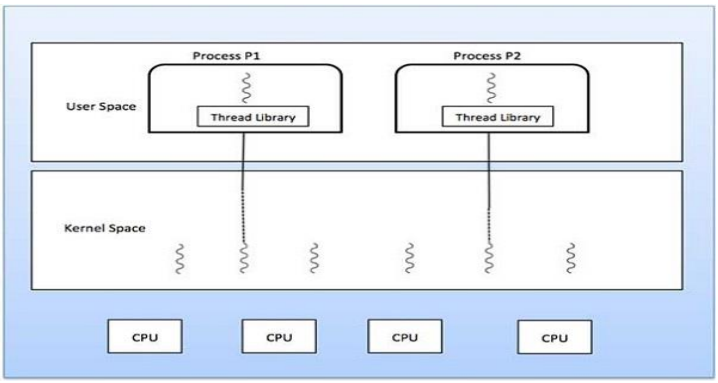
If the user-level thread libraries are implemented in the operating system in such a way that the system does not support them, then the Kernel threads use the many-to-one relationship modes.



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 also 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 applications cannot take advantage of multiprocessing.	Kernel routines themselves can be