THREADS

Thread is part of a Process that can be separately managed by scheduler. Thread is also known as lightweight process. A process can have many threads. All threads of a process share a common data memory, code memory and heap memory area. But each thread has its own thread status such as CPU register values, Program counter and stack.

Multithreading (a process having multiple threads) leads to more speedy execution of the process and the efficient utilisation of the processor time and resources. If the process is split into multiple threads, there will be a main thread and other threads will be created within the main thread. Following figure shows the relationship between process and threads.

Code memory		
Data memory		
Stack	Stack	Stack
Registers	Registers	Registers

Thread1
 	
`void main(void)`
`{`
`//create child thread 1`
`................`
`................`
`//create child thread 2`
`.........................`
`.........................`
`}`
	Thread2
	
	`int ChildThread1(void)`
	`{`
	`.................`
	`................`
	`}`
		Thread3
		`int ChildThread2(void)`
		`{`
		`.................`
		`................`
		`}`

Advantages of multithreading:

- Better memory utilisation
- Improves overall performance
- Efficient CPU utilisation
- Resource sharing
- Provides concurrency within a process.

Thread Pre-emption:

Thread pre-emption is the act of pre-empting the currently running thread (stopping the currently running thread temporarily). The execution switching among threads are known as 'Thread context switching'. There are two types of Threads:

1. User level thread
2. Kernel level thread

The user-level threads are managed by users and the kernel is not aware of it. Even if a process contains multiple user level threads, the OS treats it as single thread and will not switch the execution among the different threads of it. User level threads are non pre-emptive.

Kernel level threads are individual units of execution and managed by kernel. It also known as system level threads. Kernel level threads are pre-emptive.

Comparison of Thread and Process:

Thread	Process
Thread is a single unit of execution and is part of process	Process is a program in execution and contains one or more threads.
It shares common address space of the process	Process has its own address space
Threads are inexpensive to create	Processes are very expensive to create. Involves many OS overhead.
Context switching is fast	Context switching is complex and slow
If a thread expires, its stack is reclaimed by the process	If a process dies, the resources allocated to it are reclaimed by the OS

MULTIPROCESSING AND MULTITASKING

- *Multiprocessing* is the ability to execute multiple processes simultaneously. Such systems consists of multiple CPUs
- The ability of the operating system to have multiple programs in memory, which are ready for execution, is referred as *multiprogramming*.
- In a uniprocessor system, it is not possible to execute multiple processes simultaneously. The ability of an operating system to hold multiple processes in memory and switch the processor (CPU) from executing one process to another process is known as *multitasking*.
    - Multitasking creates the illusion of multiple tasks executing in parallel.
    - Multitasking involves the switching of CPU from executing one task to another
    - The act of switching CPU among the processes is known as ‘Context switching’
    - The act of saving the current context of the process at the time of CPU switching is known as ‘Context saving’.
    - The process of retrieving the saved context details for a process, which is going to be executed due to CPU switching, is known as ‘Context retrieval’

Types of Multitasking:
Depending on the method of context switching, there several types of multitasking, such as:

- Co-operative Multitasking
- Pre-emptive Multitasking
- Non pre-emptive Multitasking

In co-operative multitasking, switching to another task/process will be done only after completing the execution of the current task/process or the current task voluntarily relinquishes the CPU.

In pre-emptive multitasking, every task/process is executed based on a time slot or task priority. When and how much time a process gets is dependent on the implementation of the pre-emptive scheduling.

In non pre-emptive multitasking, the current process/task is allowed to execute until it is completely executed or it enters in a wait state for want of resources.

The difference between co-operative multitasking and non pre-emptive multitasking is that, in co-operative multitasking, the currently executing process/task need not relinquish the CPU when it enters the ‘Blocked/Wait’ state whereas in non-preemptive multitasking the currently executing task relinquishes the CPU when it enters wait state.

TASK SCHEDULING

Determining which task/process is to be executed at a given point of time is known as task/process scheduling. This is implemented using scheduling algorithm and executed by ‘Scheduler’ in kernel. The scheduling decision is taken when a process changes its state. For example, to ‘Ready’ state from ‘Running’ state or to ‘Blocked/Wait’ state from ‘Running’ state etc.

The following factors are to be considered for the selection of scheduling algorithm:

• CPU Utilisation	(how much percentage of the CPU is being utilised)
• Throughput	(number of processes executed per unit of time)
• Turnaround Time	(total time taken by a process for completing its execution)
• Waiting Time	(time spent by a process in the ‘Ready’ queue)
• Response Time	(time elapsed between the submission of a process and the first response.)

A good scheduling algorithm has high CPU utilisation, minimum Turn Around Time (TAT), maximum throughput and least response time.

For task scheduling, operating system maintains the following queues:

• Job Queue	:	Job queue contains all the processes in the system
• Ready Queue	:	Contains all the processes ready for execution and waiting for CPU
• Device Queue	:	Contains all the processes, which are waiting for I/O device.

Based on the scheduling algorithm used, the scheduling can be classified into the following categories:

1. Non-preemptive Scheduling
2. Preemptive Scheduling

Non-preemptive Scheduling

In this scheduling type, the currently executing task/process is allowed to run until it terminates or enters the ‘Wait’ state waiting for an I/O or system resource. It implement non-preemptive multitasking model. Following are the non-preemptive scheduling algorithms:

- First-Come-First-Served (FCFS)/ FIFO Scheduling
- Last-Come-First Served (LCFS)/LIFO Scheduling
- Shortest Job First (SJF) Scheduling
- Priority Based Scheduling

Key features of non-preemptive Scheduling algorithms

1. First-Come-First-Served (FCFS)/ FIFO Scheduling:
    - Allocates CPU time to the processes based on the order in which they enter the ‘Ready’ queue.
    - The first entered process is serviced first.
2. Last-Come-First Served (LCFS)/LIFO Scheduling:
    - Allocates CPU time to the processes based on the order in which they are entered in the ‘Ready’ queue.
    - The last entered process is serviced first.
3. Shortest Job First (SJF) Scheduling:
    - Sorts the ‘Ready’ queue’ each time a process relinquishes the CPU to pick the process with shortest estimated run time.
    - The process with the shortest estimated run time is scheduled first, followed by the next shortest process, and so on.
4. Priority Based Scheduling:
    - Process with high priority is serviced first
    - Priority can be assigned to each task/process at the time of creation