Process and Multithreading

1 Operating System Processes
2 Process Scheduling
3 CPU Scheduling
4 Introduction to Threads
5 Process Synchronization
6 Classical Synchronization Problems
7 Deadlocks

Process and Multithreading

1. Operating System Processes
2. Process Scheduling
3. Operations on Process
4. CPU Scheduling
5. Introduction to Threads

1. Operating System Processes

What is a Process?

A program in the execution is called a Process. Process is not the same as program. A process is more than a program code. A process is an 'active' entity as opposed to program which is considered to be a 'passive' entity. Attributes held by process include hardware state, memory, CPU etc.

Process memory is divided into four sections for efficient working :

1. The text section is made up of the compiled program code, read in from non-volatile storage when the program is launched.
2. The data section is made up the global and static variables, allocated and initialized prior to executing the main.
3. The heap is used for the dynamic memory allocation, and is managed via calls to new, delete, malloc, free, etc.
4. The stack is used for local variables. Space on the stack is reserved for local variables when they are declared.

PROCESS STATE

Processes can be any of the following states :

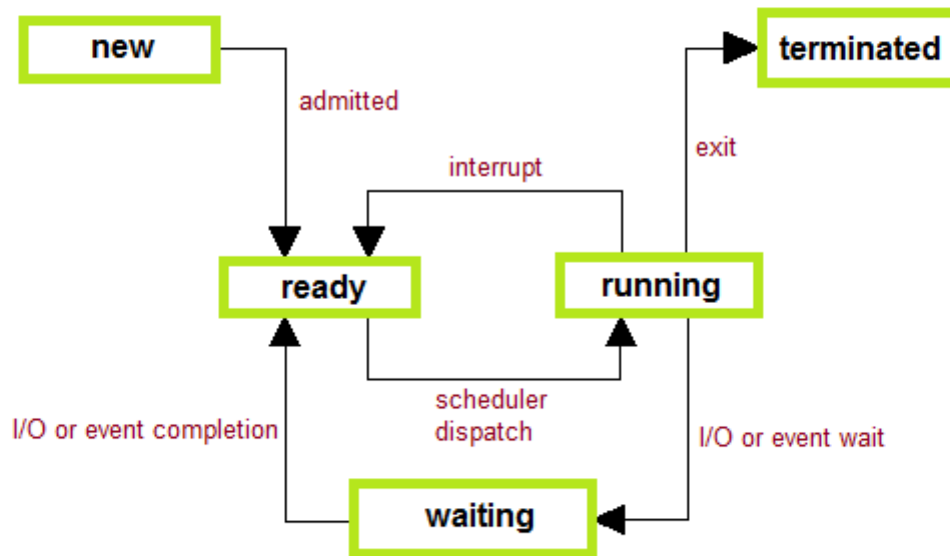
1 **New** - The process is in the stage of being created.

2 **Ready** - The process has all the resources available that it needs to run, but the CPU is not currently working on this process's instructions.

3 **Running** - The CPU is working on this process's instructions.

4 **Waiting** - The process cannot run at the moment, because it is waiting for some resource to become available or for some event to occur.

5 **Terminated** - The process has completed.



PROCESS CONTROL BLOCK

There is a Process Control Block for each process, enclosing all the information about the process. It is a data structure, which contains the following :

4. Process State - It can be running, waiting etc.
5. Process ID and parent process ID.
6. CPU registers and Program Counter. **Program Counter** holds the address of the next instruction to be executed for that process.
7. CPU Scheduling information - Such as priority information and pointers to scheduling queues.
8. Memory Management information - Eg. page tables or segment tables.
9. Accounting information - user and kernel CPU time consumed, account numbers, limits, etc.
10. I/O Status information - Devices allocated, open file tables, etc.

PROCESS STATE
PROCESS NUMBER
PROGRAM COUNTER
REGISTERS
memory limits
list of open files
. . .

2. Process Scheduling

The act of determining which process in the ready state should be moved to the running state is known as Process Scheduling.

The prime aim of the process scheduling system is to keep the CPU busy all the time and to deliver minimum response time for all programs. For achieving this, the scheduler must apply appropriate rules for swapping processes IN and OUT of CPU.

Schedulers fell into one of the two general categories :

5 **Non pre-emptive scheduling.** When the currently executing process gives up the CPU voluntarily.

6 **Pre-emptive scheduling.** When the operating system decides to favour another process, pre-empting the currently executing process.

Scheduling Queues

6 All processes when enters into the system are stored in the **job queue**.

7 Processes in the Ready state are placed in the **ready queue**.

8 Processes waiting for a device to become available are placed in **device queues**. There are unique device queues for each I/O device available.

Types of Schedulers

There are three types of schedulers available :

11 **Long Term Scheduler** : Long term scheduler runs less frequently. Long Term Schedulers decide which program must get into the job queue. From the job queue, the Job Processor, selects processes and loads them into the memory for execution. Primary aim of the Job Scheduler is to maintain a good degree of Multiprogramming. An optimal degree of Multiprogramming means the average rate of process creation is equal to the average departure rate of processes from the execution memory.

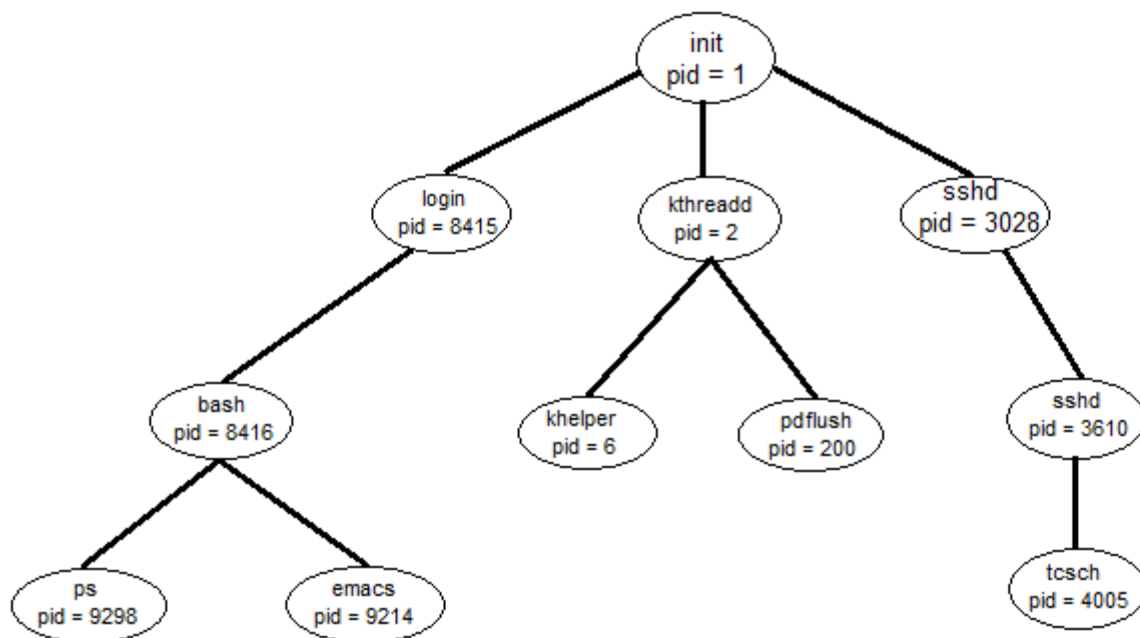
12 **Short Term Scheduler** : This is also known as CPU Scheduler and runs very frequently. The primary aim of this scheduler is to enhance CPU performance and increase process execution rate.

13 **Medium Term Scheduler** : During extra load, this scheduler picks out big processes from the ready queue for some time, to allow smaller processes to execute, thereby reducing the number of processes in the ready queue.

3. Operations on Process

Process Creation

Through appropriate system calls, such as fork or spawn, processes may create other processes. The process which creates other process, is termed the **parent** of the other process, while the created sub-process is termed its **child**.
Each process is given an integer identifier, termed as process identifier, or PID. The parent PID (PPID) is also stored for each process.
On a typical UNIX systems the process scheduler is termed as `sched`, and is given PID 0. The first thing done by it at system start-up time is to launch `init`, which gives that process PID 1. Further Init launches all the system daemons and user logins, and becomes the ultimate parent of all other processes.



A Tree of processes on a typical Linux system

A child process may receive some amount of shared resources with its parent depending on system implementation. To prevent runaway children from consuming all of a certain system resource, child processes may or may not be limited to a subset of the resources originally allocated to the parent.
There are two options for the parent process after creating the child :

- Wait for the child process to terminate before proceeding. Parent process makes a `wait()` system call, for either a specific child process or for any particular child process, which causes the parent process to block until the wait() returns. UNIX shells normally wait for their children to complete before issuing a new prompt.

- Run concurrently with the child, continuing to process without waiting. When a UNIX shell runs a process as a background task, this is the operation seen. It is also possible for the parent to run for a while, and then wait for the child later, which might occur in a sort of a parallel processing operation.

Process Termination

By making the `exit`(system call), typically returning an int, processes may request their own termination. This int is passed along to the parent if it is doing a `wait()`, and is typically zero on successful completion and some non-zero code in the event of any problem.
Processes may also be terminated by the system for a variety of reasons, including :

- The inability of the system to deliver the necessary system resources.
- In response to a KILL command or other unhandled process interrupts.
- A parent may kill its children if the task assigned to them is no longer needed i.e. if the need of having a child terminates.
- If the parent exits, the system may or may not allow the child to continue without a parent (In UNIX systems, orphaned processes are generally inherited by `init`, which then proceeds to kill them.)

When a process ends, all of its system resources are freed up, open files flushed and closed, etc. The process termination status and execution times are returned to the parent if the parent is waiting for the child to terminate, or eventually returned to init if the process already became an orphan.
The processes which are trying to terminate but cannot do so because their parent is not waiting for them are termed **zombies**. These are eventually inherited by init as orphans and killed off.

4. CPU Scheduling

CPU scheduling is a process which allows one process to use the CPU while the execution of another process is on hold(in waiting state) due to unavailability of any resource like I/O etc, thereby making full use of CPU. The aim of CPU scheduling is to make the system efficient, fast and fair.

Scheduling Criteria

There are many different criterias to check when considering the "best" scheduling algorithm :

7 **CPU utilization**To make out the best use of CPU and not to waste any CPU cycle, CPU would be working most of the time(Ideally 100% of the time). Considering a real system, CPU usage should range from 40% (lightly loaded) to 90% (heavily loaded.)

8 **Throughput**It is the total number of processes completed per unit time or rather say total amount of work done in a unit of time. This may range from 10/second to 1/hour depending on the specific processes.

9 **Turnaround time**It is the amount of time taken to execute a particular process, i.e. The interval from time of submission of the process to the time of completion of the process(Wall clock time).

10 **Waiting time**The sum of the periods spent waiting in the ready queue amount of time a process has been waiting in the ready queue to acquire get control on the CPU.

11 **Load average**It is the average number of processes residing in the ready queue waiting for their turn to get into the CPU.

12 **Response time**Amount of time it takes from when a request was submitted until the first response is produced. Remember, it is the time till the first response and not the completion of process execution(final response).

In general CPU utilization and Throughput are maximized and other factors are reduced for proper optimization.

Scheduling Algorithms

We'll discuss four major scheduling algorithms here which are following :

1. First Come First Serve(FCFS) Scheduling
2. Shortest-Job-First(SJF) Scheduling
3. Priority Scheduling
4. Round Robin(RR) Scheduling
5. Multilevel Queue Scheduling

First Come First Serve(FCFS) Scheduling

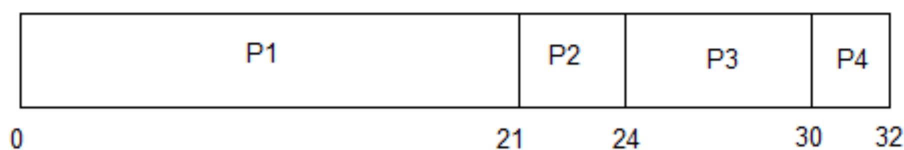
14 Jobs are executed on first come, first serve basis.

15 Easy to understand and implement.

16 Poor in performance as average wait time is high.

PROCESS	BURST TIME
P1	21
P2	3
P3	6
P4	2

The average waiting time will be = (0 + 21 + 24 + 30)/4 = 18.75 ms



This is the GANTT chart for the above processes

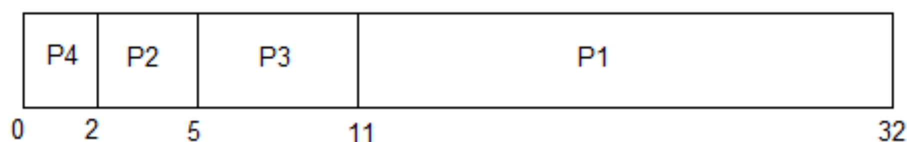
Shortest-Job-First(SJF) Scheduling

- Best approach to minimize waiting time.
- Actual time taken by the process is already known to processor.
- Impossible to implement.

PROCESS	BURST TIME
P1	21
P2	3
P3	6
P4	2

In Shortest Job First Scheduling, the shortest Process is executed first.

Hence the GANTT chart will be following :

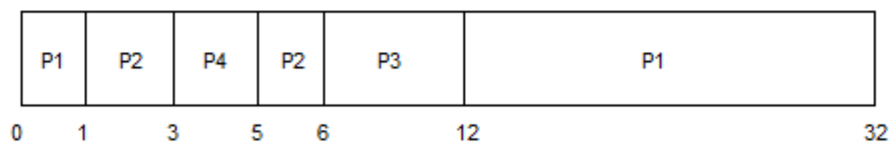


Now, the average waiting time will be = (0 + 2 + 5 + 11)/4 = 4.5 ms

In Preemptive Shortest Job First Scheduling, jobs are put into ready queue as they arrive, but as a process with short burst time arrives, the existing process is preempted.

PROCESS	BURST TIME	ARRIVAL TIME
P1	21	0
P2	3	1
P3	6	2
P4	2	3

The GANTT chart for Preemptive Shortest Job First Scheduling will be,



The average waiting time will be, ((5-3) + (6-2) + (12-1))/4 = 4.25 ms

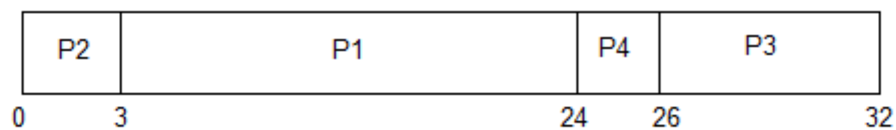
The average waiting time for preemptive shortest job first scheduling is less than both, non-preemptive SJF scheduling and FCFS scheduling.

Priority Scheduling

- Priority is assigned for each process.
- Process with highest priority is executed first and so on.
- Processes with same priority are executed in FCFS manner.
- Priority can be decided based on memory requirements, time requirements or any other resource requirement.

PROCESS	BURST TIME	PRIORITY
P1	21	2
P2	3	1
P3	6	4
P4	2	3

The GANTT chart for following processes based on Priority scheduling will be,



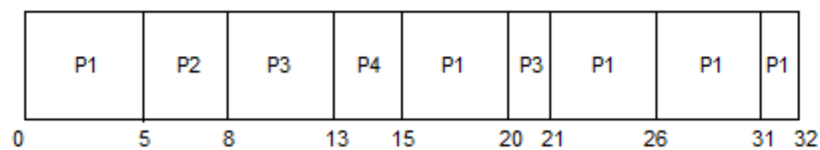
The average waiting time will be, (0 + 3 + 24 + 26)/4 = 13.25 ms

Round Robin(RR) Scheduling

- A fixed time is allotted to each process, called **quantum**, for execution.
- Once a process is executed for given time period that process is preemptied and other process executes for given time period.
- Context switching is used to save states of preemptied processes.

PROCESS	BURST TIME
P1	21
P2	3
P3	6
P4	2

The GANTT chart for round robin scheduling will be,



The average waiting time will be, 11 ms.

Multilevel Queue Scheduling

- Multiple queues are maintained for processes.
- Each queue can have its own scheduling algorithms.

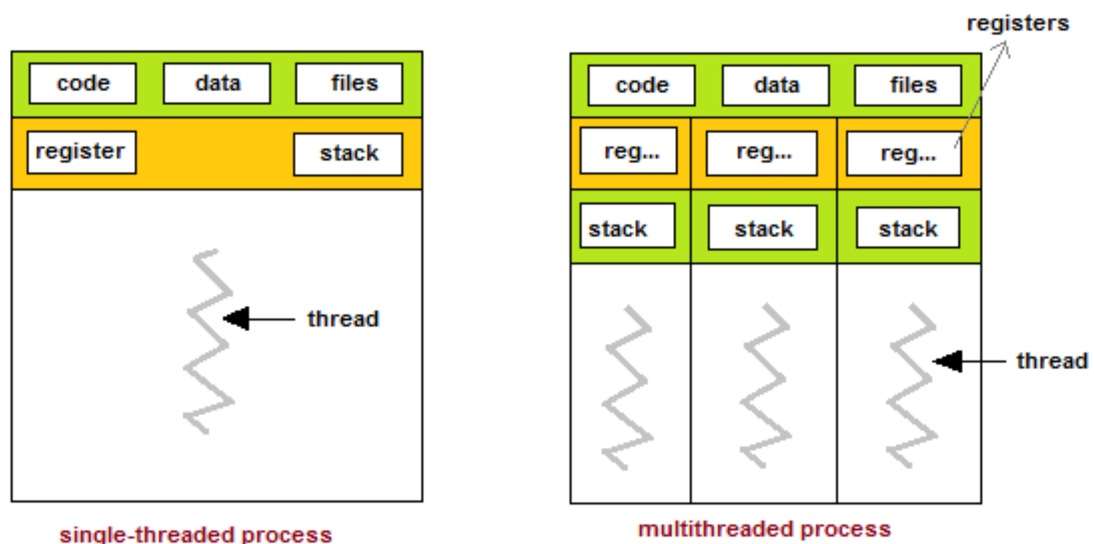
Priorities are assigned to each queue.

5. Introduction to Threads

What are Threads?

Thread is an execution unit which consists of its own program counter, a stack, and a set of registers. Threads are also known as Lightweight processes. Threads are popular way to improve application through parallelism. The CPU switches rapidly back and forth among the threads giving illusion that the threads are running in parallel.

As each thread has its own independent resource for process execution, multpile processes can be executed parallely by increasing number of threads.



Types of Thread

There are two types of threads :

13 User Threads

14 Kernel Threads

User threads, are above the kernel and without kernel support. These are the threads that application programmers use in their programs.

Kernel threads are supported within the kernel of the OS itself. All modern OSs support kernel level threads, allowing the kernel to perform multiple simultaneous tasks and/or to service multiple kernel system calls simultaneously.

Multithreading Models

The user threads must be mapped to kernel threads, by one of the following strategies.

9 Many-To-One Model

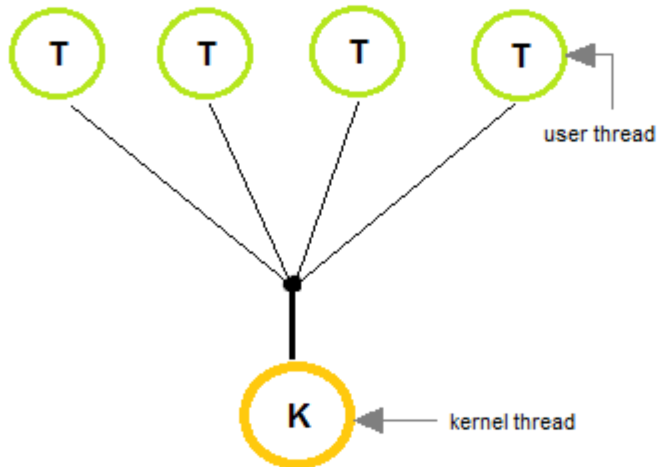
10 One-To-One Model

11 Many-To-Many Model

Many-To-One Model

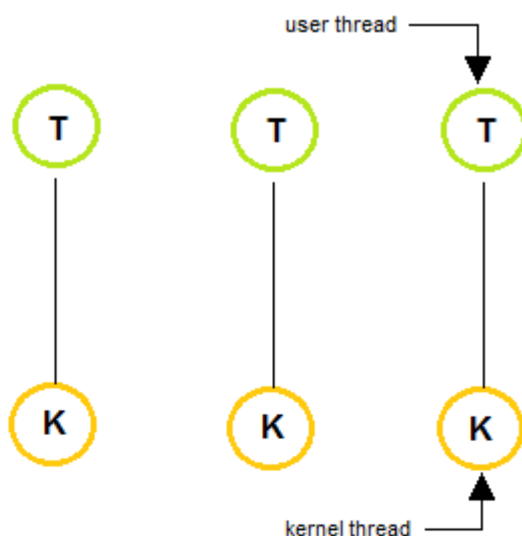
17 In the many-to-one model, many user-level threads are all mapped onto a single kernel thread.

18 Thread management is handled by the thread library in user space, which is efficient in nature.



One-To-One Model

- The one-to-one model creates a separate kernel thread to handle each and every user thread.
- Most implementations of this model place a limit on how many threads can be created.
- Linux and Windows from 95 to XP implement the one-to-one model for threads.

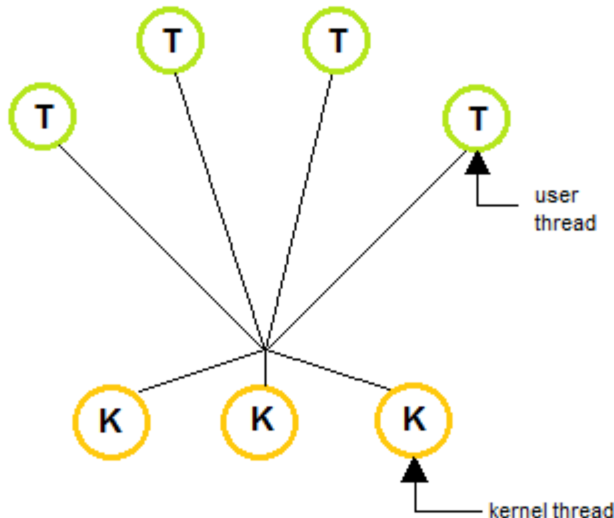


Many-To-Many Model

- The many-to-many model multiplexes any number of user threads onto

an equal or smaller number of kernel threads, combining the best features of the one-to-one and many-to-one models.

- Users can create any number of the threads.
- Blocking the kernel system calls does not block the entire process.
- Processes can be split across multiple processors.



Thread Libraries

Thread libraries provides programmers with API for creating and managing of threads.
Thread libraries may be implemented either in user space or in kernel space. The user space involves API functions implemented solely within user space, with no kernel support. The kernel space involves system calls, and requires a kernel with thread library support.

There are three types of thread :

- POSIX Pitheads, may be provided as either a user or kernel library, as an extension to the POSIX standard.
- Win32 threads, are provided as a kernel-level library on Windows systems.
- Java threads - Since Java generally runs on a Java Virtual Machine, the implementation of threads is based upon whatever OS and hardware the JVM is running on, i.e. either Pitheads or Win32 threads depending on the system

Benefits of Multithreading

- Responsiveness
- Resource sharing, hence allowing better utilization of resources.
- Economy. Creating and managing threads becomes easier.
- Scalability. One thread runs on one CPU. In Multithreaded processes, threads can be distributed over a series of processors to scale.

- Context Switching is smooth. Context switching refers to the procedure followed by CPU to change from one task to another.

Multithreading Issues

1. **Thread Cancellation**. Thread cancellation means terminating a thread before it has finished working. There can be two approaches for this, one is **Asynchronous cancellation**, which terminates the target thread immediately. The other is **Deferred cancellation** allows the target thread to periodically check if it should be cancelled.
2. **Signal Handling**. Signals are used in UNIX systems to notify a process that a particular event has occurred. Now in when a Multithreaded process receives a signal, to which thread it must be delivered? It can be delivered to all, or a single thread.
3. **fork() System Call**. fork() is a system call executed in the kernel through which a process creates a copy of itself. Now the problem in Multithreaded process is, if one thread forks, will the entire process be copied or not?
4. **Security Issues** because of extensive sharing of resources between multiple threads.

There are many other issues that you might face in a multithreaded process, but there are appropriate solutions available for them. Pointing out some issues here was just to study both sides of the coin.