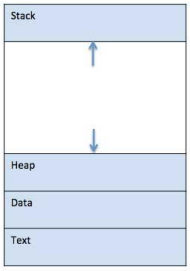
LECTURE 2 - PROCESS MANAGEMENT.

A process is a program in execution. The execution of a process must progress in a sequential fashion.

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 layout of a process inside main memory −



S.N.	Component & Description
1	**Stack -** The process Stack contains the temporary data such as method/function parameters, return address and local variables.
2	**Heap -** This is dynamically allocated memory to a process during its run time.
3	**Text -** This includes the current activity represented by the value of Program Counter and the contents of the processor's registers.
4	**Data** This section contains the global and static variables.

Program

A program is a piece of code which may be a single line or millions of lines. A computer program is usually written by a computer programmer in a programming language.

Process Life Cycle

When a process executes, it passes through different states. These stages may differ in different operating systems, and the names of these states are also not standardized.

In general, a process can have one of the following five states at a time.

S.N.	State & Description
1	**Start** This is the initial state when a process is first started/created.
2	**Ready** The process is waiting to be assigned to a processor. Ready processes are waiting to have the processor allocated to them by the operating system so that they can run. Process may come into this state after **Start** state or while running it by but interrupted by the scheduler to assign CPU to some other process.
3	**Running**

	Once the process has been assigned to a processor by the OS scheduler, the process state is set to running and the processor executes its instructions.
4	**Waiting** Process moves into the waiting state if it needs to wait for a resource, such as waiting for user input, or waiting for a file to become available.
5	**Terminated or Exit** Once the process finishes its execution, or it is terminated by the operating system, it is moved to the terminated state where it waits to be removed from main memory.

Process Control Block (PCB)

A Process Control Block is a data structure maintained by the Operating System for every process. The PCB is identified by an integer process ID (PID).

Interprocess communication provides a mechanism to allow processes to communicate and to synchronize their actions. There are three issues related to inter-process communication:

- How one process can pass information to another.
- How to ensure that two or more processes do not get in each other's way when engaging in critical activities.
- Proper sequencing of where dependencies are present.

2.2.1 Race Conditions

In some operating systems, processes working together may share some common storage that each can read and write. When two or more processes are reading or writing some shared data and the final result depends on who runs when are called *race conditions*.

2.2.2 Critical Sections

The solution to race condition is to prevent more than one process from reading and writing the shared data at the same time.
We need **mutual exclusion –** a way to ensure that if one process is using a shared resource (variable or file) the other processes will be excluded from doing the same thing. The choice of appropriate operations for achieving mutual exclusion is a major design issue in any operating system.

The problem of avoiding race conditions can also be formulated in an abstract way. Part of the time, a process does computations that do not lead to race conditions. However, other times a process may be accessing shared memory or files or doing other critical things that could lead to race conditions. The part of the program where shared memory is accessed is called the **critical region** or **critical section**. If no two processes were ever in their critical sections at the same time, we could avoid race conditions.

Although this requirement avoids race conditions, this is not sufficient for having parallel processes cooperate correctly and efficiently using shared data. We need four conditions to hold to have a good solution:

a) No two processes may be simultaneously inside their critical regions
b) No assumptions may be made about the speeds or number of CPUs
c) No process running outside its critical region may block other processes
d) No process should wait forever to enter its critical region

Process Scheduling

When more than one process can be run, the operating system must decide which one to run first. The part of the operating system that makes this decision is known as the **scheduler** and the algorithm it uses is called the **scheduling algorithm.**

The scheduler is usually concerned with deciding the policy and not the mechanism. There are various consideration when deciding a good scheduling algorithm. Some of them include:

1. Fairness – making sure that each process gets its fair share of the CPU
2. Efficiency – keep the CPU busy 100% of the time
3. Response time – minimize response time for interactive users
4. Turnaround time – minimize the time that users must wait for output
5. Throughput – maximize the number of jobs processed per unit time.

Some of these goals are contradictory. Any scheduling algorithm that favors some class of jobs hurts another class of jobs.

It is also difficult to predict the behavior of all jobs. Some may be CPU-intensive while others are I/O-intensive and you cannot determine when one process will block. All computers have an electronic time that causes interrupts periodically.

At each clock interrupt, the operating system decides whether the currently running process should be allowed to continue or be terminated.

The strategy of allowing processes that are logically runnable to be temporarily suspended is called **pre-emptive scheduling** as opposed to **non-preemptive scheduling** where a process is allowed to run to completion.

The non-preemptive scheduling though easy to implement, it is not suitable for general-purpose systems with multiple competing users because letting one process to run for as long as it wanted would mean denying service to other processes indefinitely.

Round Robin Scheduling

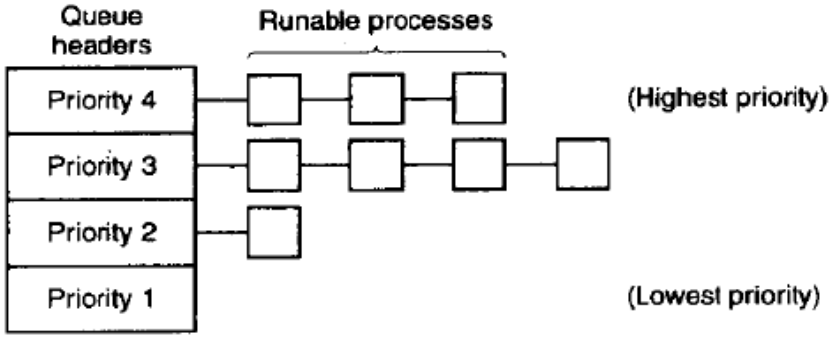
- Round Robin is the preemptive process scheduling algorithm.
- Each process is provided a fix time to execute, it is called a **quantum**.
- Once a process is executed for a given time period, it is preempted and other process executes for a given time period.
- Context switching is used to save states of preempted processes.
- To implement this algorithm, the scheduler maintains a list of runnable processes. When a process uses up its quantum it is put on the end of the list.

The major consideration in this algorithm is the length of the quantum. Since context switching also takes time, if the length of the quantum is short, the CPU will spend more time doing context switching rather than doing actual processing work. To improve the CPU efficiency, we should set a longer quantum time, but not too long since it may cause poor response time.

Round robin scheduling

Priority Scheduling

- Priority scheduling is a non-preemptive algorithm and one of the most common scheduling algorithms in batch systems.
- Each process is assigned a priority. Process with highest priority is to be executed first and so on.
- Processes with same priority are executed on first come first served basis.
- Priority can be decided based on memory requirements, time requirements or any other resource requirement.
- It is often convenient to group processes into priority classes and use priority scheduling among the classes but round robin scheduling within each class.



Priority Scheduling

Shortest Job First

- This is also known as **shortest job next**
- This is a non-preemptive, pre-emptive scheduling algorithm.
- Best approach to minimize waiting time.
- Easy to implement in Batch systems where required CPU time is known in advance.
- Impossible to implement in interactive systems where required CPU time is not known.
- The processer should know in advance how much time process will take.

When several equally important jobs are sitting in the input queue waiting to be started, the scheduler should use the **shortest job first**. By running them in that order, the

turnaround time for A is 8 minutes, B is 12 minutes, for C is 16 minutes and for D is 20 minutes for an average of 14 minutes.

shortest job first Scheduling

Now let us consider running these jobs using the shortest job first as shown in figure 16(b). The turnaround times are now 4,8,12 and 20 minutes for an average of 11 minutes. Shortest job first is optimal.

First Come First Serve (FCFS)

- Jobs are executed on first come, first serve basis.
- It is a non-preemptive, pre-emptive scheduling algorithm.
- Easy to understand and implement.
- Its implementation is based on FIFO queue.
- Poor in performance as average wait time is high.

CPU Scheduling:

- What is CPU scheduling? Determining which processes run when there are multiple runnable processes. Why is it important? Because it can can have a big effect on resource utilization and the overall performance of the system.
- Basic assumptions behind most scheduling algorithms:
    - There is a pool of runnable processes contending for the CPU.
    - The processes are independent and compete for resources.
    - The scheduler distributes the CPU resource ``fairly'' and in a way that optimizes some performance criteria.

In general, these assumptions are starting to break down.

1. CPUs are not really that scarce - almost everybody has several, and pretty soon people will be able to afford lots.
2. Many applications are starting to be structured as multiple cooperating processes. So, a view of the scheduler as mediating between competing entities may be partially obsolete.

- How do processes behave? First, CPU/IO burst cycle. A process will run for a while (the CPU burst), perform some IO (the IO burst), then run for a while more (the next CPU burst). How long between IO operations? Depends on the process.
    - IO Bound processes: processes that perform lots of IO operations. Each IO operation is followed by a short CPU burst to process the IO, then more IO happens.
    - CPU bound processes: processes that perform lots of computation and do little IO. Tend to have a few long CPU bursts.

One of the things a scheduler will typically do is switch the CPU to another process when one process does IO. Why? The IO will take a long time, and don't want to leave the CPU idle while wait for the IO to finish.

- When do scheduling decisions take place? When does CPU choose which process to run? Are a variety of possibilities:

    - When process switches from running to waiting. Could be because of IO request, because wait for child to terminate, or wait for synchronization operation (like lock acquisition) to complete.
    - When process switches from running to ready - on completion of interrupt handler, for example. Common example of interrupt handler - timer interrupt in interactive systems. If scheduler switches processes in this case, it has preempted the running process. Another common case interrupt handler is the IO completion handler.
    - When a process switches from waiting to ready state (on completion of IO or acquisition of a lock, for example).
    - When a process terminates.
- How to evaluate scheduling algorithm? There are many possible criteria:
    - CPU Utilization: Keep CPU utilization as high as possible.
    - Throughput: number of processes completed per unit time.
    - Turnaround Time: mean time from submission to completion of a process.
    - Waiting Time: Amount of time spent ready to run but not running.
    - Response Time: Time between submission of requests and first response to the request.
    - Scheduler Efficiency: The scheduler doesn't perform any useful work, so any time it takes is pure overhead. So, need to make the scheduler very efficient.
- Big difference: Batch and Interactive systems. In batch systems, typically want good throughput or turnaround time. In interactive systems, both of these are still usually important (after all, want some computation to happen), but response time is usually a primary consideration. And, for some systems, throughput or turnaround time is not really relevant - some processes conceptually run forever.
- Difference between long and short term scheduling. Long term scheduler is given a set of processes and decides which ones should start to run. Once they start running, they may suspend because of IO or because of preemption. Short term scheduler decides which of the available jobs that long term scheduler has decided are runnable to actually run.

Scheduling algorithms.

- **First-Come, First-Served**. One ready queue, OS runs the process at head of queue, new processes come in at the end of the queue. A process does not give up CPU until it either terminates or performs IO.
- Consider performance of FCFS algorithm for three computer-bound processes. What if have 3 processes P1 (takes 24 seconds), P2 (takes 3 seconds) and P3 (takes 3 seconds). If arrive in order P1, P2, P3, what is
    - Waiting Time? (24 + 27) / 3 = 17
    - Turnaround Time? (24 + 27 + 30) = 27.
    - Throughput? 30 / 3 = 10.

  What about if processes come in order P2, P3, P1? What is
    - Waiting Time? (3 + 3) / 2 = 6
    - Turnaround Time? (3 + 6 + 30) = 13.
    - Throughput? 30 / 3 = 10.

- **Shortest-Job-First (SJF)** can eliminate some of the variance in Waiting and Turnaround time. In fact, it is optimal with respect to average waiting time. Big problem: how does scheduler figure out how long will it take the process to run?
- **Preemptive vs. Non-preemptive SJF scheduler**. Preemptive scheduler reruns scheduling decision when process becomes ready. If the new process has priority over running process, the CPU preempts the running process and executes the new process. Non-preemptive scheduler only does scheduling decision when running process voluntarily gives up CPU. In effect, it allows every running process to finish its CPU burst.
- Consider 4 processes P1 (burst time 8), P2 (burst time 4), P3 (burst time 9) P4 (burst time 5) that arrive one time unit apart in order P1, P2, P3, P4. Assume that after burst happens, process is not reenabled for a long time (at least 100, for example). What does a preemptive SJF scheduler do? What about a non-preemptive scheduler?
- **Priority Scheduling**. Each process is given a priority, then CPU executes process with highest priority. If multiple processes with same priority are runnable, use some other criteria - typically FCFS. SJF is an example of a priority-based scheduling algorithm. With the exponential decay algorithm above, the priorities of a given process change over time.
- Assume we have 5 processes P1 (burst time 10, priority 3), P2 (burst time 1, priority 1), P3 (burst time 2, priority 3), P4 (burst time 1, priority 4), P5 (burst time 5, priority 2). Lower numbers represent higher priorities. What would a standard priority scheduler do?
- Big problem with priority scheduling algorithms: starvation or blocking of low-priority processes. Can use aging to prevent this - make the priority of a process go up the longer it stays runnable but isn't run.
- How well does RR work? Well, it gives good response time, but can give bad waiting time. Consider the waiting times under round robin for 3 processes P1 (burst time 24), P2 (burst time 3), and P3 (burst time 4) with time quantum 4. What happens, and what is average waiting time? What gives best waiting time?
- What happens with really a really small quantum? It looks like you've got a CPU that is 1/n as powerful as the real CPU, where n is the number of processes. Problem with a small quantum - context switch overhead.
- **Multilevel Queue Scheduling** - like RR, except have multiple queues. Typically, classify processes into separate categories and give a queue to each category. So, might have system, interactive and batch processes, with the priorities in that order. Could also allocate a percentage of the CPU to each queue.
- Multilevel Feedback Queue Scheduling - Like multilevel scheduling, except processes can move between queues as their priority changes. Can be used to give IO bound and interactive processes CPU priority over CPU bound processes. Can also prevent starvation by increasing the priority of processes that have been idle for a long time.
- The system always runs the highest priority process. If there is a tie, it runs the process that has been ready longest. Every second, it recalculates the priority and CPU usage field for every process according to the following formulas.
    - CPU usage field = CPU usage field / 2
    - Priority = CPU usage field / 2 + base priority

- In general, multilevel feedback queue schedulers are complex pieces of software that must be tuned to meet requirements.
- Anomalies and system effects associated with schedulers.
- Priority interacts with synchronization to create a really nasty effect called priority inversion. A priority inversion happens when a low-priority thread acquires a lock, then a high-priority thread tries to acquire the lock and blocks. Any middle-priority threads will prevent the low-priority thread from running and unlocking the lock. In effect, the middle-priority threads block the high-priority thread.
- How to prevent priority inversions? Use priority inheritance. Any time a thread holds a lock that other threads are waiting on, give the thread the priority of the highest-priority thread waiting to get the lock. Problem is that priority inheritance makes the scheduling algorithm less efficient and increases the overhead.
- Preemption can interact with synchronization in a multiprocessor context to create another nasty effect - the convoy effect. One thread acquires the lock, then suspends. Other threads come along, and need to acquire the lock to perform their operations. Everybody suspends until the lock that has the thread wakes up. At this point the threads are synchronized, and will convoy their way through the lock, serializing the computation. So, drives down the processor utilization.
- If have non-blocking synchronization via operations like LL/SC, don't get convoy effects caused by suspending a thread competing for access to a resource. Why not? Because threads don't hold resources and prevent other threads from accessing them.
- Similar effect when scheduling CPU and IO bound processes. Consider a FCFS algorithm with several IO bound and one CPU bound process. All of the IO bound processes execute their bursts quickly and queue up for access to the IO device. The CPU bound process then executes for a long time. During this time all of the IO bound processes have their IO requests satisfied and move back into the run queue. But they don't run - the CPU bound process is running instead - so the IO device idles. Finally, the CPU bound process gets off the CPU, and all of the IO bound processes run for a short time then queue up again for the IO devices. Result is poor utilization of IO device - it is busy for a time while it processes the IO requests, then idle while the IO bound processes wait in the run queues for their short CPU bursts. In this case an easy solution is to give IO bound processes priority over CPU bound processes.

In general, a convoy effect happens when a set of processes need to use a resource for a short time, and one process holds the resource for a long time, blocking all of the other processes. This causes poor utilization of the other resources in the system

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.

Exercise

A portable operating system is one that can be ported from one system architecture to another without any modification. Explain why it is infeasible to build an operating system that is completely portable. Describe two high-level layers that you will have in designing an operating system that is highly portable.