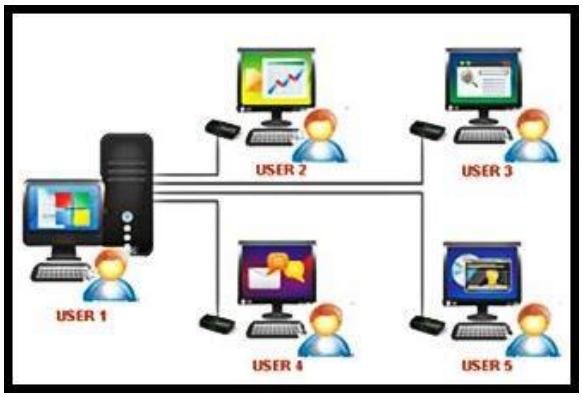
Lecture 3 – Introduction to Process Scheduling

One of the roles of an operating system is to schedule the usage of various resources among competing processes. In this section, we will focus on algorithmic approaches to scheduling the processor among the tasks that are ready to execute. We'll begin with some basic concepts of multitasking and multiprocessing and then present, evaluate and compare the performance of some simple algorithms for process scheduling, to flesh out their merits and get a grasp of scheduling objectives.

Multitasking

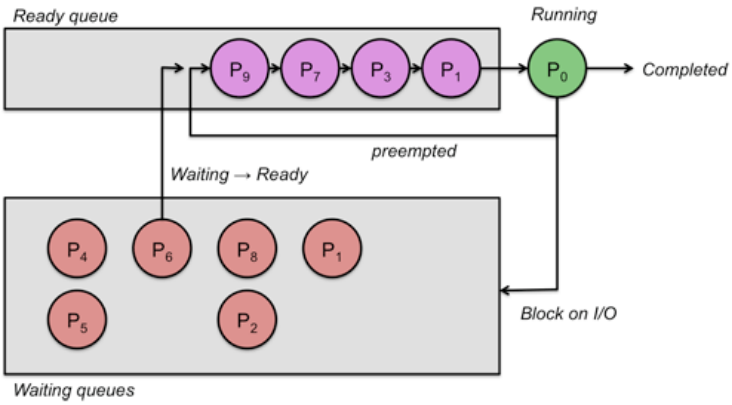
A multitasking operating system allows multiple processes of a user to be resident and active on the computer system at one time.

The purpose of this is to make better use of available resources to achieve better performance and return on investment. A sequential process can only perform one activity at a time. It might execute on the CPU, then do some I/O (disk, network etc), then use the CPU again and so on. While it is using one resource, the other resources remain idle. By overlapping it's activity with that of other processes, more resources of the computer can be used in parallel and better performance is achieved overall for the collection of processes.



In a multiuser multitasking system, some of these processes may belong to different users. When multiple users have access to the system, some kind of protection mechanism is needed to prevent one user from accessing or interfering with the processes and data of another.

So we would have a login authentication system and each user would have permissions over a subset of resources.



Process Scheduling

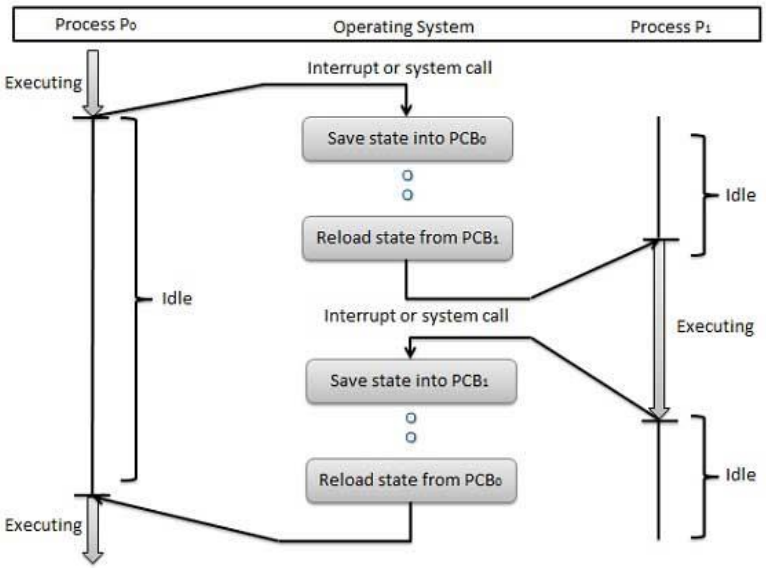
The number of processes in a multitasking system is generally much greater than the number of processors.

Each process must compete with others for the available resources.

CPU time is valuable and the operating system must decide which processes are assigned to use the processor(s) and for how long.

We saw in the last class, that when a process makes a system call or when its time quantum on the processor expires, the resulting software or hardware interrupt causes the CPU to stop fetching instructions from the currently assigned process and switch to designated code in the kernel for handling the interrupt.

It is while the operating system has control of the processor that it can make a scheduling decision. It may decide to allocate the CPU to another process, in which case it will execute code to perform a context switch.

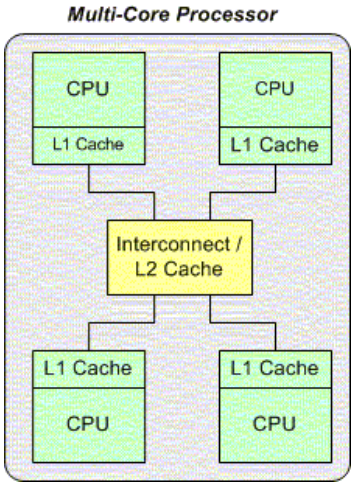


During a **context switch** operation, the run-time state of the current process is saved in the process's kernel stack space so that it can be restored and continued later and then the kernel chooses a process from the 'ready' queue and executes dispatcher code to load the run-time state of the newly chosen process for the CPU to continue executing instead. The memory space pointers of the processor will be changed to that of the new process.

In the diagram, P0 is running first when it is interrupted, its state is saved, P1 is identified and chosen from the ready queue and its state loaded into the processor. It then executes and in turn when it is interrupted, another context switch might take place to restore the original process P0 again.

The context switch operation must occur very frequently to give the illusion that all processes are progressing in parallel and to ensure a good response time for their activities.

If you are ever optimising a program to make it run faster, you should always look at the parts that are in loops or execute the most times and focus on making these parts as fast as possible. So the context switch code of the kernel must be designed to be very efficient as it is frequently used. If it was slow, then it would represent a significant overhead in terms of wasting processor time doing operating system admin work instead of executing user processes. As it is so important, the context switch operation may be supported by a special processor instruction in hardware rather than as a software algorithm involving several instructions that must be fetched and executed.



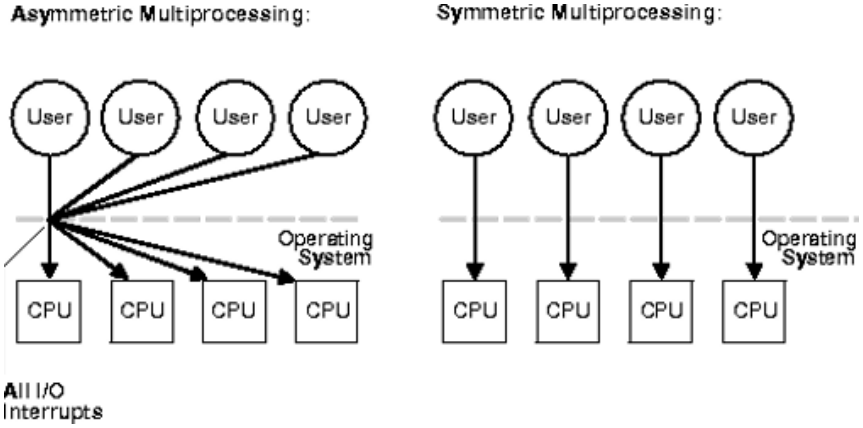
Multiprocessing

Multiprocessing is the use of more than one CPU in a system, usually separate physical CPUs but the idea may also apply to a single CPU with multiple execution cores.

You might think that doubling the number of processors would halve the execution time, but it is not as simple as that. For a start, all processors would have to be equally busy all of the time. The other issue is that the processors would have to share access to other hardware resources like the memory system and even the kernel algorithms themselves, and the contention and waiting to access these resources in turn can reduce the overall benefit of having more processors.

The benefit of increased processing power is only realised where all that processing power can be busy doing useful work in parallel. If there are any things that must be done in sequence by only one processor at a time, then this reduces performance. Imagine 50 waiters employed in a tiny restaurant with one Chef. You won't get your food 50 times faster. This is basically what is meant by Amdahl's Law.

In a mutliprocessor system, there are two simple ways of organising the processor workload.



With **asymmetric multiprocessing**, one processor, the master, centrally executes operating system code and handles all I/O operations and the assignment of workloads to the other processors (the slaves) which execute user processes. With this scheme, only one processor is accessing system data structures for resource control and scheduling.
While this makes it easier to code the operating system functions, in small systems with few processors the master may not have enough slaves to keep it busy, so maximum hardware performance is not achieved.

Symmetric multiprocessing is a system where all processors carry out similar functions and are self scheduling. The identical processors use a shared bus to connect to a single shared main memory and have full access to all I/O devices and are controlled by a single operating system instance. Each processor will examine and manipulate the operating system queue structures concurrently with others when selecting a process to execute. This access contention must be programmed carefully to protect the integrity of the shared data structures.

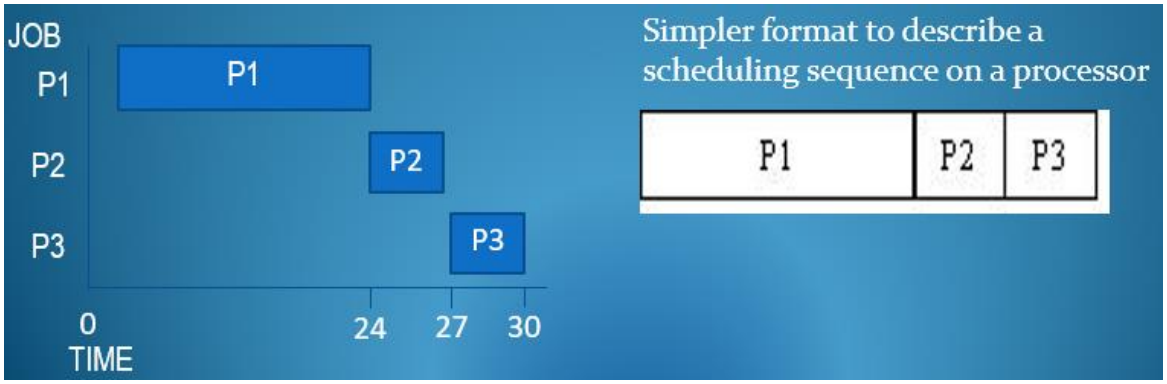
Process Scheduling Algorithms

In this section, we are going to focus on servicing the queue associated with processes waiting to be assigned to processor(s), also known as the "Ready" queue. These processes are ready for their instructions to be executed in contrast to others that may be waiting for I/O operations. Later we look at approaches to scheduling requests for reading hard disks.

Some evaluation criteria are necessary to allow us to compare different queuing policies and their effects on system performance so we define the following basic terms so we can calculate some comparative statistics for the algorithms that we look at.

Processor Utilisation = (Execution Time) / (Total Time)
Throughput = Jobs per Unit time
Turnaround Time = (Time job finishes) - (Time job was submitted)
Waiting Time = Time doing nothing in a queue
Response Time = (Time job is first scheduled on cpu) - (Time job was submitted)

The order in which processes are serviced from a queue can be described by a *Gantt Chart*, a type of bar chart that illustrates a schedule sequence.



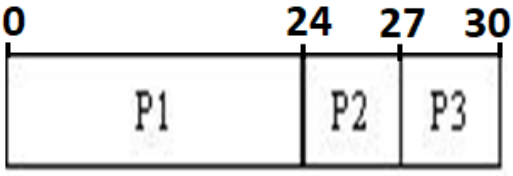
Scheduling algorithms can be ***preemptive*** or ***non-preemptive***.
All multitasking systems would employ preemptive scheduling, using a hardware interval timer to periodically generate an interrupt, as otherwise, one process might never give up control of the CPU.
In order to compare some scheduling algorithms, we will define a workload pattern and compare the performance results between different algorithms using the criteria mentioned earlier.

We'll look at two **non-preemptive** algorithms first, FCFS (First Come First Server) and SJF (Shortest Job First).

Job	CPU (Burst Time)
1	24
2	3
3	3

Consider the following, where three jobs arrive in the following order and a scheduling decision must be made:- The CPU Burst Time is the immediate period of execution time required by each task on the CPU. Note this would not normally be known in advance.

For the purposes of our study, let's say the burst time above is known in advance of the scheduling decision, in practice, a scheduler couldn't know how long a process will want to use the CPU for before scheduling it, but it could estimate.



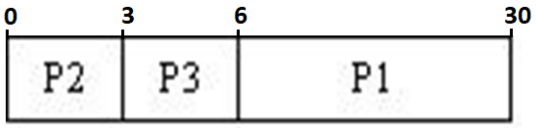
With the **FCFS algorithm**, the jobs are chosen for scheduling in the order they arrive into the system, we get the following scheduling sequence:-

FCFS gives the following performance results:-

Job	Waiting Time	Response Time	Turnaround Time
1	0	0	24
2	24	24	27
3	27	27	30
Average	17	17	27

So job P1 didn't have to wait at all, it's response time was also 0 as it is capable of generating responses once it receives the CPU at time 0, and it took 24 time units to execute. P2 had to wait for P1 to finish, so it's ability to respond and its overall turnaround time were severely impacted by sitting in the ready queue waiting for the long job ahead of it to finish. Similarly for P3 which was waiting the longest for service and finished last.

Using the **Shortest Job First** (SJF algorithm instead) we get the following order:



Giving the following performance evaluation:-

Job	Waiting Time	Response Time	Turnaround Time
1	6	6	30
2	0	0	3
3	3	3	6
Average	3	3	13

So the long job P1 was executed last and had to wait 6 time units in the queue for the other two jobs to finish. The overall turnaround times for all three jobs is the same, it's the same amount of work, just done in a different sequence. Notice the average response time for the three tasks is much improved. In fact, SJF gives the minimal response time for this set of tasks.

Analysis of non-preemptive algorithms used

FCFS is an inherently **fair** algorithm but **performs badly** for interactive systems where the response time is important and in situations where the job length differs greatly.

SJF is the **optimal** algorithm in this situation in terms of average throughput, waiting time and response performance but is **not fair**.

SJF favours short jobs over longer ones. The arrival of shorter jobs in the ready queue postpones the scheduling of the longer ones **indefinitely** even though they may be in the ready queue for quite some time. This is known as **starvation**.

FCFS is an easy algorithm to implement in practice, you just add new tasks to the back of a first come first served queue and take the next from the front of the queue. There are no complicated calculations or comparisons involved.

In the case of SJF however, we need future knowledge of each tasks CPU burst time in order to choose the shortest for each scheduling decision.

Estimating CPU Burst Length for SJF

In practice it is not possible to know the CPU burst length of jobs in advance in a general purpose system, but this can be estimated using a smoothing formula (a heuristic guess) based on a task's historical performance as follows:-

So say a task had a CPU burst of 10, then it did some I/O, then another burst of 6 followed by more I/O followed by a third burst of 5. We could calculate the average length of its bursts which would be (10+6+5)/3 = 7, and we might say that we estimate its next burst will be 7 to enable SJF to make a choice.
Or we might say, the long term history is not relevant, we are only interested in what it did last which was a burst of 5, so we might estimate its next burst will also be 5.
Or we could give weighting to both, so maybe half of 7 plus half of 5, giving us a weighted average of 6 for its next burst. The weighting doesn't have to be exactly half, we could favour history over recent usage etc by changing the multiplier.

So in general,

Let t_n = length of burst n
Let T_n = average of previous n bursts
The next burst t_{n+1} of a process may be estimated from the formula:-

$$t_{n+1} = at_n + (1-a)T_n$$

where $0 <= a <= 1$
Where a is a chosen weighted value based on recent activity and average historical activity.

Choosing a process using the SJF calculation based on approximated CPU burst is more complex than FCFS.
You must maintain cumulative history information and perform the calculations required for predicting burst length each time you come to choose the next task.

Need For Other Algorithms

With FCFS, long jobs hold up short jobs.
While SJF solves this, a continual stream of short jobs will block long jobs indefinitely and all tasks in a system should be able to progress at some point.
It may be a better idea that the longer a job is in the system waiting for the CPU, the greater chance it has of being scheduled. Can we build a waiting time into the formula for choosing the next job so that jobs increase in priority the longer they wait and will eventually get serviced.

The **(HRN) highest response ratio next** algorithm endeavors to meet this objective. It is a type of priority based algorithm.

The Response Ratio of a job is calculated as follows:-
Response Ratio = (Waiting Time) / (Service Time)

The Response Ratio determines the ordering of the jobs. As the job waits in the ready queue, its priority increases. When the processor becomes free, the scheduler chooses the task with the Highest (Waiting Time)/(Service Time) ratio.

We will look at an example of this in practice in the next class.