Process Scheduling Algorithms Simulation

This project's main aim is to simulate the following 8 processes:

1. First Come First Serve
2. Shortest Job First Preemptive
3. Shortest Job First Non-Preemptive
4. Priority Preemptive
5. Priority Non-Preemptive
6. Round Robin
7. Multi-level queue
8. Multilevel feedback queue

All of the above processes have been coded in the language C++ and the graph related simulation is done in Python.

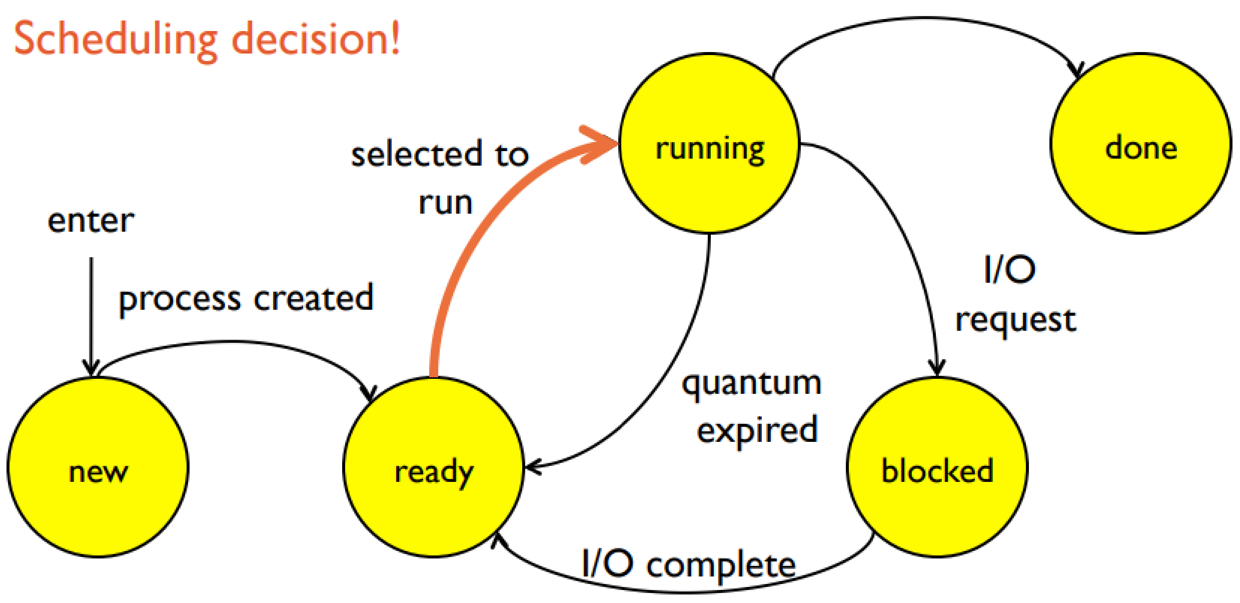
The project has been completed by:

1. Mohit Sharma (2022201060)
2. Bhanuj Gandhi (2022201068)
3. Avishek Kumar Sharma (2022202024)
4. Yash Singhal (2022201004)

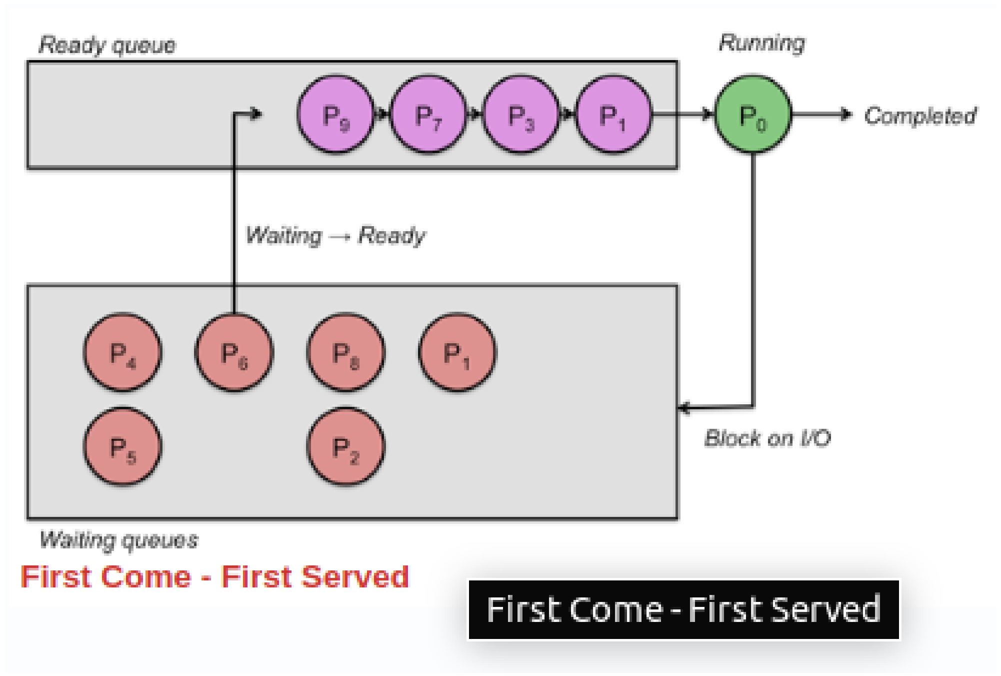
Introduction

The simulations assumes the processes to be in 5 states:

1. NEW - The process is being created, and has not yet begun executing.
2. READY - The process is ready to execute, and is waiting to be scheduled on a CPU.
3. RUNNING - The process is currently executing on a CPU.
4. WAITING - The process has temporarily stopped executing, and is waiting on an I/O request to complete.
5. TERMINATED - The process has completed.



FCFS (First come First Serve - Preemptive and non Preemptive):



General Overview:

Possibly the most straightforward approach to scheduling processes is to maintain a FIFO (first-in, first-out) run queue. In this,new processes go to the end of the queue. When the scheduler needs to run a process, it picks the process that is at the head of the queue. This scheduler is non-preemptive. If the process has to block on I/O, it enters the *waiting* state and the scheduler picks the process from the head of the queue. When I/O is complete and that waiting (blocked) process is ready to run again, it gets put at the end of the queue.

With first-come, first-served scheduling, a process with a long CPU burst will hold up other processes, increasing their turnaround time. Moreover, it can hurt overall throughput since I/O on processes in the *waiting* state may complete while the CPU bound process is still running.

Now devices are not being used effectively. To increase throughput, it would have been great if the scheduler instead could have briefly run some I/O bound process so that could run briefly, request some I/O and then wait for that I/O to complete. Because CPU bound processes don't get preempted, they hurt interactive

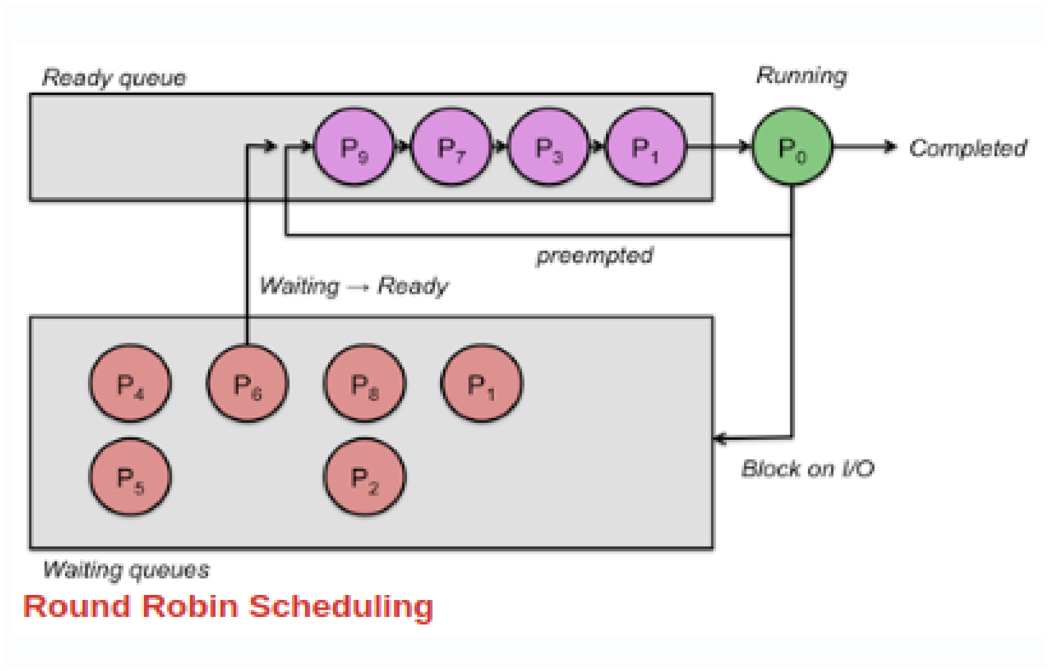
performance because the interactive process won't get scheduled until the CPU bound one has completed.
Advantage: FIFO scheduling is simple to implement. It is also intuitively fair (the first one in line gets to run first).

Disadvantage: The greatest drawback of first-come, first-served scheduling is that it is not preemptive. Because of this, it is not suitable for interactive jobs. Another drawback is that a long-running process will delay all jobs behind it.

Implementation:

- A process PCB is created for each process with all the required fields that are used to simulate the FCFS.
- A normal queue data structure is used in the implementation, where all the incoming processes are sorted in increasing order.
- Then, each process is dequeued and scheduled for the burst_time1 only.
- Assuming infinite I/O devices the io_time is handled and then burst_time2 is scheduled again in the same increasing arrival time fashion.
- The I/O-bound processes have been given higher priorities than the CPU-bound processes.
- The PCB is created using struct data structure.
- The details such as: Turn around time, Response time, Waiting time etc are calculated and updated in the PCB itself.

<u>Round Robin Scheduling:</u>



General Overview:

Round robin scheduling is a preemptive version of first-come, first-served scheduling. Processes are dispatched in a first-in-first-out sequence but each process is allowed to run for only a limited amount of time. This time interval is known as a time-slice or quantum. If a process does not complete or get blocked because of an I/O operation within the time slice, the time slice expires and the process is preempted. This preempted process is placed at the back of the run queue where it must wait for all the processes that were already in the queue to cycle through the CPU.
If a process gets blocked due to an I/O operation before its time slice expires, it is, of course, enters a blocked because of that I/O operation. Once that operation completes, it is placed on the end of the run queue and waits its turn. A big advantage of round robin scheduling over non-preemptive schedulers is that it dramatically improves average response times. By limiting each task to a certain amount of time, the operating system can ensure that it can cycle through all ready tasks, giving each one a chance to run.

With round robin scheduling, interactive performance depends on the length of the quantum and the number of processes in the run queue. A very long quantum makes the algorithm behave very much like first come, first served scheduling since it's very likely that a process with block or complete before the time slice is up. A small quantum lets the system cycle through processes quickly. This is wonderful for interactive processes. Unfortunately, there is an overhead to context switching and having to do so frequently increases the percentage of system time that is used on context switching rather than real work.

Advantage: Round robin scheduling is fair in that every process gets an equal share of the CPU. It is easy to implement and, if we know the number of processes on the run queue, we can know the worst-case response time for a process.

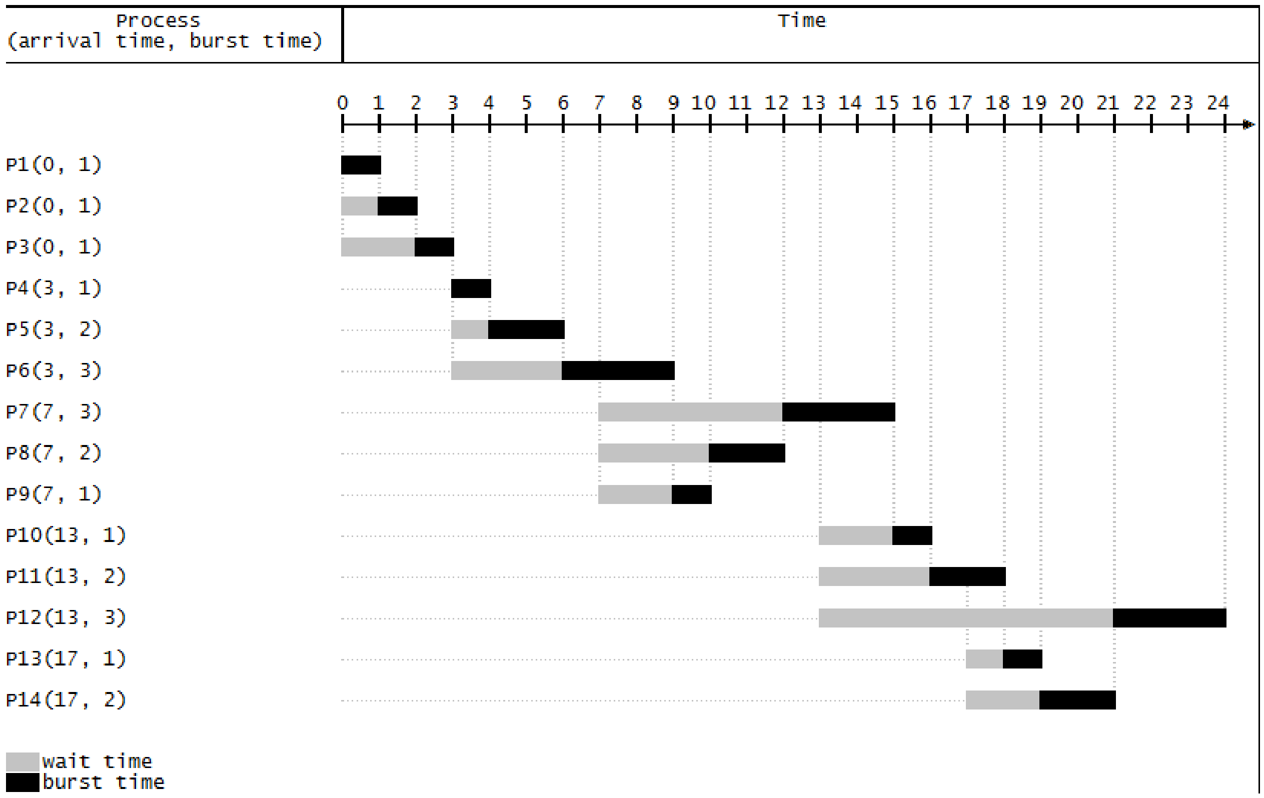
Disadvantage: Giving every process an equal share of the CPU is not always a good idea. For instance, highly interactive processes will get scheduled no more frequently than CPU-bound processes.

Implementation:

- The time slice for the Round Robin is asked as an input from the user.
- A process PCB is created for each process with all the required fields that are used to simulate the Round Robin.
- A Round Robin queue is maintained in which processes are enqueued according to the algorithm.
- As the assigned process' time slice runs out, it is popped from the running queue and pushed into this Round Robin queue.
- As burst_time1 runs out, assuming infinite I/O devices the io_time is handled and then burst_time2 is scheduled again in the same enqueueing into the queue fashion.
- The PCB is created using struct data structure.
- The details such as: Turn around time, Response time, Waiting time etc are calculated and updated in the PCB itself.

Shortest Job First:

General Overview:



Shortest job first is a scheduling algorithm in which the process with the smallest execution time is selected for execution next. Shortest job first can be either preemptive or non-preemptive. Owing to its simple nature, the shortest job first is considered optimal. It also reduces the average waiting time for other processes awaiting execution.

Shortest job first depends on the average running time of the processes. The accurate estimates of these measures help in the implementation of the shortest job first in an environment, which otherwise makes the same nearly impossible to implement. This is because often the execution burst of processes does not happen beforehand. It can be used in interactive environments where past patterns are available to determine the average time between the waiting time and the commands. Although it is disadvantageous to use the shortest-job-first concept in short-term CPU scheduling, it is considered highly advantageous in long-term CPU scheduling. Moreover, the throughput is high in the case of the shortest job first.

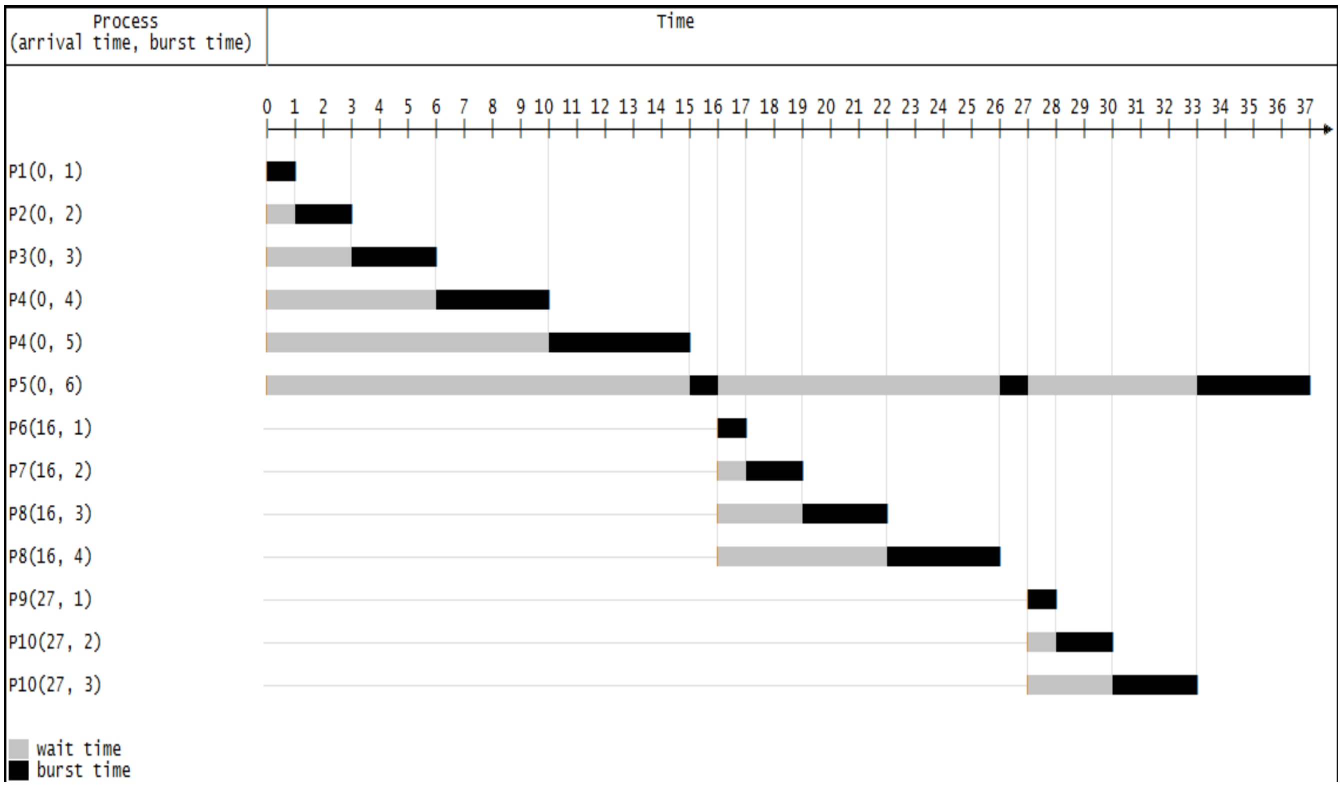
Advantage: SJF is very easy to implement and provides one of the lowest waiting times among all other scheduling processes.

Disadvantage: It can cause process starvation for longer jobs if there are a large number of shorter processes. Another is the need to know the execution time for each process beforehand.

Implementation:

- A process PCB is created for each process with all the required fields that are used to simulate the Shortest job First.
- The implementation details of SJF and FCFS are almost identical in many ways such as - they are not preemptive, the scheduling is done on the basis of a time parameter (in sorted order).
- The only difference lies in the fact that SJF is based on the current burst time and is non preemptive, whereas FCFS is based solely on the arrival times.
- The I/O-bound processes have been given higher priorities than the CPU-bound processes.
- The PCB is created using struct data structure.
- The details such as: Turn around time, Response time, Waiting time etc are calculated and updated in the PCB itself.

Shortest Remaining Time First:



General Overview:

Shortest remaining time, also known as shortest remaining time first (SRTF), is a scheduling method that is a preemptive version of SJF scheduling. In this scheduling algorithm, the process with the smallest amount of time remaining until completion is selected to execute. Since the currently executing process is the one with the shortest amount of time remaining by definition, and since that time should only reduce as execution progresses, the process will either run until it completes or get preempted if a new process is added that requires a smaller amount of time.

Shortest remaining time is advantageous because short processes are handled very quickly. The system also requires very little overhead since it only makes a decision when a process completes or a new process is added, and when a new process is added the algorithm only needs to compare the currently executing process with the new process, ignoring all other processes currently waiting to execute.

Advantage: It has been observed that the SRTF algorithm has a superior Turn Around Time than the SJF as it always attends to the lowest time job present in the system at that moment.

Disadvantage: The context switch is done a lot more times in SRTF than in SJN, and consumes CPU's valuable time for processing.

Implementation:

- A process PCB is created for each process with all the required fields that are used to simulate the Shortest job First.
- SRTF is essentially a preemptive SJF algorithm.
- To get the next time point at which a new process might be available for consideration, there are 2 priority queues implemented - one sorted in the arrival time and second sorted on the burst time.
- Using these 2 priority queues, we get one schedulable process at a time and the burst time gets exhausted with the same amount as the time duration (that is calculated).
- The I/O-bound processes have been given higher priorities than the CPU-bound processes.
- The PCB is created using struct data structure.
- The details such as: Turn around time, Response time, Waiting time etc are calculated and updated in the PCB itself.

Priority Scheduling (Preemptive and non-Preemptive)

General Overview:

The idea here is that each process is assigned a priority (just a number). Of all processes ready to run, the one with the highest priority gets to run next (there is no general agreement across operating systems whether a high number represents a high or low priority; UNIX-derived systems tend to use smaller numbers for high priorities while Microsoft systems tend to use higher numbers for high priorities).With a priority scheduler, the scheduler simply picks the highest priority process to run.

If the system uses preemptive scheduling, a process is preempted whenever a higher priority process is available in the run queue.Priorities may be internal or external. Internal priorities are determined by the system using factors such as time limits, a process' memory requirements, its anticipated I/O to CPU ratio, and any other system-related factors. External priorities are assigned by administrators.

Priorities may also be static or dynamic. A process with a static priority keeps that priority for the entire life of the process. A process with a dynamic priority will have that priority changed by the scheduler during its course of execution. The scheduler would do this to achieve its scheduling goals. For example, the scheduler may decide to decrease a process' priority to give a chance for a lower-priority job to run. If a process is I/O bound (spending most if its time waiting on I/O), the scheduler may give it a higher priority so that it can get off the run queue quickly and schedule another I/O operation.

Static and dynamic priorities can coexist. A scheduler would know that a process with a static priority cannot have its priority adjusted throughout the course of its execution. Ignoring dynamic priorities, the priority scheduling algorithm is straightforward: each process has a priority number assigned to it and the scheduler simply picks the process with the highest priority.

An example of priority Scheduling in simple terms can be understood as -

1. FCFS is a non preemptive priority scheduling algorithm in which the process with lower arrival time is given the higher priority.
2. SRTF is a preemptive scheduling algorithm in which the process with lower burst time (at that instant) is given the higher priority.
3.

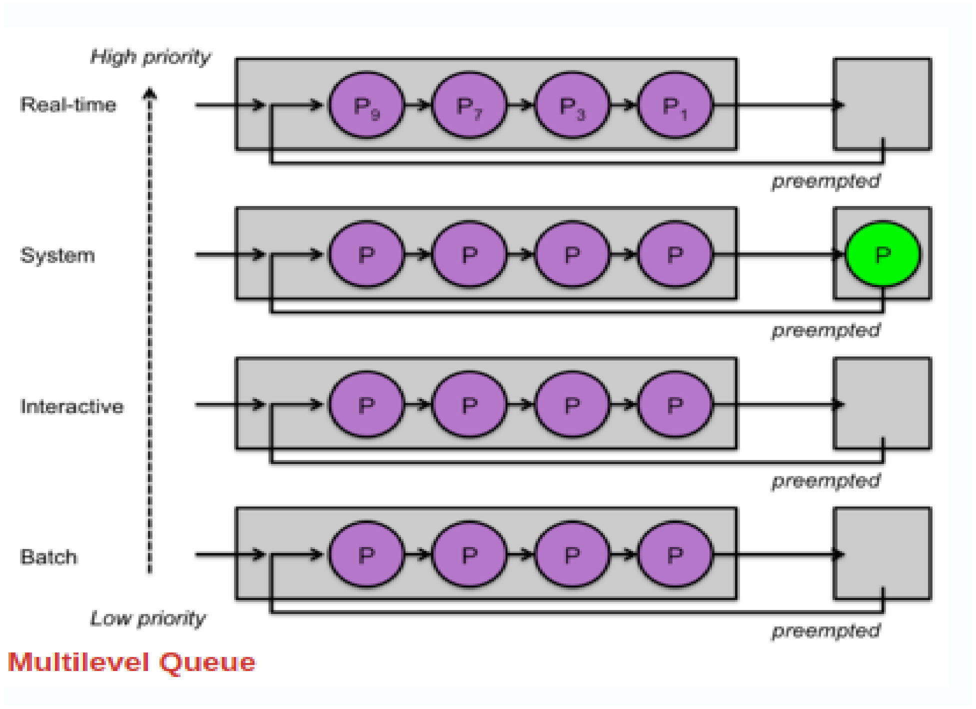
Advantage: Priority scheduling provides a good mechanism where the relative importance of each process may be precisely defined.

Disadvantage: If high priority processes use up a lot of CPU time, lower priority processes may starve and be postponed indefinitely, leading to starvation. Another problem with priority scheduling is deciding which process gets which priority level assigned to it.

Implementation:

- A process PCB is created for each process with all the required fields that are used to simulate the Shortest job First.
- In our implementation, the PCB stores priority for each process and whenever the priority is not required, it just stores 0 as the priority has to be mentioned by the user.
- The I/O-bound processes have been given higher priorities than the CPU-bound processes - the first level of priority.
- Similar to the SRTF implementation, to get the next time point at which a new process might be available for consideration, there are 2 priority queues implemented - one sorted in the arrival time and second sorted on the priorities.
- As the name suggests, in the preemptive variation, we always look at the time instant and select the process available with the highest priority.
- And in the non preemptive variation every schedulable entity runs out its burst time1 and then burst time2.
- The details such as: Turn around time, Response time, Waiting time etc are calculated and updated in the PCB itself.

<u>Multi-level queue scheduling</u>



General Overview:

We can group processes into priority classes and assign a separate run queue for each class. This allows us to categorize and separate system processes, interactive processes, low-priority interactive processes, and background non-interactive processes. The scheduler picks the highest-priority queue (class) that has at least one process in it. In this sense, it behaves like a priority scheduler.

Each queue may use a different scheduling algorithm, if desired. Round-robin scheduling per priority level is the most common. As long as processes are ready in a high priority queue, the scheduler will let each run for their time slice. Only when no processes are available to run at that priority level will the scheduler look at lower levels. Alternatively, some very high priority levels might implement the non-preemptive first-come, first-served scheduling approach to ensure that a critical real-time task gets all the processing it needs.

The scheduler may also choose a different quantum for each priority level. For example, it is common to give low-priority non-interactive processes a longer quantum. They won't get to run as often since they are in a low priority queue but, when they do, the scheduler will let them run longer.

Multi-level queues are generally used as a top-level scheduling discipline to separate broad classes of processes, such as real-time, kernel threads, interactive, and background processes. Specific schedulers within each class determine which process gets to run within that class.

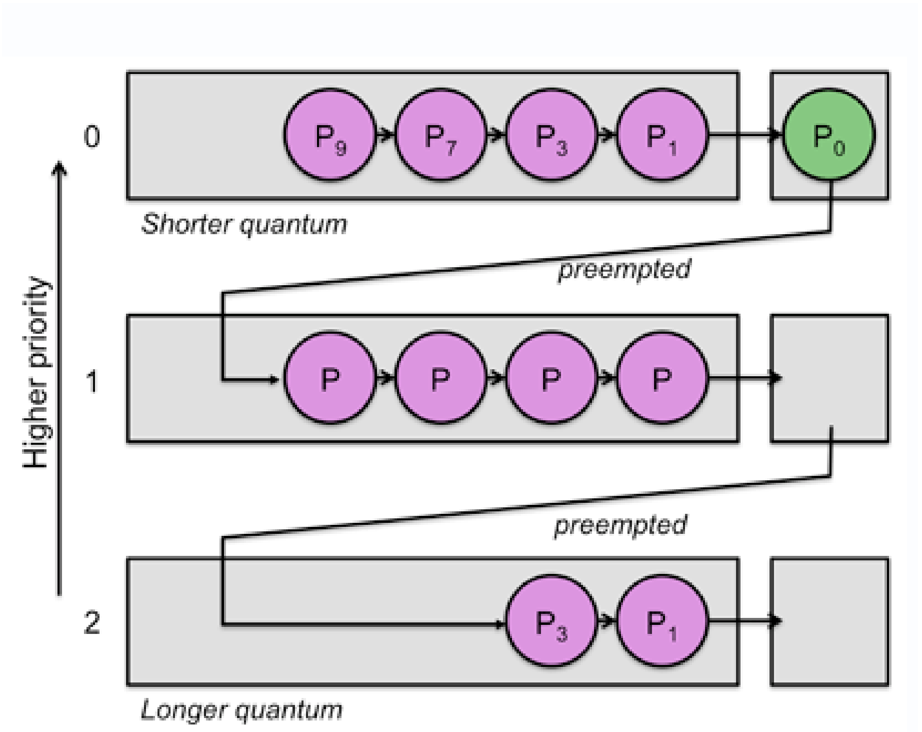
Advantage: As MLQ assigns permanent queue to the processes therefore, it has an advantage of low scheduling overhead.Also, We can use MLQ to apply different scheduling methods to distinct processes.

Disadvantage: Some processes may face starvation as higher priority queues are never becoming empty. Also, MLQ is inexorable in nature.

Implementation:

- A process PCB is created for each process with all the required fields that are used to simulate the Shortest job First.
- In our implementation, there are 2 queues - Interactive queue and background queue.
- Interactive queue is assigned a 70% time slice and the background queue is assigned a 30% time slice.
- The I/O-bound processes have been given higher priorities than the CPU-bound processes.
- For example - if we have 10 seconds then we give 7 seconds to Interactive queue process and then 3 seconds to background queue process.
- As the context switch takes place, we pop the process from the Running queue and add it to the Ready queue.
- The details such as: Turn around time, Response time, Waiting time etc are calculated and updated in the PCB itself.

Multi-level Feedback queue scheduling



Multilevel Feedback Queue

General Overview:

A variation on multilevel queues is to allow the scheduler to adjust the priority (that is, use *dynamic priorities*) of a process during execution in order to move it from one queue to another based on the recent behavior of the process.
The goal of multilevel feedback queues is to automatically place processes into priority levels based on their CPU burst behavior. I/O-intensive processes will end up on higher priority queues and CPU-intensive processes will end up on low priority queues.

A multilevel feedback queue uses two basic rules:

1. A new process gets placed in the highest priority queue.
2. If a process does not finish its quantum (that is, it blocks on I/O) then it will stay at the same priority level (round robin) otherwise it moves to the next lower priority level

With this approach, a process with long CPU bursts will use its entire time slice, get preempted and get placed in a lower-priority queue. A highly interactive process will not use up its quantum and will remain at a high priority level.

Although not strictly necessary for the algorithm, each successive lower-priority queue may be given a longer quantum. This allows a process to remain in the queue that corresponds to its longest CPU burst.

Implementation:

- A process PCB is created for each process with all the required fields that are used to simulate the Shortest job First.
- In our implementation, there are 3 queues - first queue, second queue and FCFS queue.
- The first and second queue follow the Round Robin algorithm whereas the last queue follows the First Come First Serve algorithm (FCFS) .
- The time quantum for the first queue is 4 and for the second queue, it is 8 units of time quantum.
- If the process doesn't run out, the last queue follows the FCFS algorithm.
- As the context switch takes place, we pop the process from the Running queue and add it to the Ready queue.
- The details such as: Turn around time, Response time, Waiting time etc are calculated and updated in the PCB itself.

Question for Round Robin

In a real OS, the shortest timeslice possible is usually not the best choice. Why not?
It takes some time to change the context between two tasks. Let's assume that a given system has two runnable tasks. Every context switch will occur in around the same amount of time, say 0.1ms. Each job is given 9.9ms to perform, thus if context switches happen every 10ms, 99% of each 1ms interval is used for task execution and 1% is used for context shifts. Context transitions waste 10% of the time if we increase the frequency to once every millisecond. In the worst case scenario, if we increase again to have a context transition every 0.1 ms, then the entire processor time is spent in context switches, leaving no time for the apps to run.

Therefore, it is ideal to have context transitions as infrequently as feasible for performance. This lengthens the time that can be used to execute applications However, if context shifts happen infrequently enough, there will be a lag before a programme responds to an outside event. This is not an issue for purely computational programmes, but it is an issue for input/output programmes. A network server may take up to one second to respond (or more if there are more than two tasks), an interactive programme may take up to one second to react to a keypress, etc., whereas computational tasks don't waste time when context switches only happen once every second.

A balance between computing performance and reactivity must therefore be struck by the operating system scheduler. I/O-bound jobs are given shorter, more frequent time slices by sophisticated schedulers, whereas computational tasks are given longer, less frequent time slices. These schedulers attempt to identify which tasks are computational and which ones are I/O-bound.