# Scheduling Algorithms

**List Pros and Cons for each of the four scheduler types listed below.**

First In First Out (FIFO)
Pros:
***Simplicity –*** *FIFO is very easy to implement.*
***Less Overhead*** *– FIFO will allow the currently running task to complete its CPU burst, which means that there is no need to preemptively take the processor away from the task and then context-switch.*

Cons:
***Convoy Effect*** *– If a large task grabs the processor, then subsequent tasks that arrive must wait for this large task to complete. This means that short tasks could be stuck waiting for the long task to complete; thus, our "task throughput" could potentially be very low.*

Round Robin Scheduler (RR):
Pros:
***Fair to small tasks*** *– Each task gets a fixed time slice (or quantum) of the CPU before it is forced to give up the processor. Thus, unlike the FIFO case, even if a lot of small tasks enter the system after a large task has entered, everyone gets to run and the small tasks will be able to make progress; thus, this scheduler is much "fairer" to short jobs.*

Cons:
***Context-Switch Overhead*** *– CPU cycles are now consumed when a task exceeds its allotted time slice: The currently running task must be taken off the processor, and the next chose task must be put on; thus, context-switching reduces the overall CPU utilization.*

***No gain for tasks with equal run times –*** *If we have ten tasks in the system which take ten seconds to run, then we will not have a single completed task until 100 seconds have passed. In the FIFO case, we would have our first task complete in 10 seconds, the second task complete in 20 seconds, etc. As a result, the "task throughput" is potentially lower.*

Shortest Job First (SJF):
Pros:
***Maximizes "task throughput" –*** *By running tasks which take less time to complete first, we can complete more tasks in a given amount of time.*

Cons:
***Unfair to larger tasks –*** *Larger tasks do not get an opportunity to run if smaller tasks keep entering the system.*

***Computers are not psychic –*** *It is not feasible to accurately predict how long a task's CPU burst-time could be.*

Shortest Remaining Time First (SRTF):
Pros:
*Similar to that of SJF, except that if we are currently running a task, and a smaller task arrives, then we pre-empt the currently running task in order to complete the smaller task. Therefore, this scheduling algorithm will have an even higher "task throughput" rate than SJF.*

*Considered to be the "optimal" scheduling policy – provides an upper bound on performance.*

Cons:
*Similar to SJF, except that now there is also context-switching overhead incurred for pre-empting a currently running task in favor of a shorter one.*

**True or False: Operating Systems which want a fast response time shouldn't use SRTF.**
***False.*** *Operating Systems which want a fast response time *should* use Shortest Remaining Time First.*

*They key thing to remember is that tasks which are I/O bound (and therefore not very CPU intensive) will tend to have shorter CPU burst times. For example, when typing information into Notepad (or vim, or whatever other editor you use), the CPU use time is minimal as most of the task's time is spent blocking on key strokes. Therefore, we want I/O-related tasks to complete their CPU burst as quickly as possible and be able to wait for further input from the user. In order to accomplish this, SRTF should be used.*

*On the other hand, tasks which *are* CPU intensive (such as a Video Encoder) will get lower priority with respect to these I/O Bound tasks. But this is acceptable because while the user won't care if the video encoding starts half a second late, he will care if he is playing CounterStrike and it takes half a second for a mouse click to register on the screen.*

**True or False: Lottery scheduling can be used to implement any other scheduling algorithm.**
***False.*** *The sheer randomness of the Lottery Scheduler makes it impossible to perfectly implement any other scheduling algorithm. For example, we could not emulate priority starvation of the Priority Scheduler if we tried, just because a low priority thread has a small probability of actually running.*

**Explain what a multi-level feedback scheduler is and why it approximates SRTF.**
*A multi-level feedback scheduler stores processes into different "tiers." Processes that exceed the allotted CPU time slice are automatically moved down to a lower "tier," while processes that make I/O requests or block will be moved to higher "tiers." The multi-level feedback scheduler will favor processes in the "higher" tiers before focusing on those in the lower tiers. Thus, processes that tend to depend a lot on I/O (and therefore dictate the response time of the system) will reside in higher tiers and have priority over processes that are strictly computational (and therefore reside in the lower tiers).*

*This does not mean that CPU-intensive processes do not get to run. When processes block on I/O, they are no longer in the RUNNABLE state, and it is at this point that the CPU-intensive processes (being the only available RUNNABLE processes in the system) will be able to make progress. Some systems also choose to allocate some CPU time to the lower-tiers, even when there are processes in the higher-tiers (of course, the percentage allocation for the lower-tiers is much lower than that allocated to the higher-tiers)*

**True or False: If a user knows the length of a CPU time-slice and can determine precisely how long his process has been running, then he can cheat a multi-level feedback scheduler**

***True.*** *Just before the user's time-slice expires, he or she can execute a dummy I/O request that makes the process appear to be "I/O Bound." Doing so allows the user to keep the task in the upper tier of the multi-level feedback scheduler, ensuring that he gets more CPU time than a CPU-intensive process should.*

*Note that today's Linux scheduler is more complicated than what we have discussed in class and guards against such abuse. This is accomplished by keeping track of how long the process sleeps vs. how long the process is active. This means that a process that blocks on a lot of I/O *and* uses a lot of CPU time will be regarded as "neutral" vs. a process that blocks on a lot of I/O and uses very little CPU time (which gets boosted up) vs. a process that spends little time on I/O and uses a lot of CPU time (which would get boosted down).*

***Consult Fall 2006 Midterm 1 (Problem 4e) for the solutions to the remaining problems. Note that I inadvertently neglected to include a row for Time Slot 0 (on the questions handout) which confused some students.***

Here is a table of processes and their associated arrival and running times.

| PID | Arrival Time | CPU Running Time |
|---|---|---|
| 1 | 0 | 2 |
| 2 | 1 | 6 |
| 3 | 4 | 1 |
| 4 | 7 | 4 |
| 5 | 8 | 3 |

**Show the scheduling order for these processes under the specified policies. The processer has a time-slice of 1. Assume no context-switch overhead and that new processes are added to the head of the queue except for FCFS, where they are added to the tail.**

| Time Slot | FIFO | SRTF | RR |
|---|---|---|---|
| 0 | | | |
| 1 | | | |
| 2 | | | |
| 3 | | | |
| 4 | | | |
| 5 | | | |
| 6 | | | |
| 7 | | | |
| 8 | | | |
| 9 | | | |
| 10 | | | |
| 11 | | | |
| 12 | | | |
| 13 | | | |
| 14 | | | |
| 15 | | | |

**Specify the average wait time and total run time for each process:**

| Scheduler | Process 1 | Process 2 | Process 3 | Process 4 | Process 5 |
|---|---|---|---|---|---|
| FIFO Wait | | | | | |
| FIFO TRT | | | | | |
| SRTF Wait | | | | | |
| SRTF TRT | | | | | |
| RR Wait | | | | | |
| RR TRT | | | | | |

**Also specify the average wait time and total run time for each type of scheduler:**

**What would be the optimal average wait time, and which of the above three schedulers would come closest to optimal? Explain.**
*SRTF is the optimal scheduler in this situation since it allows the most tasks to be completed in a fixed amount of time.*