1. CPU-bound processes are better supported by algorithms that lack implementation for preemptions. While utilizing CPU scheduling algorithms with preemptions can provide many benefits, they will interrupt CPU-bound processes with long burst times and as a result cause these processes to take much longer to complete. Non-preemptive algorithms allow for every process that starts using the CPU to finish using it until it has completed its CPU burst, preventing the processes from acquiring lengthy turnaround times. An example of this is shown in Figure 1, where process N is preempted in Shortest Remaining Time (SRT), a preemptive algorithm, and terminates later than in First Come First Serve (FCFS), a non-preemptive algorithm. Shortest Job First (SJF), another non-preemptive algorithm, sorts the queue by placing the jobs with the shortest CPU burst times in the front which are usually I/0-bound processes. This isn’t necessarily bad for CPU-bound processes but it could take longer for the CPU-bound processes to enter the running state. Therefore, FCFS is the best algorithm for the CPU-bound processes.

I/O-bound processes, on the other hand, are better supported by algorithms that utilize preemptions. I/O-bound processes usually have much shorted CPU burst times than CPU-bound processes and since preemptions kick out processes with longer running burst times for ones with shorter burst times, the I/O-bound processes can be completed much quicker. Process A preempts several other processes in SRT and as a result is able to terminate earlier than in FCFS (Figure 2). While Round Robin (RR) is dedicated to being fair to both CPU-bound and I/O-bound processes, SRT focuses on completing the processes with the shorter burst times thus making it the superior choice in algorithm for I/O bound processes.

1. Changing rr\_add from END to BEGINNING can create various changes in the average turnaround time, average wait time, CPU utilization, total number of context switches, and total number of preemptions. This is because the results are heavily dependent on the order processes arrive. Whenever a process is kicked out of the running state it is pushed on to the front of the queue again, where it is immediately put back into the running state (Figure 3). This process will keep repeating until the process completes its CPU burst or terminates. Therefore the algorithm basically becomes Last Come First Serve (LCFS). Overall, this is definitely not better than the traditional RR.
2. Obviously, since SRT is a preemptive algorithm while SJF is not, changing to SRT will result in more context switches from the preemptions and which thus causes a longer turnaround time. However the largest impact that comes from the change is in the response time of the processes. In Figure 4, process C using SJF terminated at 4113ms while using SRT it terminated at 3643ms. Also in Figure 5, process I terminated at 76256ms using SJF while it terminated at 77148ms using SRT. SRT kicks processes with longer CPU burst times for processes with shorter CPU burst times so it makes sense that process C, which has 4 CPU bursts, terminates faster in SRT than in SJT, while process I, which has 77 CPU bursts, takes longer to terminate in SRT than in SJT.
3. One limitation of the simulation is the way that ties are handled. Another is the lack of an implementation of an algorithm for an I/O queue. Another is that a real world operating system would be more likely to utilize multiple of these algorithms at once to make up for any faults just one of them has alone.
4. A different priority scheduling algorithm could be created using elements from the ones used in the simulation along with a few new ideas. The priority would be calculated from a series of important factors: arrival time, CPU burst time, the time a process has been in the queue, an rr\_add value, and a time slice value. The priority of processes will be determined using a similar fashion to RR and due to its similarities it shall be referred to as Round Robin Two (RR2). Initially, RR2 will function the same as the original RR, with adding new processes to the end to the queue (rr\_add = END). However, as more processes get added to the queue, the ratio of I/O-bound processes to CPU-bound will change and the values of the time slice and rr\_add will change accordingly. The time slice value will become whatever the average CPU burst time is in the ready queue. Using this method, the time slice value will always reflect what type of processes the majority of the queue contains. If the queue is mostly CPU-bound process, then the time slice value will be much larger, making the algorithm similar to FCFS which is more catered toward CPU-bound processes. If the CPU burst time average of the queue decreases passed a certain threshold, which could be the average CPU burst times of all the processes sent from “birth”, then the rr\_add value changes to BEGINNING. Now if the queue is mostly I/O-bound process, then the time slice value will be much smaller, making the rr\_add change to BEGINNING and the algorithm similar to SRT which is more catered toward I/O-bound processes. Each time a process is added to the queue or removed from the queue, this average is recalculated. Once the average is above the threshold, then rr\_add changes back to END. To prevent any processes from starving or facing indefinite blocking while rr\_add is BEGINNING, aging will also be implemented. Each process in the queue will have an integer “age” value. Each time a new process is inserted in the queue ahead of a process, it adds 1 to the age counter. Once the age value is 10, no new process can be inserted in front of that process. This will help balance out which processes the algorithm catering toward. There are many advantages to RR2. Processes can’t stave and it is relatively fair to CPU-bound and I/O bound processes. Some additional advantages that this algorithm has over the original RR is that it can actually become custom to the user since the threshold value used to change the rr\_add value comes from the average of all CPU burst times from process run on the CPU and it adapts to the type of processes that dominate the ready queue. Some disadvantages of RR2 overlap with RR. It has long average wait times and more overheads than the other algorithms. Unfortunately, a pretty major disadvantage is that we have to predict CPU burst times.