Operating Systems Group Project Final Report

CSCI460 Fall 2019 Final Project

Team Members:

Rusty Clayton, Rial Johnson, Tim Parrish

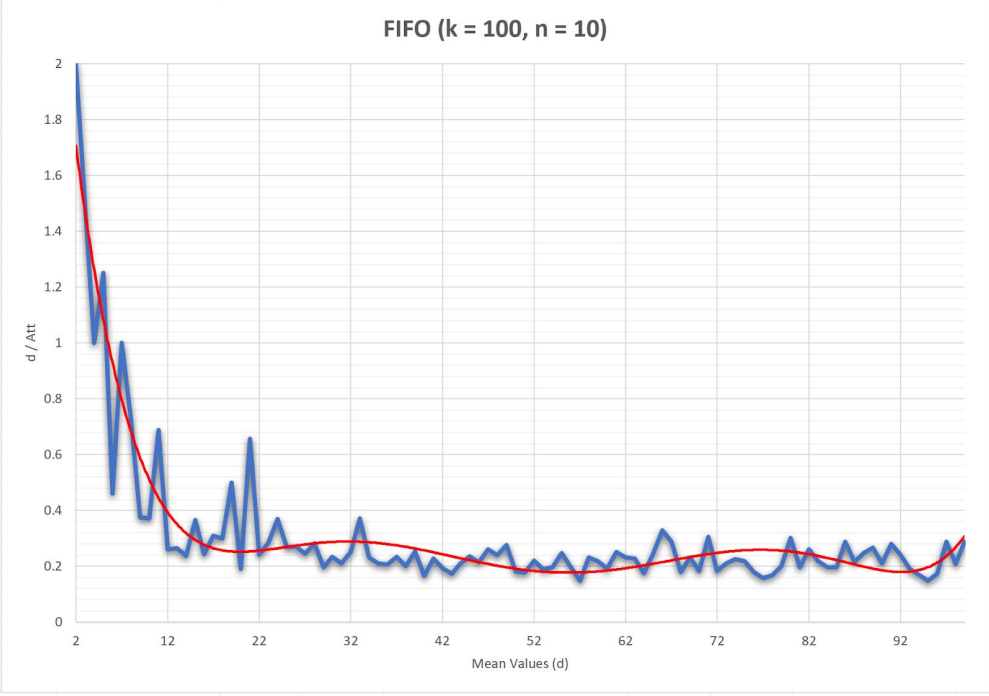
Analysis of 10 Scheduling Algorithms

Every scheduling algorithm has both benefits and drawbacks in how it performs under various load conditions. We took a look into 10 algorithms and generated an array of type struct with process data populated with random numbers using a Gaussian distribution that we were able to obtain using the GNU Scientific Library for C. In order to help ensure the test results are meaningful, we used the same array of process data for each of the 10 tests we ran to create the graphs and provide a fair comparison between them. We used 8 known scheduling algorithms as well as we created 2 random scheduling algorithms, one with and one without preemption of running processes. The focus of this project was to give readers a comparison of how a selection of scheduling algorithms can produce various results on the same processes, while each has benefits and drawbacks, to say one is better than another may depend on a specific use case.

Note: Some of the charts are missing some labels on the axises. They are the same for every chart. The x direction are mean values. Each process is assigned a random runtime (represented by number of clock ticks), but these values are generated based on a normal distribution distributed around this mean d. The y direction is a performance metric, d/Att. This is the mean over “average turnaround time”, which is the average number of clock ticks that it took the processes to finish running with the given algorithm.

Additionally, the k value is the upper limit on which processes can arrive. If k = 100, then processes can arrive any time from t = 0 to t = 100 and their arrival time is generated randomly based on a uniform distribution. N is the number of processes scheduled to run using the given algorithm.

First In First Out (FIFO)

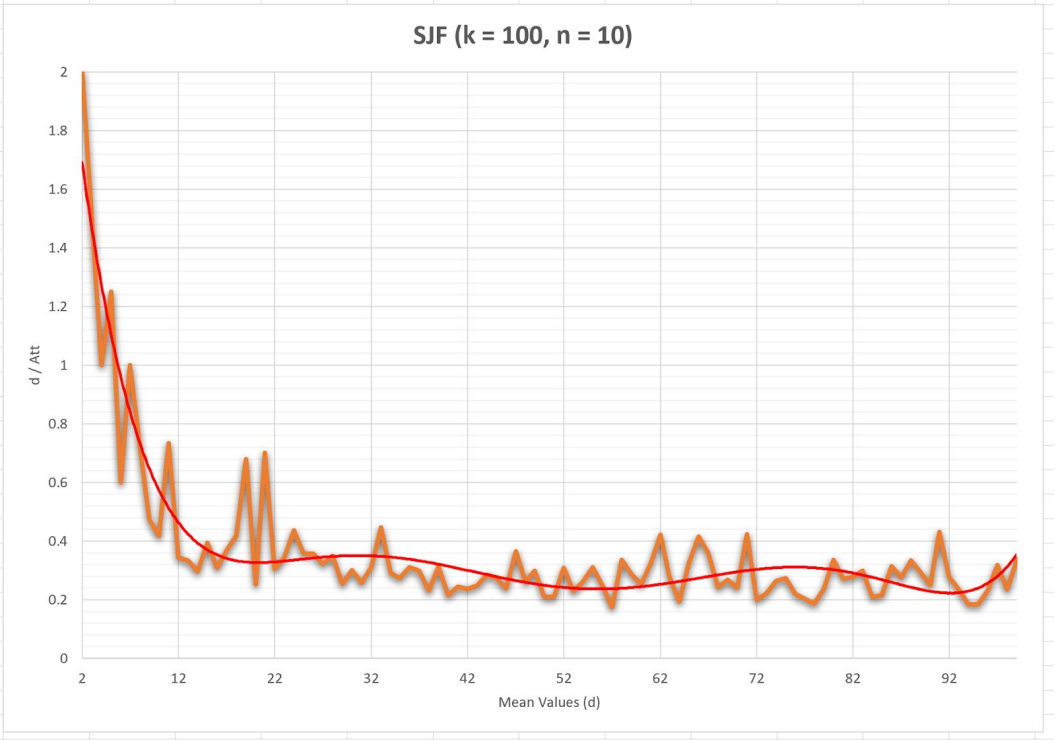


First-in First-out will ensure that as processes arrive, they are executed based on their arrival time. This algorithm has no preemption to stop long-running processes from hoarding the system resources. The response time may be high when the processes have a significant difference in process execution times.

Benefits:
First-in First-out is will give each process a chance to run and there will not be starvation of processes. FIFO is simple to implement and is the least complex algorithm to model. FIFO has a very low overhead.

Drawbacks:
If a large process arrives first, the very fast running processes that are in line behind it would have to wait for the long-running process to finish before they could be helped. There is no preemption so long-running processes cannot be stopped to let a short-running process complete. FIFO penalizes short processes and I/O bound processes.

Shortest Job First (SJF)

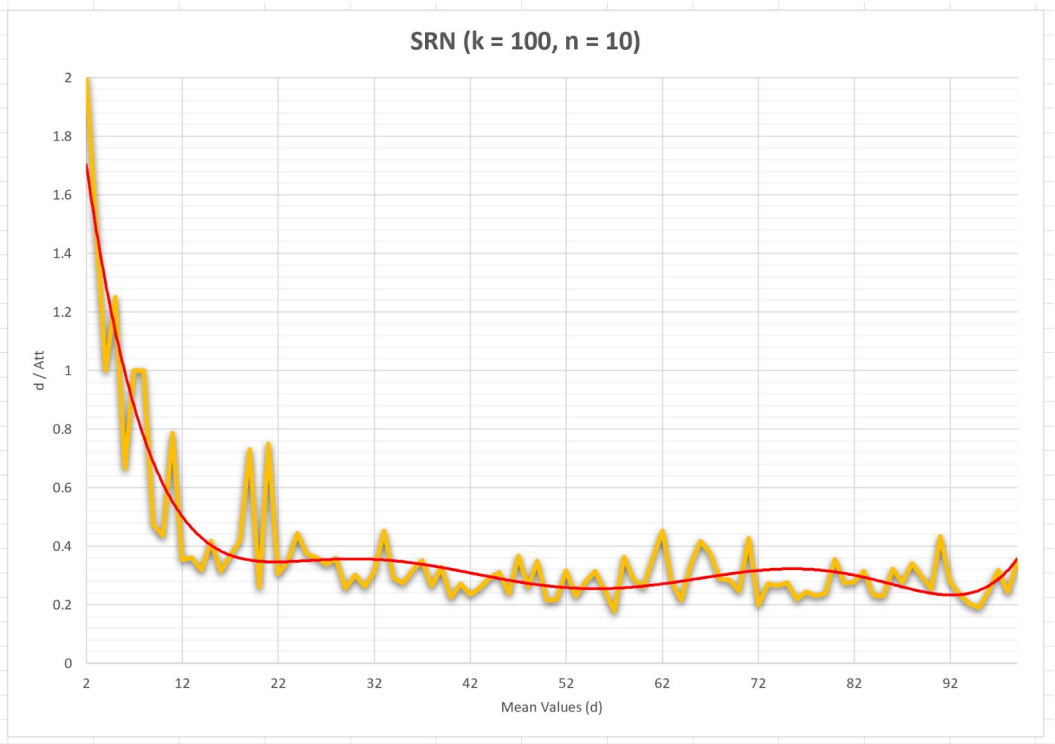


Shortest Job First(SJF) is also known as Shortest Process Next (SPN) and it is a non-preemptive scheduling algorithm that will look at the processes that are waiting to run and select the process with the shortest running time next. A variant of this algorithm is Shortest Remaining Time (SRT) that takes the same concept but allows for process preemption.

Benefits:
SJF has a high throughput of processes since it favors the shortest processes first. The response time for short processes is very good.

Downsides:
Longer processes are penalized and can have starvation, especially if many short processes are competing for system resources while a longer job is waiting to execute. SJF can have a high overhead.

Shortest Remaining Time (SRT)

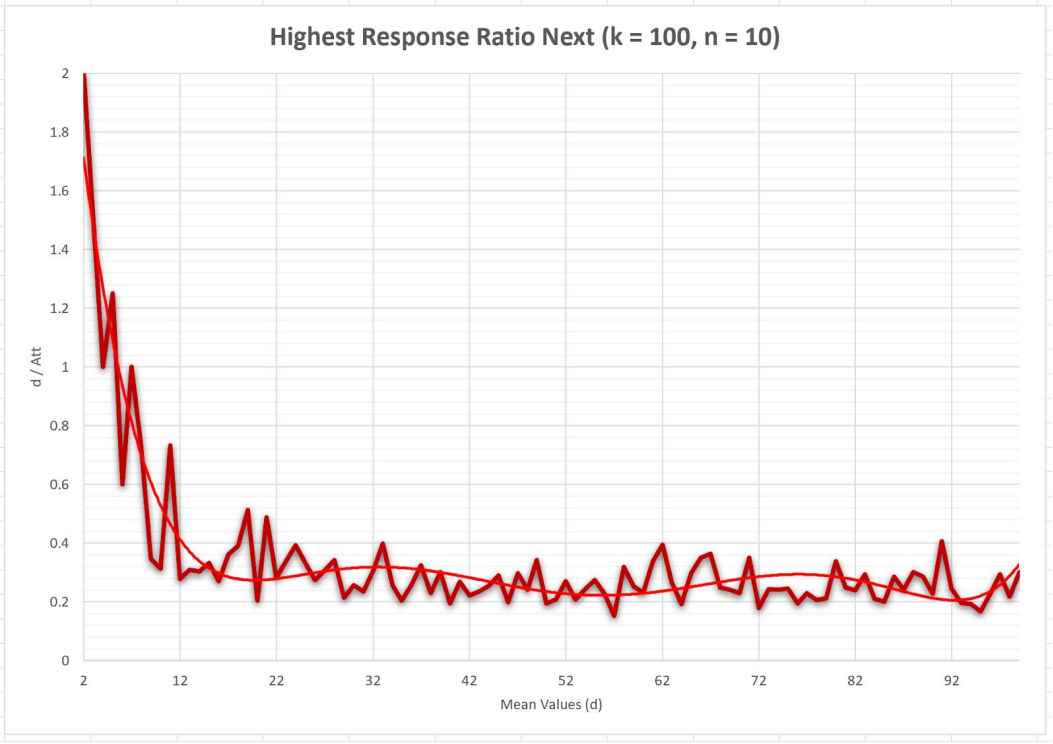


SRT is a preemptive version of Shortest Job First. Since this is a preemptive algorithm, the scheduler re-evaluates at each clock cycle what process then has the shortest remaining time. This makes it possible for the scheduler to run a particular process for just one clock cycle and then notice that a new shorter process is waiting to run and switch to the new shorter process.

Benefits:
The average turnaround time for a process with Shortest Remaining Time(SRT) should be lower overall than Shortest Job First(SJF) since the shortest running processes are given priority over everything at each tick of the CPU clock.

Downsides:
Longer processes are penalized and can have starvation, especially if many short processes are competing for system resources while a longer job is waiting to execute. SRT can have a high overhead.

Highest Response Ratio Next (HRRN)

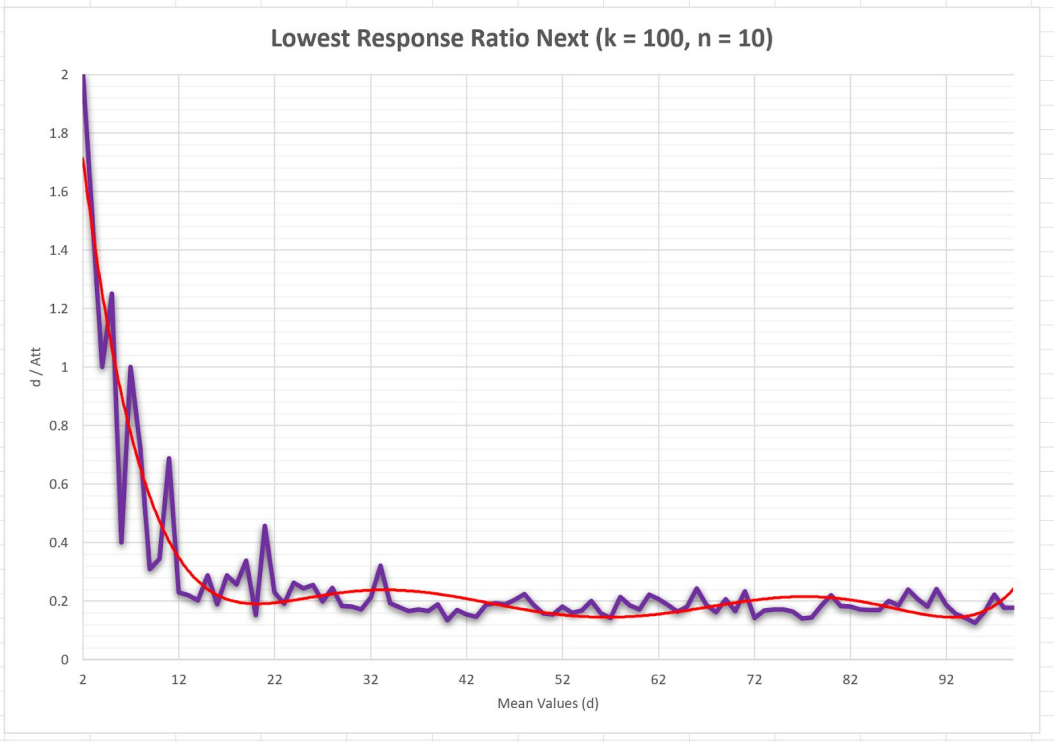


Highest Response Ratio Next favors the processes that have been waiting for the longest to run. This helps ensure that starvation of processes is very unlikely to happen. HRRN is a more balanced scheduling algorithm than algorithms such as Lowest Response Ratio Next that punishes processes that have been waiting or Shortest Job First that simply looks for the processes that will run fast.

Benefits:
The longer a process waits, the more likely it will be to run. This concept helps to ensure that there will be no starvation of longer processes.

Downsides:
HRRN is not preemptive so once a long-running process has started to execute, it will consume the system resources until it is complete. Although this algorithm will certainly prevent starvation, a very large job that may be waiting for some time as smaller processes are running will eventually get priority over short processes that need to get run. That very large job may take system resources for a long time as it runs to completion.

Lowest Response Ratio Next (LRRN)

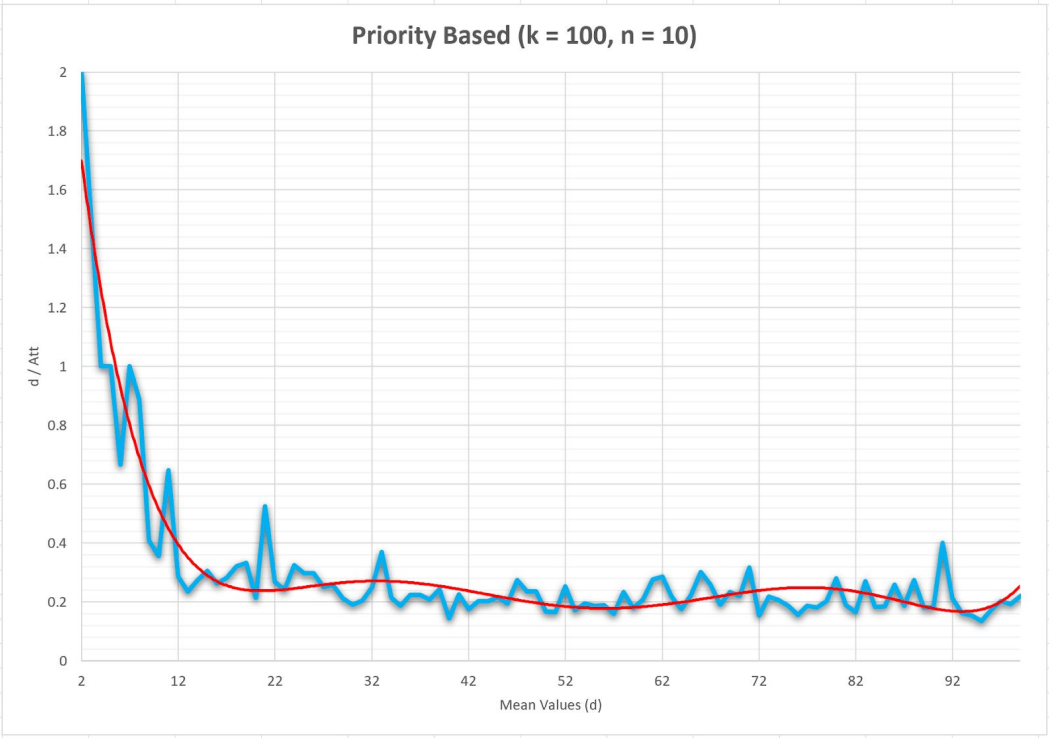


Lowest Response Ratio Next is the antithesis to HRRN. It's another preemptive algorithm. It uses the same formula for calculating the response ratio, but instead chooses the lowest. Yes, this is an algorithm designed to be intentionally bad, it was another fun one for us and used as a comparison point for the more serious algorithms.

Benefits:
A new process will almost definitely get a certain amount of runtime right away, so that's something. It also performs better in general than Round Robin, so it's not a completely terrible in average turnaround time, but there's virtually no reason to use it over HRRN

Downsides:
This algorithm virtually guarantees starvation. Long processes that arrived earliest will continue to grow their response ratio. This means that they will continue to get interupted by new processes and will take a long time to finish running unless in isolation. This is neither a fair or particularly efficient algorithm.

Priority Based (PB)

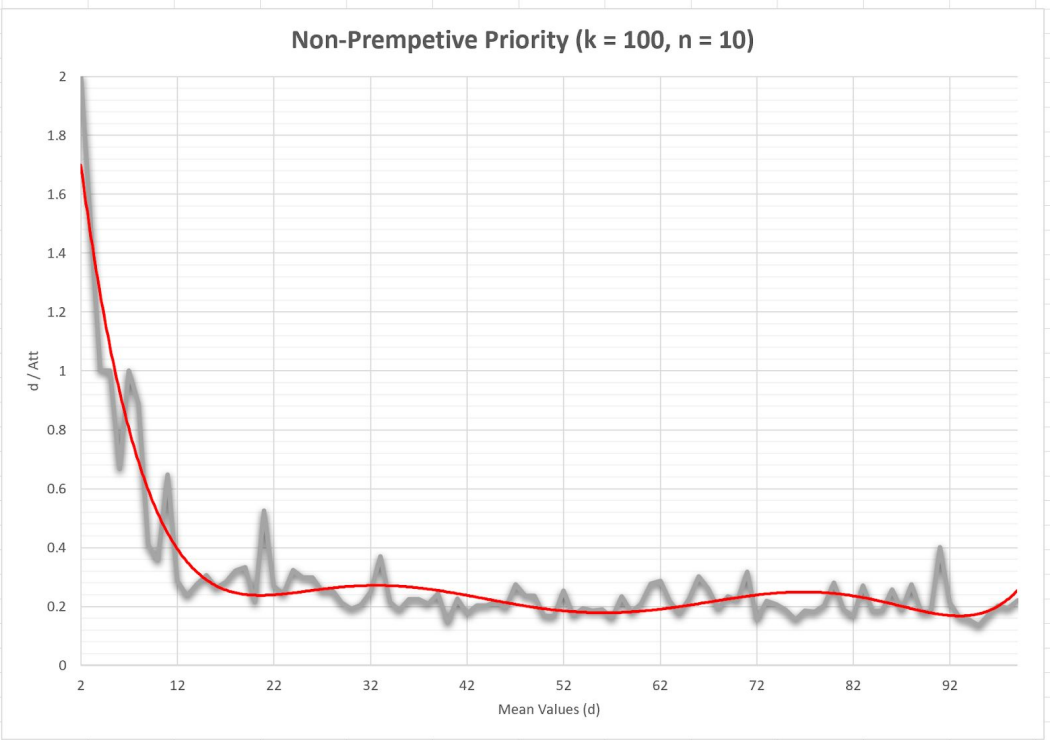


This algorithm is useful when processes has an associated priority with them. Naturally, you would want higher priority processes to run before lower priority ones. For our implementation, priorities were assigned randomly to each process using a uniform distribution. When the processor is free, the algorithm simply chooses the available process with the highest priority. Ties are broken by arrival time.

Benefits:
This algorithm works well if processes have priorities. It isn't built to be fair or efficient, but to make sure that more important processes get the processor time that they need. That being said, it still performed decently well: better than Round Robin and just slightly worse that FIFO.

Downsides:
The algorithm isn't particularly efficient, but it's not designed to be. Because it is preemptive, important processes have the potential to be interrupted. This is usually the desired effect, but could cause issues if it's important that important processes run to completion.

Non-Preemptive Priority

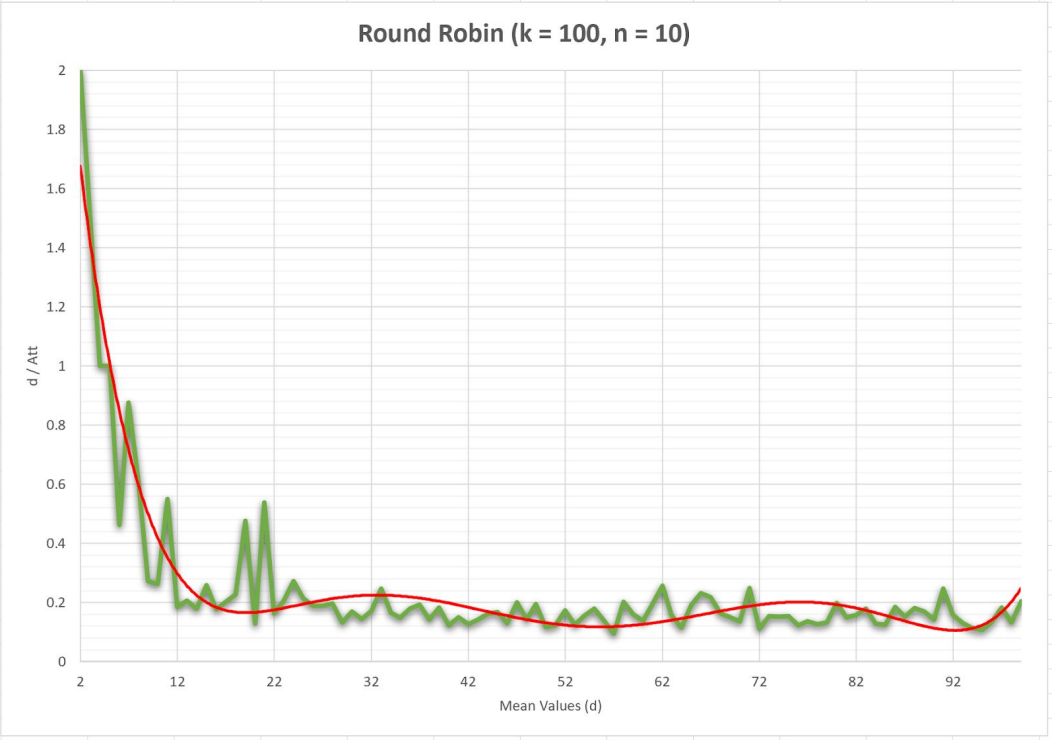


This is another algorithm that takes priority into account. It works the same as our PB algorithm when it comes to selecting processes. However, this is a non-preemptive algorithm, so once a process is selected it will run until finished, even if a higher priority process comes along. This may not be particularly practical in a real system.

Benefits:
This algorithm of course takes the priority of a process into account, the only other algorithm to do so being PB. If it's very important for a high priority process to be completed once started, then this is the algorithm for you.

Downsides:
This algorithm shares all the downsides of the The biggest downside is that this algorithm is not preemptive, so a higher priority process may come along that still doesn't get a chance to run for awhile if the current process has a lot of runtime remaining.

Round Robin (RR)

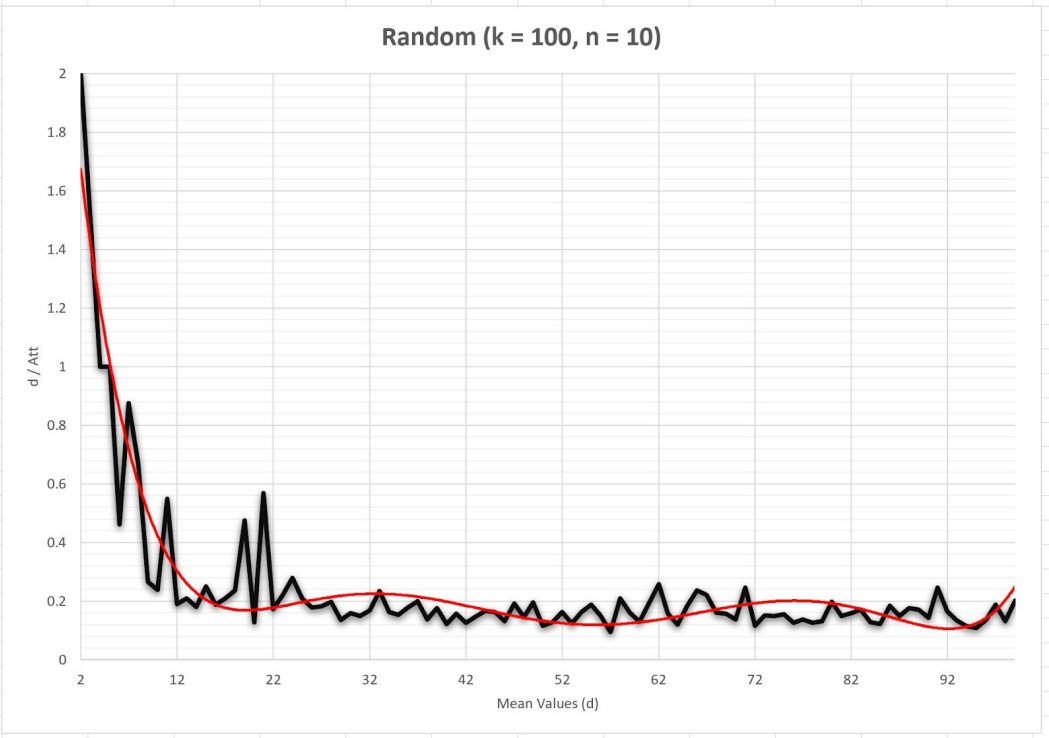


The Round Robin algorithm is designed to give each process an equal amount of time to run on the processor. Some particular time quanta is specified ahead of the algorithm running. Each process then runs in the order of arrival on the processor for that amount of time (in our simulation, the number of clock ticks). Once a process has finished, if there is still time left in the time quanta, the next process in the queue is chosen immediately.

Benefits:
Round Robin is a very “fair” algorithm. It ensures that processes will get an equal amount of run time on the processor by having them run for a particular time quanta. The lower the time quanta, the more “fair” the algorithm. As the time quanta increases, the algorithm effectively becomes first come first serve.

Downsides:
Round Robin is not very efficient. Although fair, each process will run for a short amount of time, meaning that the average turnaround time for every process will be longer. The shorter the time quanta is, the truer this is. Additionally, when looking at a real system as opposed to a simulation, frequently switching processes is especially inefficient.

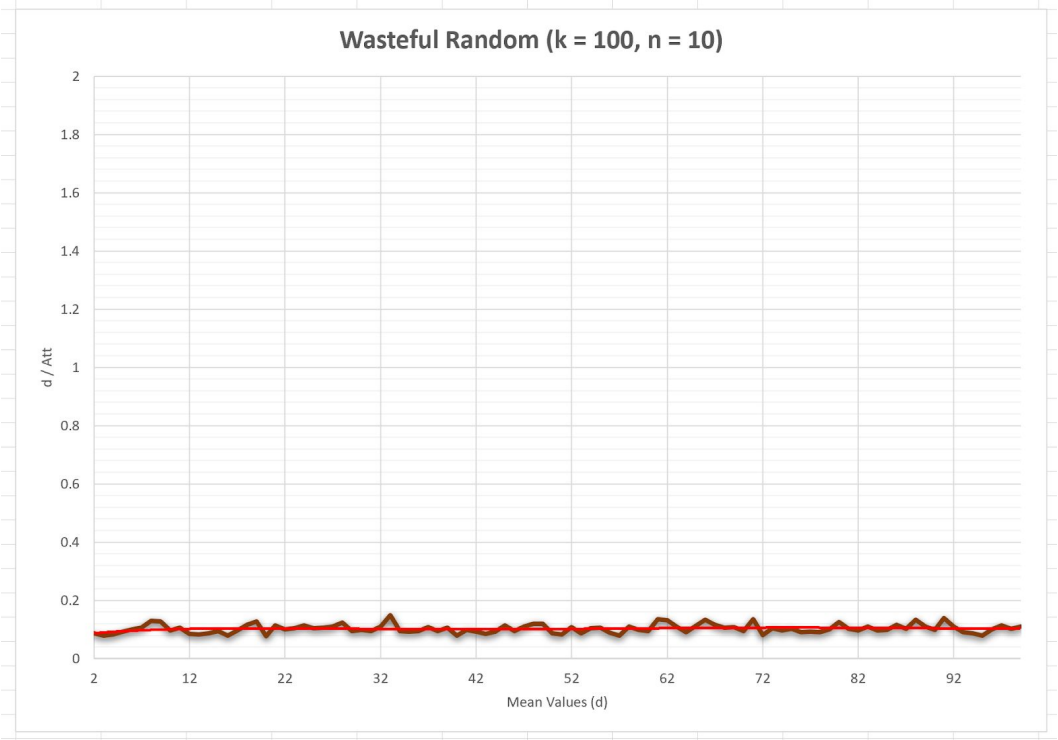
Random Scheduling



This algorithm simply picks a random process to run at each time interval. It's preemptive, so for each clock tick a process is selected. The only way a process can run for more than one clock tick is if it happens to be randomly selected again. To choose a process, all processes are iterated through to find one that's active. Of these, a random one is selected to run.

Benefits:
This is a "fair" algorithm in the sense that it does not favor long or shorter processes. It also doesn't really favor processes by arrival time, although those that arrive first will of course go into the pool of available processes first. Statistics suggest that processes will likely get a roughly fair share of processor time this way, but there is no guarantee that they will.
Downsides:
This algorithm is mostly downsides. It's not efficient at all. By choosing a random process at each clock tick, this algorithm extends the average turnaround time significantly. Even short processes will likely take a long time to finish if they are contested at all. There's not really an argument for fairness either, as an algorithm like Round Robin actually guarantees fairness, while randomness relies on probability distributions to make fairness more likely.

Wasteful Random

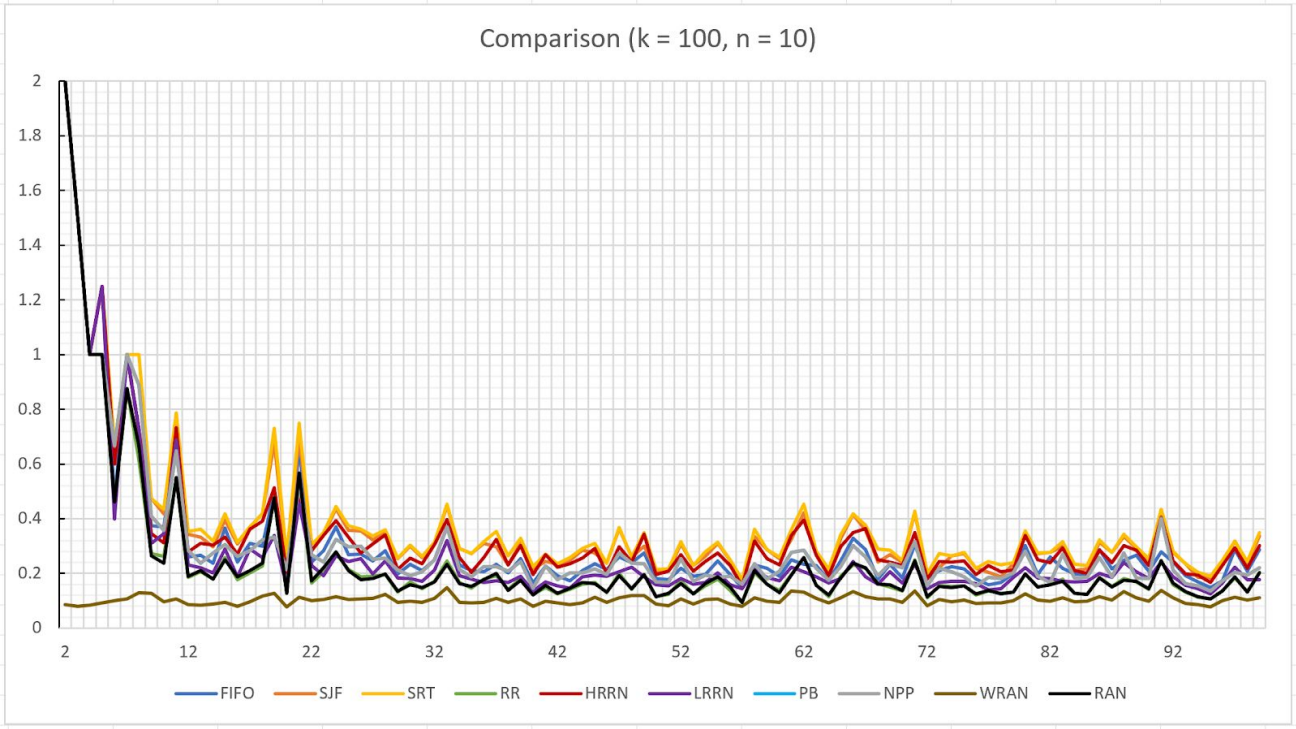


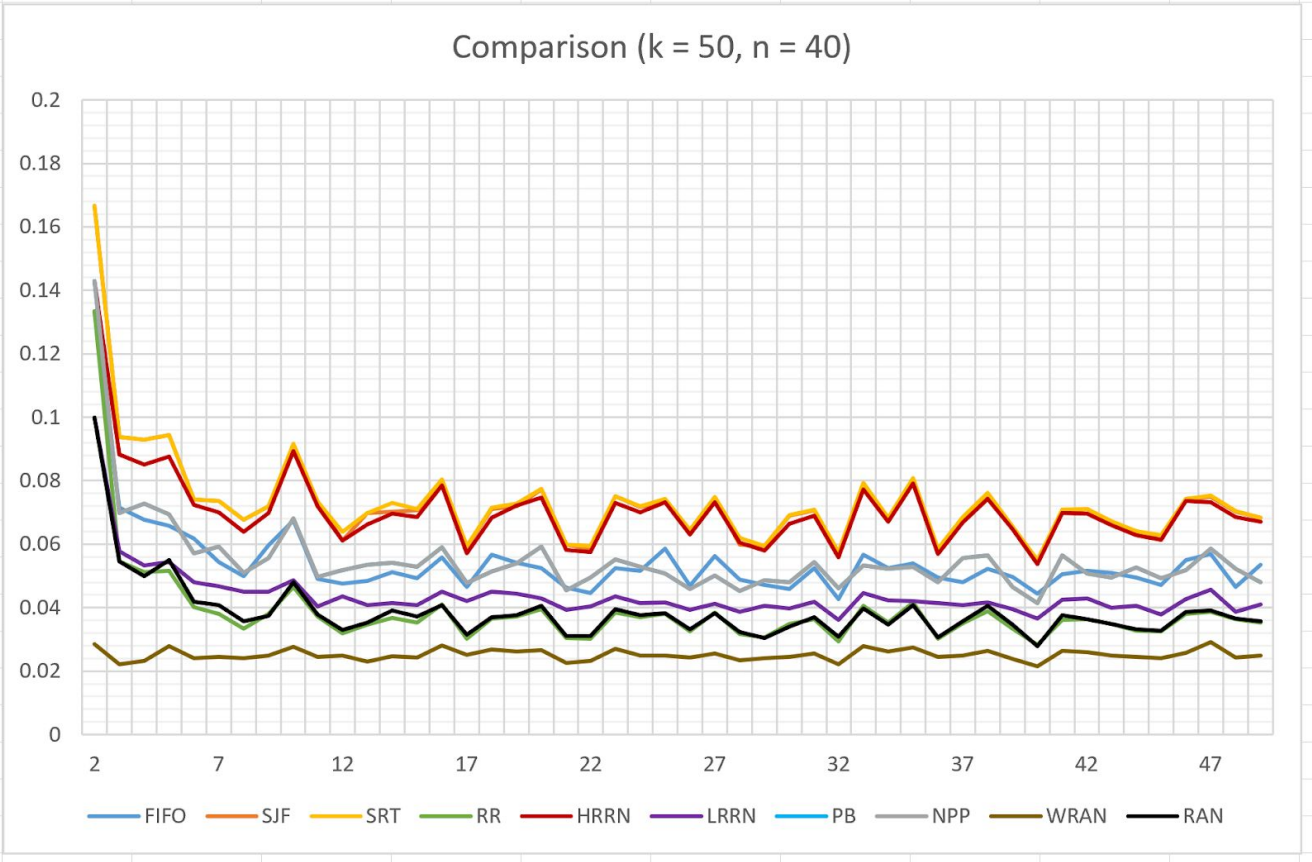
Because many of our algorithms performed fairly similarly, we wanted to have a little fun and look at an algorithm designed to perform intentionally poorly. This “wasteful” random algorithm does just that. Every time the clock ticks, a random process is chosen. If that process is not active or has already finished running, then no process is run. This wastes processor time. Each process runs for only one clock tick, meaning the average turnaround for each process is truly awful, as can be seen.

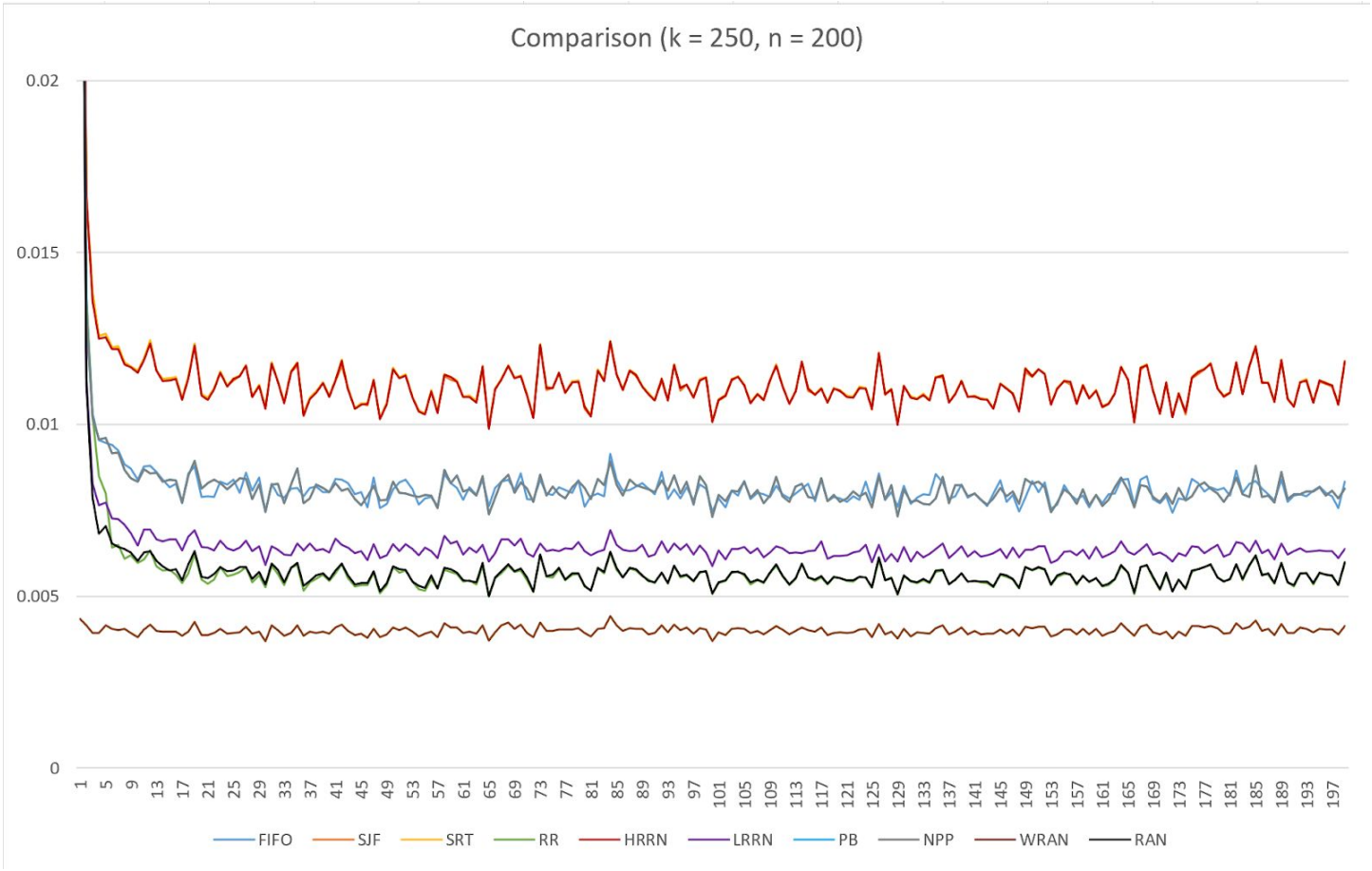
Benefits:
There are virtually no benefits to this algorithm. We essentially used it as a benchmark. If an algorithm performed worse than this, something went very poorly with the implementation. On the bright side, this algorithm is very resilient to a change in mean (it performs poorly with any mean value).

Downsides:
This algorithm does not choose processes in an intelligent way. It fails on two main metrics: “fair” and “efficient”. It is neither. Processes are chosen at random, ignoring if they are even valid processes to run. If they are invalid and chosen, processor time is wasted instead of choosing a valid process. This is a very bad algorithm, but it was fun to get a graph that had a very different shape than our other algorithms.

Comparison


Comparison (k = 100, n = 10)
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
2
12
22
32
42
52
62
72
82
92
FIFO
SJF
SRT
RR
HRRN
LRRN
PB
NPP
WRAN
RAN


Comparison (k = 50, n = 40)
0.2
0.18
0.16
0.14
0.12
0.1
0.08
0.06
0.04
0.02
0
2
7
12
17
22
27
32
37
42
47
FIFO
SJF
SRT
RR
HRRN
LRRN
PB
NPP
WRAN
RAN



When comparing these algorithms, so obvious patterns emerge. The more processes we run with, the more separation we can see between the different “classes” of algorithms. This is especially true when there is a k value that is not significantly larger than our n value, as it means there is a large amount of overlap in when processes arrive on our simulated processor. All three of these comparison charts with different tunable parameters agree about these divisions in algorithm performance.

The three best performing algorithms are SJF, SRT, and HRRN. This is no surprise to us, as the purpose of these algorithms is to increase performance over the likes of FIFO and RR. SRT appears to be a slight leader, which is more evident when looking at the first comparison chart.

The next class of algorithms is FIFO and PB. It’s not much of a surprise that these would have a similar performance, since our priority algorithms are essentially FIFO + taking priority into account. Preemptive and nonpreemptive priority don’t seem to have a discernible difference in terms of performance.

The next best algorithm is LRRN. It makes sense that this would do worse than FIFO, but it’s a little surprising to set hat LRRN performs better than RR. After some further thinking, this is likely because while LRRN does not optimize for average turnaround time, a side effect of always choosing the process with a small turnaround time is that many processes will finish quickly. This means that average turnaround time will come down a little, compared to RR and random which don’t consider this at all.

The next class is RR and random. It makes sense that these algorithms would be very similar, since they each run an available process without considering how long it’s been waiting for some time quanta without

preemption. It's a little surprising that random does seem to have a slight edge, although that may be because it always has a time quanta of 1 (something that future work could probably improve) while RR usually had a time quanta of 3 or 5.

Of course, our worst performing algorithm is wasteful random. It's actually throwing away process clock time if it doesn't happen to find a process to run on its first guess, so there's no way an algorithm could perform worse. It's also the only algorithm that isn't really susceptible to a change in d values.

Resources:

https://www.gnu.org/software/gsl/

Shidali, G. A., et al. "A New Hybrid Process Scheduling Algorithm (Pre-Emptive Modified Highest Response Ratio Next)." *Computer Science and Engineering*, Scientific & Academic Publishing, article.sapub.org/10.5923.j.computer.20150501.01.html.