The goal of this project was to create and simulate four cpu scheduling algorithms. The selected algorithms were; First Come First Served, Shortest Job First, Shortest Time Remaining, and Round Robin. To program these algorithms I created and a class called “alg” which has 4 methods that represent each individual algorithm. Each method (algorithm) starts and ends in a similar manner. To simulate a cpu queue a text file containing a list of numbers is opened and parsed. Each number entry in the text file represents a process and the time it requires for it’s operation. After the data has been read and stored the methods break off into differing directions as each different algorithm then schedules the data for the simulated cpu.

After the data from the text file has been saved into a vector called “ready”, another vector called “wait” is created and initialized to the same size as ready, with all of it’s elements at zero. The ready queue represents the time needed on the cpu by each of the processes in the system. While the wait queue represents the amount of time each process has been waiting for the cpu. Top mimic actual cpu function each scheduling algorithm selects a process from the ready queue then decrements the number value of said process in the ready queue to simulate that process having the cpu for that time slice. All elements of the wait queue except for the one corresponding to the process that was just decremented are then incremented by the same time slice. This is done to keep track of how long each process is sitting idle while the cpu is running. Afterwards once the simulations have been run and none of the values in the ready queue are not positive (meaning all processes have been run) the selected algorithm then does calculations to generate statistics for the simulation. Each algorithm has calls to other methods which analyze the wait queue and calculate the minimum, maximum, and average wait times. This is done so that strengths and weaknesses of the varying algorithms can be compared for the sake of research.

First Come First Served is the simplest of the scheduling algorithms. It runs through the ready queue in sequential order. Where each processes goes through the process of getting cpu time until it is finished. There really isn’t much too it, the first element of the ready queue is reduced to zero, then the second, and so on and so forth. Shortest Job First functions almost identically to FCFS in this simulation. The goal of SJF is to always have the job with the shortest remaining cpu time on the cpu. So to do this the SJF algorithm simply sorts the ready queue in descending order, using the standard template library sort function for vectors. Then runs through each process in the queue much like FCFS. Round Robin is a relatively simple algorithm. It runs through the ready queue in sequential order much like FCFS but it only simulates running each process for one predefined time slice before moving to the next process in the queue. The algorithm cycles through the queue, decrementing each process by one time slice, and looping until all processes have been run.

To test the relative strengths and weaknesses of the varying algorithms, I created a text file with two hundred entries. Each entry is a random number that represents the time it would take a process to run once set on the cpu. Then I ran the four algorithms I programed with the 200 data entries, and generated a graph showing the results of each algorithm.

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

Note: All of the algorithms above were run with the same default time slice, for comparisons sake.

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

As demonstrated above FCFS may be the simplest algorithm, but it is also causes the processes it is run on to spend considerably more time waiting on average than the other algorithms. This is because that FCFS has no means to run smaller processes before larger processes. Meaning that small processes can have large wait times since they cannot run before earlier items in the queue have finished. EX. Consider a ready queue with three processes of lengths, 3, 2, and 1. In FCFS the average wait time would be 8/3. However if the queue was reordered to be 1, 2, 3, then the average wait time would be 1. FCFS’s wait times are dependent upon the ordering of the ready queue. Meaning that it’s average wait time will always be higher than SJF or RR when run on the same set of data. Since SJF reorders to run the smallest items first it doesn’t encounter this problem. RR avoids this problem by cycling through all of the items in it’s queue continuously, giving the opportunity for smaller entries to be knocked out. For these reasons SJF will almost always have the smallest average wait time of all of the algorithms. However this performance comes at a price, in a real operating system SJF can wreak havoc if left unchecked. Since it prioritizes running the shortest processes, there may be processes in the ready queue that get infinitely postponed and are subsequently never run. RR doesn’t experience this issue due to it’s cyclic nature but it’s use comes at the cost of a slightly higher average wait time.

Round Robing can also fall victim to excessive context switches. Every time a cpu stops work on one process it must save all relevant information to that process, and then load all relevant information for the next process to be run. This is referred to as a context switch. In an actual system context switches take up time and reduce the efficiency of the cpu. However since the actual time a switch takes up is often very small (in the fractions of a second range) and because this is highly variable between differing hardware. I elected to not increment waiting times for the processes that are waiting to execute whenever the algorithm executes a context switch. Instead I simply added a counter that keeps track of how many context switches occur during the running of the program. This way we can achieve accurate wait times while still being informed of how many times the cpu would have to take time to stop and perform a context switch. Generally speaking the fewer context switches in a program, the more efficiently it runs because the cpu is performing processes rather than switching processes. I ran the Round Robin algorithm that I programmed against the same set of 200 processes eleven times. Each time I altered the size of the time slice in which a process is run before the cpu moves to the next entry in the queue

As you can see the larger the time slice that the algorithm was run with fewer the context switches the program experienced. This is because larger time slices mean that process require fewer time slices to finish executing. EX. A process that has a length of 100 seconds, is run against the algorithm two times. Once with a time slice of 20 and once with a time slice of 50. The first requires the algorithm to perform a context switch 5 times to perform 100 seconds of work, while the second only requires 2 context switches to run the process.

Thus we can conclude that larger time slices ultimately contribute to a more efficient system. However the law of diminishing returns applies to this concept. Once the time slices become so large that they begin to eclipse the total time to run a process, it doesn’t matter if the time slice is increased after that point. Since it will still only take one context switch to run a process of 100 seconds if the time slice is 130 or 300000.

Unfortunately increasing the time slice of a cpu can lead to complications. Using the same test data as before, I found that increasing the time slice for the algorithm lead to alterations in other statistics generated from the program. The data showed that increasing the time slice can increase both the average wait time and the maximum wait time for processes. This is because smaller processes which could be taken care of during early run through of the queue with smaller time slices, are forced to wait longer on other processes since the time slice has increased. EX. Given a queue with two processes 100, and 10 seconds long. If these two are run with a time slice of 10 seconds, the shorter process only has to wait for 10 seconds to be run. However if the time slice were larger, such as 80. The 2nd process would have to wait for the first process to go through it’s time slice, meaning that the 2nd process would have to wait for 80 seconds as opposed to 10. While altering the time slice does not guarantee a resulting change in the average or maximum wait times, it is possible.

Shortest Time Remaining is the most complicated of the algorithms. It is similar to SJF first but it is also preemptive. Meaning that if a new process arrives in the ready queue and it’s time remaining is less than the currently worked on process, the process on the cpu will be shelved and the other process will be placed onto the cpu. However our current cpu implementation involves selecting from a static ready queue initialized at the start of the program. So if we were to run this algorithm it would function identically to SJF. To counteract this, for STR only I added code to make the ready queue dynamic. Meaning that the algorithm pulls a set amount of processes from the text file at the start of the program. Then it selects the minimum value of the ready queue by using the standard template library minimum function for vectors, and begins running it. After each time slice has passed the program pulls a new process from the text file and adds it to the ready queue (wait and bool queues are updated to reflect initialization of a new entry). The shortest process is then selected and run for the next time slice. Since this algorithm doesn’t involve re-ordering the ready queue, there had to be a way to ensure that processes that have finished don’t get time added to their ready queue when other processes run. To accomplish this, I implemented a vector of Boolean values. This vector has just as many elements as the ready and wait queue. So if bool [i] is false then ready [i] hasn’t finished running, and wait [i] can be incremented because the process on ready[i] can still be run. This way we can create an algorithm that can function differently than SJF under the right set of circumstances. We have achieved a working model of the STR algorithm, however it cannot be accurately be used in comparison to our other algorithms (FCFS, SJF, RR) on the same set of data. The simulation is correct in regards to how a real cpu would function, but running STR with only some of the processes in the ready queue at initialization would lead to seemingly better results for STR than the other algorithms. This is because the wait times for all of the processes that are added in dynamically after the algorithm has started would be lower than if they were included in initialization. Much like a real operating system a process that doesn’t even exist until iteration 50 of the ready queue won’t have any waiting time for the 50 previous entries. Thus STR is not fit for comparison against the other algorithms developed. And for this reason will not be part of the comparative analysis between algorithms.

In conclusion, First Come First Served is a simple and easy to implement cpu scheduling algorithm. It has the worst average wait time of all algorithms but it doesn’t have any pitfalls either. This algorithm may take a while to run, but it makes up for it in stability. Due to it’s structure starvation is impossible, overhead is very low and context switches are kept to an absolute minimum. This type of scheduling would be ideal for a system with little to no user input due to potential long wait times. Or a system that must run all processes fed to it as efficiently as possible without neglecting anything. Shortest Job First is a algorithm with excellent average wait time scores. Only a small amount of initial overhead to reorder the ready queue, and efficiency on par with FCFS since they share the same number of context switches. This algorithm would be best for a system with much user interaction. Since it can deliver excellent response time to short I/O processes. It would not fare well for heavy computing or operating for extended periods of time, due to it’s tendency to starve longer processes. Round Robin is an amiable compromise between FCFS and SJF. It provides adequate average wait times and prevents starvation. However it can perform poorly depending upon the time slice selected and the hardware it is run on. This being due to it’s potential for large number of context switches which can be difficult for some computers to handle. Shortest Time Remaining is basically a slightly more advanced version of SJF. It will be slightly more responsive due to it’s preemptive nature but isn’t much different otherwise. Each algorithm has there strengths and weaknesses. Selecting which one to employ is ultimately matter of the hardware one has on hand and the desired usage of the computer. Although to build a full operating system that is stable enough and responsive enough to actually use, I’d definitely recommend a combination of these algorithms to be integrated for optimal performance and redundancy.