Operating Systems

Casey Price

SPSU

Marietta, GA, USA

Cprice2@spsu.edu

1. **Introduction**

This project is an operating system simulator. It was built to test how different long term scheduler algorithms and different configurations of hardware affect the system. We tested 3 different algorithms for the long term scheduler, first come first serve, priority, and shortest job first. For the hardware we tested with variable size of ram and variable CPU's.

1. **Related Works**

Silberschatz, Abraham, Peter B. Galvin, and Greg Gagne. Operating System Concepts. Hoboken, NJ: J. Wiley & Sons, 2009. Print.

.NET Framework 4." .NET Framework 4. Microsoft, n.d. Web. 29 Nov. 2014.

"Stack Overflow." Stack Overflow. N.p., n.d. Web. 29 Nov. 2014.

1. **Methodology**

I implemented my operating system simulator using C# and the .Net framework version 4.5. I used visual studio 2012 for my IDE, which helped debug the multithreaded portion of the code. I chose these tools because I am most familiar with C# as a language, and I know how to do everything required for this project in that language. Threading was extremely easy to implement since it was only 1 line of code to kick off a new thread when using a thread pool ThreadPool.QueueUserWorkItem(cpu.Execute). The thread pool class in C# handles creating, managing, and deleting all threads for you. All you have to do is give it a delegate and it will create a new thread running that method. However, thread pools remove some control that you would otherwise have when creating threads manually. For example, you can't guarantee the thread will be created at the exact moment you queue the thread. You also can't name your threads for the debugger. All threads created in a thread pool will have the name 'Worker thread' in visual studio, which can be a little tedious when debugging if you have hundreds of threads running at the same time.

When I designed my simulator, I used the MVC software design model which abstracts the frontend UI code from the backend. I created a frontend project which interfaces with my backend library who doesn't even know the frontend exists. The frontend is a simple Winform project that takes user input through combo boxes and text boxes for the algorithm, number of CPU's, size of ram, and iterations, to run the simulation. The user can select what input file to read from for jobs in the CPU. This design allows anyone to easily develop a new UI with my backend because all necessary methods to run the simulation (and only those methods) are public to the frontend.

I used static classes for the LTS and STS objects because they don't hold any data and only methods that get called from the main operating system object. The LTS only does 1 thing, it takes jobs in the order they are in System Memory, which is a singleton that holds all the PCB's that were read in from the file, and puts them into RAM. It also has 2 methods for sorting System Memory based on the selected algorithm.

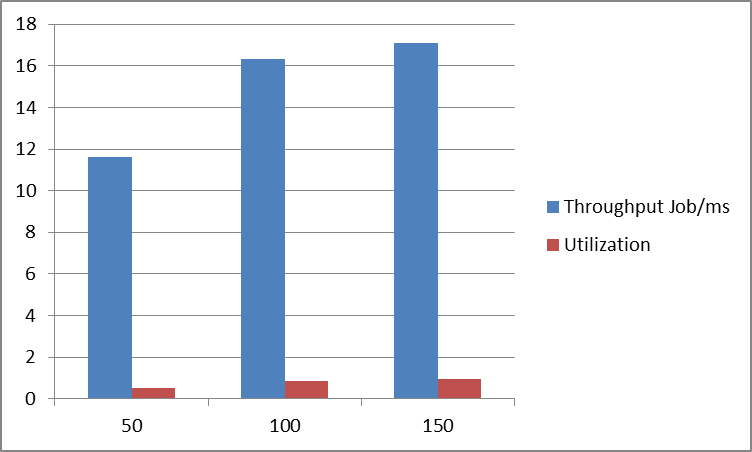
The HDD and RAM are a mirror of each other. They are a wrapper class for a list of instruction objects. The only difference between the two is that ram has a max size that is set when it is created. Ram is also thread safe, meaning that 2 threads can accesses it at any time and ram will wait until it is done servicing one thread to service another. This is done to prevent ram being accessed while it is being compacted.

The bulk of the CPU is one giant switch statement that decodes the given instruction and then executes it. The CPU class is also designed after the Van-Neumann machine design in which the CPU fetches, decodes, and then executes each instruction, in that order. The CPU is given a PCB with a starting location in ram and a program counter that shows where the PCB currently is in execution. The CPU uses this data to fetch the next instruction in RAM before decoding and executing it, followed by increasing the PC in the it's PCB. When the CPU is done executing, it saves its current Register and Accumulator fields back to the PCB and then puts it back into system memory and waits for the short term scheduler to give it another PCB to execute. This process continues until all PCB's have been fully executed and the results are printed in the window.

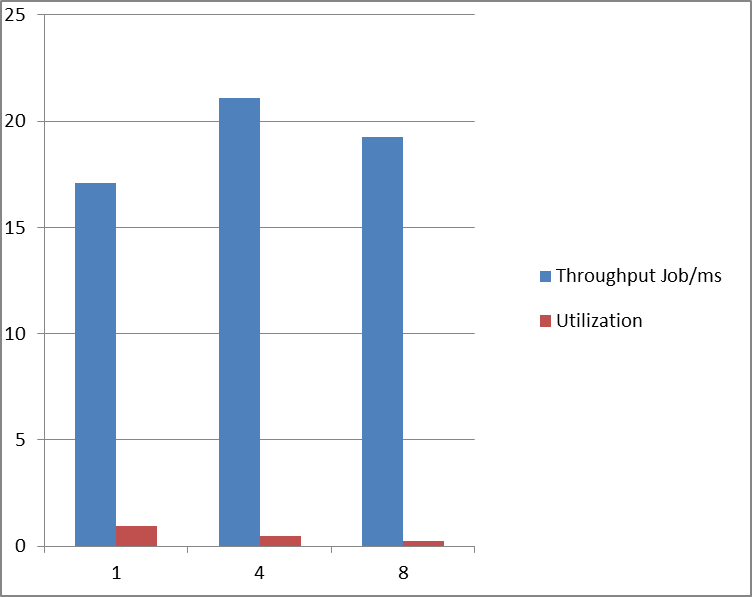
For each test done, I ran the selected configuration 1000 times, the configurations were 1, 4, and 8 CPU’s with 50, 100, and 150 instructions being able to fit in ram. I then took the average

1. **Results**

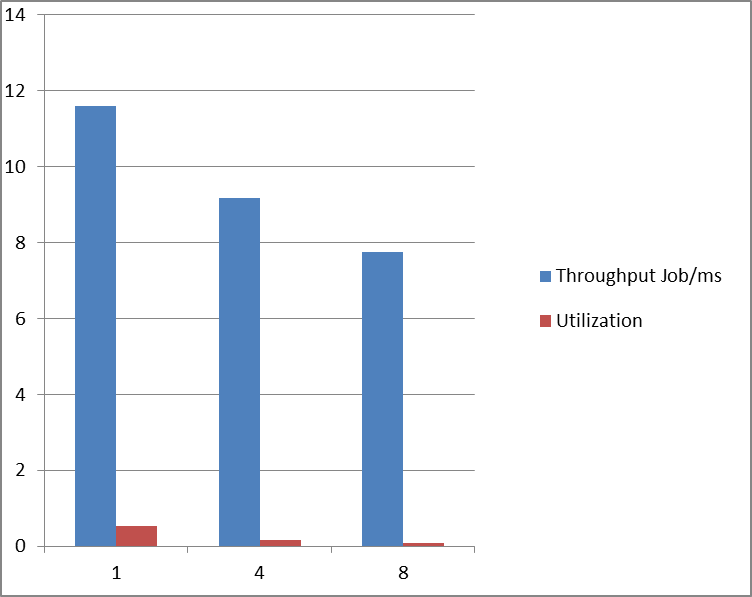
I found in my results that increasing the amount of ram increased the amount of CPU utilization and throughput. This is true because the more jobs that are in ram at the same time, the more that can be executed by a CPU.



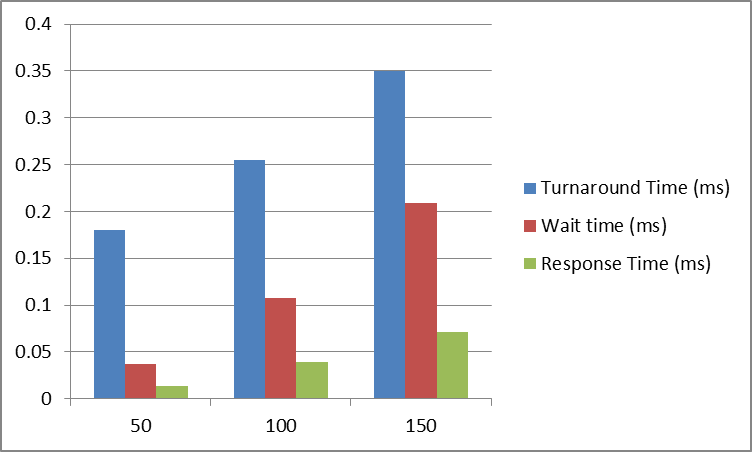
As you can see in the graph, as the size of ram increased, throughput increased greatly. This test was run with the same Priority algorithm for all 3 runs and 1 CPU. However, when I ran the same algorithm with varying CPU’s I noticed that Throughput and utilization actually declined from 4 to 8 CPU’s, this is due to the added overhead of having CPU’s that don’t do anything. As you can see the utilization was also very low due to ram bottlenecking the CPU’s from being able to get anything done. This test was run using the priority algorithm and 150 lines of ram.



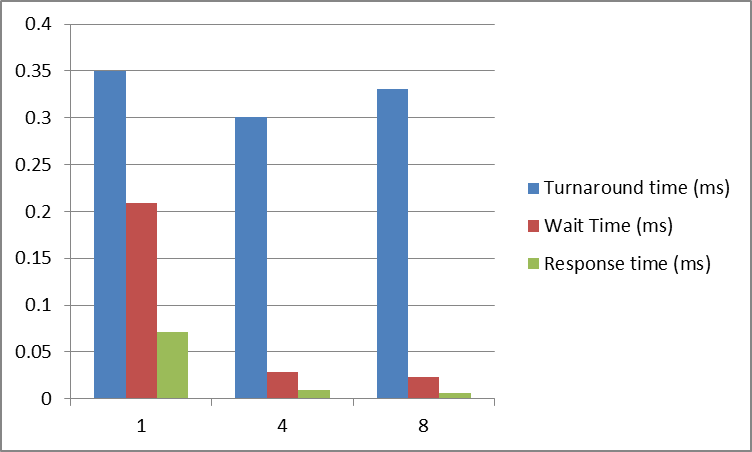
However, this was only the case when there was sufficient ram. The following were run using 50 lines of ram and the priority algorithm.

  
As you can see throughput now only goes down when increasing the CPU count. This is because the bottleneck is ram, and adding on more CPU’s at this point only adds additional overhead, since were constantly creating threads for CPU’s to do absolutely nothing.

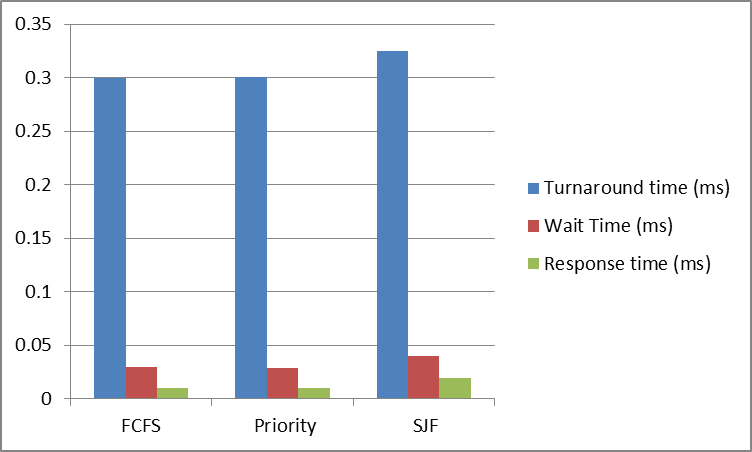
Turnaround time, wait time, and response time all increased as ram increased, which makes sense, because the more you have in ram, the longer you’re going to have to wait in ram. More ram also means more to sort through and more operations adding and compacting ram, which is more time the system is not executing processes in the CPU, assuming a thread finishes before the big loop gets back to its execute section. The following was run with 1 CPU, priority algorithm, and 50, 100, and 150 lines of ram.



As the number of CPU’s increased the response and wait time decreased, while the turnaround time stayed roughly the same since jobs aren’t being executed faster, just more at the same time.



Lastly, the algorithms had some interesting results. In theory SJF should minimize both wait and response time, however my results showed otherwise. Across the board SJF had the worst wait and response times for all configurations. Here is an example for 150 lines of ram and 4 CPU’s, the most efficient configuration.



1. **Conclusion**

My final conclusion is that RAM should be the first priority when building a computer, because it gave the largest performance increase. And only when there is enough ram, did increasing the number of CPU's make a positive difference. In fact, having 8 CPU’s made performance worse because it added extra overhead. RAM is also very cheap compared to the other parts of your computer, so spending more on RAM makes sense.

As far as algorithms, I think Priority is the best if used correctly. FCFS sounds nice in theory, since all jobs are treated equally, but in the real world certain processes have deadlines and need to be done in a certain amount of time. For example, in a jet fighter there are many time sensitive processes that need to be done or a threat might not respond in proper amount of time.

Shortest job is a close second, as it gets the most amount of jobs in ram as quickly as possible to increase CPU utilization and minimize wait times at first, but as the bigger jobs begin to come in CPU utilization slows down. SJF ultimately had the worst performance times across the board, which was weird, because in theory it should minimize the wait and response times, not increase them. I believe the reason my data showed the exact opposite was because of how many stop, wait, IO, and error commands occurred. SJF is based on the idea that a job will take however long it states to execute, and assumes that a job with 15 instructions will take half as long as a job with 30 instructions. However, since almost every job terminated early, the actual length of each job was not the same as the length of the instructions inside of it. The fact is, you cannot tell how long a job will take based off the number of instructions alone because of errors that can occur.