CS 3243 – Operating Systems Phase 1 Report

Robbie Diaz, Kevin Debrito, David Butcher



Table of Contents

Introduction 3

System Architecture and Design 3

Driver 3

Loader 3

Long Term Scheduler 3

Short Term Scheduler 3

CPU 3

Disk 3

RAM 4

PCB 4

Process Queue 4

Logger 4

Compilation 4

Execution 4

Group Organization and Project Management 4

Work Division 4

Data Collection and Analysis 5

Results 5

Expected results 5

Unexpected results 5

Figure 1 9

Figure 2 10

Figure 3 11

Figure 4 12

# 

# Introduction

Our project is to design and implement an OS simulator. This simulator will run on a virtual CPU using a custom instruction set provided by the professor. The OS itself is composed of 10 classes and 3 utility class. These classes are: the CPU, Disk, Driver, Loader, LongTermScheduler, PCB , ProcessQueue, Ram, ShortTermScheduler, and MemoryManager.The utility classes are the Logger, DataFaultException, and PageFaultException.

# System Architecture and Design

## Driver

The driver is the main entry point to our simulator. In this class, we initialize all the other main classes, and begin creating processes. The newly created processes are then given to the various schedulers and eventually to the CPU itself. During the different execution phases of each process, statistics are gathered about the process and are recorded for later use.

## Loader

As the first main component called from the Driver class, the Loader is responsible for loading each process from a text file into the Disk, creating a PCB for that process, filling in the PCB with the process’s information, and handing the completed PCB to the new queue.

## Long Term Scheduler

The Long Term Scheduler (LTS) is called when there are processes that need to be moved from the new queue to the ready queue. When the LTS is first called, it loads the first 4 pages of each process’s instructions into Ram.

## Short Term Scheduler

The Short Term Scheduler (STS) is called when a CPU needs a new process to run. Currently, the STS can choose between three different scheduling algorithms: First In First Out, Shortest Job First, and Priority first.

## CPU

As the core of the OS, the CPU is responsible for executing all of the instructions for a given process. The CPU has the logic to decode instructions, modify internal registers, and manipulate the contents of Ram. A simplistic version of the CPU’s main run loop is as follows: load an instruction, break the instruction into its core components, execute the proper instruction based on the components, and repeat.

## PCB

This is the main data structure in the OS. This class contains all the information about a process: its id, instruction location on disk and ram, the number of cycles it has run, etc. Without this class, there would be no way of representing a process to the OS.

## Process Queue

This is a wrapper class for an ArrayList of PCBs. There’s nothing more to it.

## Memory Manager

This class controls all of the memory the operating system uses. It creates the RAM and Disk and holds all the functions that the RAM and Disk class can use (i.e. read and write). The functions of this class allow for a unified interface between the internal memory structures of the Ram and Disk, and the outside world.

## Logger

This utility class is used as a convenience class to make logging simpler. It allows the OS to output its log messages to either a log file or to the console.

## StatsLogger

This utility class is used to print statistics to an output file so that we can gather it after the simulation has finished.

## Data Fault Exception

This utility class is used to throw an exception when a fault occurs.

## Page Fault Exception

This utility class is used to throw an exception when a fault occurs.

For a diagram of the project, please see Figure 4 at the bottom of this document.

# Compilation

There are two ways to compile the project. If you have Eclipse installed, then you can open the attached source code as a Java project. The project will have everything set up, except for passing in a data file, which isn’t hard. In the run configurations for the project, you have to specify an argument that points to the correct data file. Then hit run, and watch it go.

If you don’t have Eclipse installed, or just prefer to run it from a command line, then compile the classes with javac like you would any other java class. The main class is the edu.spsu.cs3243.Driver class. This class requires that a file path be given to it as an argument. This file path is the path to the data file that contains the processes we will be loading.

# Execution

To see the results of the OS simulator, you must simply run the simulator once. Depending on how the Logger class is configured, the output will either be put on the console, or in a file called “output.log”. This file is a simple text file, so Notepad or TextEdit will open it without problems.

# Group Organization and Project Management

## Work Division

We divided the work up as follows: Robbie worked on the Short Term Scheduler and CPU, Kevin worked on the Long Term Scheduler, RAM and Disk, David worked on the PCB and Loader. The Driver was a collaborative effort. Due to the different programming styles we had some trouble piecing everything together, but we did manage to get them working in tandem. In Phase 2, Robbie continued working on the Short Term Scheduler, CPU, and Loader. Kevin worked on the Long Term Scheduler, RAM, and Disk. Near the end of the Phase 2, Robbie combined the Disk and RAM classes and made a Memory Manager and made the different scheduling algorithms (i.e. SJF, FCFS, HPF) to work with the project. Then both Kevin and Robbie worked on the report.

# Data Collection and Analysis

To collect the data from the simulator, we added some logging that printed out certain counters at specific times, and dumps the process’s information at the end of the simulation. We used these pieces of data to generate the graphs below. There is even more detailed information in the output.log file and the stats.log file, which is obtained by running the simulation. At the bottom of the log file, there is a dump of all the processes in the terminated queue. All of the PCB’s information is in that print out. There are also a few lines that show the total time taken to run the simulation. The reason those lines are not in this document is because the lines are too long, and would word wrap, causing major readability issues and taking up a lot of space.

# Observations and Findings

# Phase 1 Results:

## Expected results

We expected that the processes would run and would complete successfully according the stated expectations in the project description. We also expected that the processes wouldn’t get stuck in any infinite loops during executing, which none did. Another expected result was that the process wait time would be horrible. This is solely because we are executing the processes in their entirety, instead of time slicing or some other technique. In the 2nd phase of the project, this result will disappear.

## Unexpected results

One unexpected result was that the Ram would fill up completely after loading the 15th process. The problem was fixed by clearing the contents of Ram after the 15th process finished executing. This could have been avoided if the Ram had more space, but the project specification was extremely precise on the amount of Ram we could have.

# Phase 2 Results:

## Expected results

We expected that the number of page faults and cycles run will always be the same regardless of what scheduling policies is used. We also expected that the run times on each process will be slower, but the waiting times are a lot shorter than phase 1.

## Unexpected results

There were a few processes that ran for a bit longer than all the others (as shown in the graphs). The reason for this is not known exactly, but it is possible that because of Java’s synchronization mechanisms and the way we used them, that the longer run times were caused by the synchronization overhead.

# Graphical Representation of Results

## Phase 1 Data Discussion

Referring to Figure 1, this graph represents the average cycles run in a single cpu, FIFO configuration. As you can see, the average wait time is quite large. This is because each process must wait on all of the processes before it to finish running before it can run. Also, because there is only one cpu, only one process can run at a time.

Looking at Figure 2, this graph displays the average run time in milliseconds of all the processes. Just like Figure 1 showed, the average waiting time is quite large, due to the serial execution of the processes. Figure 3 reinforces this claim by showing the actual values for the run time/wait time of each process. Even though the run time is very small, it quickly builds up for all of the remaining processes in the queue.

## Phase 2 Data Discussion

The graphs were too big to place in the document, so please refer to the files named “chart\_x.png” in the docs folder. Also, each of the timing graphs were recorded in nanoseconds (using Java’s System.nanoTime()). This was done because during the development of phase 2, the performance drastically increased and measuring using milliseconds because useless. Each chart has also been aggregated to show all the scheduling algorithms used in one graph, instead of 3 separate graphs. I will also point out that we didn’t actually measure the differences between paged and non-paged memory, though I will make some theoretical comments at the end of this section. All of the graphs in phase 2 are using paging.

Chart\_1 displays the wait time for each process running in a single core environment. The wait time was calculated by keeping track of how much time each process spent on the ready queue. A process would be pushed to the ready queue if it caused a context switch, or if it hadn’t been run yet. Looking more closely at the SJF algorithm’s data, it appears that programs with smaller instruction sizes didn’t have to wait as long as others. It is interesting though, that there were multiple processes that waited the same amount of time near the end of execution.

Chart\_2 displays the run time for each process running in a single core environment. The run time was calculated by keeping track of the amount of time the process ran on the cpu. Something interesting that this graph shows is that on average, most processes took the same amount of time to run, regardless of what scheduling algorithm was used. Another interesting point of data is that we have a few processes that took MUCH longer to run than the others. As I stated above in the Unexpected Results section, I believe that this might have something to do with the synchronization mechanisms in Java.

Chart\_3 shows the fault time of each process in a single core environment. This was calculated by summing all the time each process spent in a faulted state. Not very surprisingly, the scheduling algorithm has little to no effect on the time spent in this state. The one random process that spent a really long time faulting I will blame on Java ☺.

Chart\_4 shows the waiting time of each process in a quad core environment. A surprising result of adding the extra cores is that overall, the processes waited longer than the single core processes, but it is possible that is just the synchronization calls causing this. I do believe that synchronization has a big effect on this time because instead of there being a single thread accessing the critical sections, now 4 threads are trying to access those sections at the same time.

Chart\_5 shows the run time of each process in a quad core environment. Overall, each process seems to have run for about the same amount of time, except for the few outliers, which I cannot account for the exact reason why. I can say though, that adding more cores did not really help the overall run time of each process. That is determinate on the process size, not the number of cores. It did, however, actually slow done the run time of the processes overall, though I believe that this is only a side-effect of having multiple threads trying to access critical sections of code and the overhead associated with that.

Chart\_6 shows the fault time of each process in a quad core environment. Overall, each process appears to not have waited too long in a faulted state, except, of course, for the few random outliers. I really wish I knew why they kept showing up… However, it appears that by adding more cores, we actually spent more time in a faulted state then when we only had one core.

Chart\_7 shows the amount of ram (in pages) used over the course of the quad core simulation. This data was gathered by recording how much memory was left over after each paging operation (read or write). At the beginning of the simulation, we load 4 pages of each process into ram, and then more are loaded as needed. The drops you see are when a process has been terminated and has released the pages it owned. The only reason I can give for the FIFO algorithm using more pages than the others is that the first number of processes required a large amount of pages before they could finish. It still doesn’t quite explain all of it, but I think it helps.

Chart\_8 shows the amount of ram (in pages) used over the course of the single core simulation. We can see the same general trends in usage as with the quad core graph, though interestingly SJF has more large drops then the quad core graph. Also, the priority line in actually lower than it is with multiple cpus. I can only guess that this is because more processes are requesting pages faster in the quad core simulation than in the single core simulation.

Now for the theoretical comments on paged vs. non-paged memory. Paged memory is supposed to be much more efficient with memory than non-paged memory. This is actually proven (even though I don’t have a graph to support this, just raw running’s of the simulation) by looking at the code in phase 1 compared to phase 2. In phase 1, after we ran ~15 processes, we had to completely clear out the ram so we could start over again at index 0. With the paged memory system, we don’t have to do that anymore. This is because the non-paged memory system worked by loading all of the process into memory at once, therefore taking a large amount of space to load only a few processes. With the paged memory system, it has been shown with the supplied graphs, that we never actually reach the maximum amount of memory (we do come close though with FIFO…). This is because only parts of the process are being loaded into memory at once, and then a process has to request extra pages. Another key difference in the two models is the clearing out of the memory when a process is terminated. The paged system allows for this, but the non-paged system doesn’t. In order to allow the non-paged system to do this, we would have to determine the largest process we loaded, and allocate exactly that much space to each process in ram. That way, we can easily replace terminated processes when they finish. However, this is not an optimal solution at all.

# Conclusion

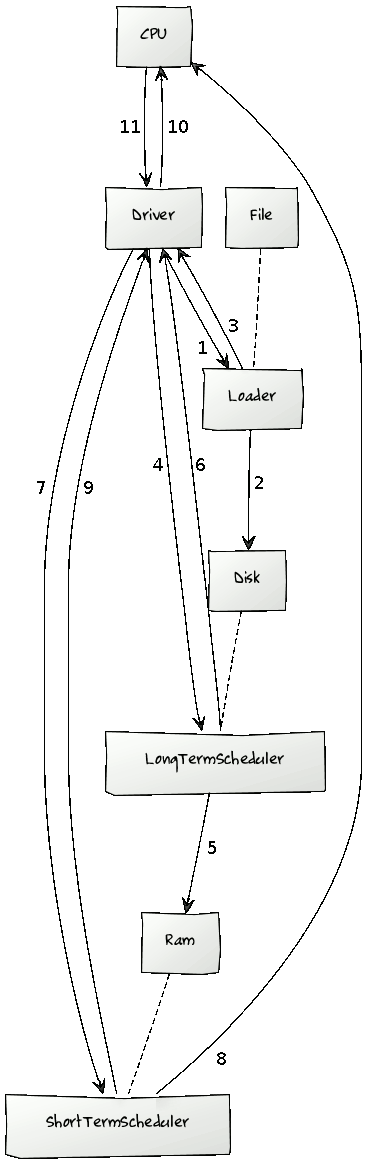
In comparing the run time, memory usage, and other performance factors of the different simulations, I believe the most optimal solution would be to implement a system that had: paged memory that supports removing terminated processes; a combination of scheduling algorithms, for example SJF & Priority with Aging; and a multi core system, though anything past 4 just seems silly. I believe I can say this with confidence based on what I have learned in class and what the simulation has shown me. However, when implementing a multi core system, great care would have to be taken in the critical sections of the system. As I believe my simulation has shown, improper use of the synchronization mechanisms in Java has caused a slow down in performance.

## Phase 1:

Figure 1

Figure 2

Figure 3



Figure