Phonepaseuth Viengxay,Ricky Viengxay,Dennis Lin

Kennesaw State University

February 28th, 2019

CS 3502, Operating Systems

Introduction

The purpose of this document is to inform you on our ideas and thought process on how we tackled this Operating Systems project. As a group we collectively decided the most optimal IDE to use for this project was C++, because of its strong pointer functionalities. Because this is a hybrid project, we have decided to use GitHub as our source to share and modify files. In our initial meeting as a group we decided that the most optimal way to approach this project was to assign each member a part of the Operating System. The in-class students worked jointly together as communication was not much of a barrier for us, so the F2F we collectively worked together on the CPU, Loader, and scheduler. While it is imperative that we’re in constant communication, we assigned the online students the parts of the OS, that could be worked on independently with minimal communication like the decoding, fetching, and memory-based part of the project. During our initial meeting as a group we decided on one way to implement all our functions and pointers for example, the ready queue will be implemented using a link list that will point to each process based on priority or first in first out. Our initial plans were to get the Operating system running on the basics first, like for example for our short-term scheduler, we are going to do an SJF algorithm just to make sure everything is running fluidly before divulging into another algorithm like a round robin. Our plan was to get the Operating System up and running by February 9th and then coming together as a group to collectively polish up the code and algorithms.

As stated above, we decided to write our code in C++ because we felt like the use of pointers was going to come in handy when dealing with the multiple attributes within the CPU. As we learned in class our CPU should be the main driving force, within our CPU.cpp class the CPU, loads/unloads/executes processes, decodes/fetches instructions, and we also made a method which converts Hex into Binary. The next two important parts of our code are the loader and the disk. Within our loader class, we created three strings which determined if line of instructions given to us from the program-file is a Job, Data, or End. After doing so, we parsed through each line of the code using the string-stream function and broke the string into a vector so that we could separate the Job’s Job number, priority, amount of instructions, and the index of the job. After determining these attributes, we write this into the PCB and the disk for later use. Now our disk holds our PCB and instructions to be chosen by the Long-term Scheduler. The Long-Term Scheduler takes the first 1024 instructions and puts them into a list, which will then be passed to ram. The scheduler checks whether the ram is full or not, from a method in the RAM that returns a Boolean true if it is full and false if it’s not. If not full, the scheduler continues by adding in the instructions into ram which is the size 2048. Once ram is full, method is called to put the process into the ready queue. From this point we have a method within our short-term scheduler that acquires the PCB from our ram. We decided to implement a first in first out scheduler so that we could test all our initial inputs, and in phase two we will change our short-term scheduler to either a round robin or shortest burst time first. Once selected by the short-term scheduler, the scheduler calls upon the dispatcher to set the PCB within the CPU for execution. The CPU fetches instructions from the ram, where it will be decoded by using the stoi function within C++, which converts Hex characters into binary. From there our decoder reads the substring of the instruction and executes based off the format provided (00,01,10, etc) and sets the opcode. Following the setting the op code, it goes into the execute method where it’s the opcode is compared to a case depending on what job it’s supposed to do. After a process is done executing, the process will be unloaded (terminated) through context switching and the next process will be loaded in. This occurs until all processes are loaded and we can exit our loop.