OS SIMULATION REPORT

Author: Stefan Murga
Student Number: 100976625
Course Code: SYSC 4001
Professor: Gabriel Wainer
Submission Date: 22 December 2016
Partner: Nwakpadolu Soluzochukwu John

ABSTRACT

This report discusses the structure of the operating system simulation, and its results, which was developed as a final project for SYSC 4001 at Carleton University. The software was written in C++ and uses boost libraries. The simulation is modelled in accordance with PDEVS formalism. The project was carried out in a team of two.

The developed software simulates essential components of an operating system. In the simulation there are models for the kernel, memory manager, main memory, disk storage, system call interface, read/write functions, and programs. The models are event based and respond to external messages from other models. The kernel can be directed to use the first-come first-serve (FCFS) or priority queue algorithm to schedule the simulated processes. New processes are allocated an address space on the disk storage. The system makes use of virtual memory and swapping to allow many processes to operate simultaneously with a relatively low main memory size. Processes in the simulation are able to make read, write, exit, and fork system calls. The system call interface will then call upon the proper components to complete the requests. Together, all these components simulate the workings of a basic operating system.

Table of Contents

1.0 INTRODUCTION

This report discusses the elements of a developed operating system simulation. The aim of developing the simulation was to explore and learn about the essential components of an operating system. The simulation was developed in C++ and makes use of the boost libraries. The simulation structure follows PDEVS formalism.

The next section of this report will explain, in detail, the features and components of the simulation. There will be a discussion on each individual component, and how they interact with each other. Following the simulation explanation, this report will outline some of the challenges that arose during development and their solutions. Next, there will be a discussion and analysis of the simulation results. The report conclusion will provide a summary of the report findings and will examine further development and usage of the simulation.

2.0 SIMULATION STRUCTURE

The simulation is made up of seven atomic models: kernel, memory manager, main memory, disk space, program, system interface, and read/write function. The following sub-sections will describe each atomic model in detail.

The simulation models interact with each other by passing messages. Each message contains a destination port and seven optional values. Different messages make use of the seven optional values differently. Models can have multiple ports that it writes to or reads from. The port name defines the message type. As an example, a message with the port name "readSCI" is sent out by a program to ask the system call interface to perform a read from memory. For the most part, the models are all event driven. They wait indefinitely until they receive a message. The handling and response to said message is usually performed instantly. Then the model waits for the next message. Figure 1 in the appendix shows an overview of the how the models are connects and the messages ports that are used.

When executing the simulation from the command line, two arguments must be present. The first argument will be used to tell the kernel which scheduling algorithm to use. The second argument is the path to an input file. Input files are used to tell the kernel with new processes arrive and are ready to be scheduled. In effect, the input file acts as a sort of long-term scheduler for the kernel.

2.1 Kernel Model

The kernel model can be thought of as the brains of the simulation. It schedules processes, handles fork and exit system calls, handles messages generated from the input file, and maintains a PCB table of all active processes. The kernel is entirely event based and all actions are carried out in 1 time unit. When calling the kernel model object constructor, the type of scheduling algorithm to use is passed as a parameter. The kernel can use either the FCFS or priority queue scheduling algorithms.

The two scheduling algorithms classes inherit the same interface making it easy to dynamically choose one. The kernel only knows that's it's using a scheduler with two functions: schedule and getNext. The kernel maintains a PCB table that tracks the PID, state, type, and priority of all active processes. It is the process's PCB that gets passed to the scheduling algorithm. The FCFS algorithm simply makes use of std::queue<PCB>. The priority queue algorithm is implemented as a linked list.

The kernel requests memory for new processes, creates an entry in the PCB table, and sends their PCB to the scheduler. Whenever the CPU is made idle, the kernel will ask the scheduler for the next process that is scheduled for a CPU burst. The kernel will be made aware when that process starts I/O or completes. The kernel will also request that the memory set aside for a process be de-allocated when that process finishes.

In the simulation there is only one kernel object that handles the scheduling and memory allocation/de-allocation of all processes in the system.

2.2 Main Memory Model

The memory model is implemented as a vector of integers. When processes perform I/O they can read/write integers to the main memory. In this simulation, the memory size was set to be low to showcase the virtual memory feature. This topic will be discussed further in the section explaining the memory manager model.

Frames, five integers in size, can be swapped from disk storage to the main memory, and vice versa. The main memory model plays a passive role in the simulation.

2.3 Disk Storage Model

The disk storage model is also implemented as a vector of integers. In this simulation, the disk size is set to be much larger than the main memory size.

Process read/write functions do not interact with the disk space directly. The only events that the disk model supports and the allocation of disk space for new processes, the de-allocating of disk space for terminated processes, swapping a frame from the disk into memory, and swapping a frame from the memory into disk space.

2.4 Memory Manager Model

In the simulation there are two types of memory, main memory and disk storage. The memory manger coordinates the two and handles requests from read/write functions to translate a process' logical address into a physical memory addresses.

Each process has its own logical address space that starts at index zero, the size of which is specified at process creation. When a process requests a read or write, it provides the logical address to perform the operation on. The system function will send a message to the memory manager to translate the process logical address into a main memory address.

Read/write functions only operate on the main memory level, not the disk space. The memory manager is responsible for coordinating the swap-in and swap-out of memory frames (or pages) between the main memory and the disk storage. Virtual memory is implemented using these swapping functions. The manager also determines if the memory for new

processes can be allocated. It will search for the next continuous hole that can fit the space requirements for the process.

2.5 Program Model

The program model is made to simulate a user application. The program interacts with the operating system by making requests to the system call interface. Each program has its own logical address space. As such, programs don't care at all where on the disk their data is stored.

There are two types of simulated programs. The "I/O program" performs five cycles of a write system call followed by a read system call. After starting a CPU burst, these programs will wait a random time interval before calling on the I/O function. After their five cycles, the programs will call make the exit call to the system call interface. The other function is can be referred to as the "forking program". This program only has one CPU burst. In it's burst, it will make 15 fork system calls, spread out by random time intervals. This has the effect of creating and adding 15 more processes to the kernel scheduler.

The simulation supports up to 20 active processes at once, there are 20 program model objects built into it.

2.6 System Call Interface Model

The system call interface acts as a medium between user application and the operating system. As long operating system updates conform to the same system interface, they will maintain compatibility with older user applications. The system call interface is also a safe guard against malicious or poorly written code when accessing critical operating system resources.

In the simulation, the system call interface listens to the running processes and handles their requests. The interface will forward the requests the to appropriate model. For example, a fork or exit system call will be forwarded to the kernel.

The implementation of the interface is straightforward. It listens for messages and forwards them in the same time unit. There is one interface in the simulation that handles the system calls for all active processes.

2.7 Read/Write Model

The read/write models simulate a process requesting to read/write to memory. A construct parameter determines if the object acts a read or write system function. For simplicity sake, each active process has its own read and write functions. That means there are twice as many read/write models as there are program models.

When a process requests a read or write, it will pass the logical address it want to interact with as a parameter. The system function will query the memory manager for the physical memory address where that data is located. After getting the memory address, the system function will directly read/write to the main memory model.

The systems functions will also notify the kernel when starting and stopping. The kernel uses this information to schedule and start other processes.

3.0 DISCUSSION

3.1 Virtual Memory

The creative component that I added to the simulation is the implementation of virtual memory via page swapping. Virtual memory is a memory management technique that allows a process to execute without it's entire address space loaded in memory. As different parts of a process' address space are required, they are swapped into the main memory from the swap space on the hard disk. Figure 2 in the appendix shows how the main memory can be mapped to different parts of the disk.

In this simulation the memory size was set to 20 units and the disk size was set to 10,000 units. This is done to illustrate the benefits of virtual memory. Without virtual memory, only one or two processes could be executing since their entire address space would fill the small memory. After adding in virtual memory, the number of processes that can be concurrently executing is significantly higher. The other benefit of virtual memory is that each process gets its own logical address space. This means that the processes don't care about the implementation details of where their data is actually stored in the system. This means that application developers don't need to worry about using memory addresses relative to the start of their allocated memory block.

The memory manager model handles all the swapping related functions that make this possible. When a system read/write function requests a logical address be translated to a physical memory address, the manager will check to see if that data is already in the memory. If it is, the manager will simply return the address of the data in memory. If the data isn't already in memory, the manager will chose a frame of memory to write out to the disk. Then it will swap in the disk frame that contains the desired data. In the simulation the frame size if five integers. This way, if a subsequent read/write function occurs there is the possibility that the desired data might already be in memory.

In this simulation, there are only considerations for a process' data. In reality, a process' address space also contains it's own program instructions. These program instructions would also have to be swapping in as required. Ideally, they would be swapping in just before being required.

Today, most operating systems make use of virtual memory. The notable exception is embedded systems, which place a higher priority on speed of memory access.

3.2 Challenges

One of the major challenges that I encountered was trying to understand the simulation framework. I previously had no experience with DEVS or PDEVS formalism. To overcome this challenge I did plenty of research on the web and frequently referenced the ABP simulation example that was provided. For future classes that attempt this project, I believe some sort of documentation provided with the framework would be extremely helpful.

Another challenge that I came across was debugging. When I saw incorrect values or had segmentation faults I would try to debug the simulation in GDB. Any debugging attempts that I made were useless. The debugger kept jumping lines and would throw segmentation faults in different areas depending on where I set my breakpoints. I tried setting the compiler optimization to zero, but the results were the same. To overcome this barrier I had to result to printing to the screen for all my debugging needs.

The last challenge that I will mention is when I had an issue with re-occurring messages in the kernel's external function. The kernel seemed to be reacting to multiple messages of the same type, but I could only see that I was being sent. I tried changing many things before I actually fixed the problem, and it was only by luck. I had the kernel responding to all of

its external events in the same time unit (ie. next_internal was set to 0). I changed it to respond one time unit later. That solved all the issues. The problem must have been that too much was going on in the same time unit and there was possibly concurrent reading issues. The PDEVS formalism dictates that the order of messages that arrive in one time unit is not guaranteed.

3.3 Analysis

Running two processes concurrently using the FCFS scheduling algorithm keep the CPU consistently busy. Despite being random, the CPU bursts of processes were always longer than the I/O bursts. This meant that when one process was about to start I/O, the other would have finished its I/O and would be waiting in the scheduler queue. The two processes each did their cycle of five write/read system calls, and then did an exit system call.

Scaling the simulation to 20 processes running concurrently using the FCFS scheduling showed similar results. Of course, the CPU was always busy. It was interesting to note that the output of events showed the same system calls happening for each process in the order that the processes were started. There was never any variation, and they processes all terminated around the same time. This is a symptom of using the FCFS scheduling algorithm. Results could have been different if the time interval for process I/O bursts was random. With random I/O bursts, one process could finish its I/O before that started I/O before it. This would move that process up in the scheduling order. The process all did their five cycles of write/read, and then exited.

To test the forkexec system call, the input file only told the kernel to start one "forking" process. This process then forked 15 times to start the other processes. To fork, the process makes a call to the system interface. The interface forwards the request onto the kernel. The kernel then searches the PCB table to find a free spot. If there isn't a free spot the request fails. Once a free spot is found, the kernel asks the memory manager to allocate space on disk for the process. The memory manager searches through the disk space to find a hole large enough to meet the specified

space requirements. If a hole was found, it would be blocked off and recorded as being used by the new process.

The process that forks would continue executing and finish in one CPU burst. After that process exited, the other 15 processes would continue on using FCFS algorithm. The start of the output file showed the first process making fork system calls and all the messages that were passed to create the new processes. After the first process terminates, the output file looks the same as a simulation with 15 processes starting at the same time would look. Each process performed a write, and then each process performed a read. This cycle happened five teams before each process exited.

It would have been interesting to see the results on a multi-core simulation or a round-robin scheduling algorithm. That way output from the parent and child processes would appear to be concurrent instead of the parent finishing before the child starts. A round-robin scheduling algorithm would accomplish this by pre-empting the parent process.

The initial version of the memory manager simply marks array elements as taken when they were "allocated" to a process. This is a very basic implementation and the only implications it had on the overall simulation is that a process might not be create if a hole of the appropriate size couldn't be located. Creating processes with varying memory size requirements tested this. When a suitable hole couldn't be found, an error message was printed and the process wasn't sent to the scheduler. In effect the process just died right there. A better implementation would have been to add a queue that would hold processes that a memory hole couldn't be found for. When memory was release it would re-attempt to find it a suitable hole.

Following the output file it can be see that when a new process is being created by the kernel, the kernel makes a request to the manager to allocate space. If the manager says the operation was successful, the kernel finishes creating the process and sends its PCB to the scheduler. In the event that the manager sends a return message saying the operation was not successful, the kernel doesn't finish creating the new process.

The second scheduling algorithm that was added was the priority queue. A scheduling interface class was created to abstract it implementation. The FCFS algorithm class and the priority queue algorithm class implement the interface. This allows the kernel model to be able to dynamically choose which to use, based on the command line argument presented. To test the priority queue algorithm the simulation was ran with multiple processes that all started at the same time. It was easy to see that the algorithm was working by filtering the output to only see when the processes completed. The processes that had a higher priority finished well ahead of processes that had a lower priority.

The second iteration of the memory manager made it actually usable and added virtual memory. Simulated processes could now write to the memory, and then read back the same value at a different time in the simulation. To test this, the input file instructed the kernel to start multiple processes. Each process would write a random value somewhere in its address space, and then read it back on a different I/O burst. Reading through the output file, you can see a process sends a write system call to the system call interface and provides the logical address to write to. The interface forwards the request onto that process' dedicated write function. Then, the write function asks the memory manager to translate the logical address into a physical memory address. Since this is the first time the process is trying to access that piece of memory, it isn't loaded in the main memory. This forces the memory manager to swap in the desired frame from disk space to the main memory. Once that is complete, the memory manager sends a message to the write function containing the address of memory that holds the data. The write function then writes a value to memory. While this I/O is taking place, the next process is performing a CPU burst. Once all of the processes have completed their first round of CPU bursts, the process that we've been follow tries to read back the value it had previously written. Inspecting the output file, it can be seen that a similar process occurs. The process makes a read call to the interface, the interface forwards to the read function. The function asks the memory manager to translate the address. The difference is that this time when the manager determines the requested data is not already in the memory, it has to swap a frame of data out of the main memory and onto its place on the disk. This is because now all the frames in the

memory are filled with data from other processes. A frame has to be chosen to be swapped out before swapping in, or else the data of another process would be lost. In this simulation, the memory manager chooses which frame of memory to swap out by running a looping counter. In an ideal world, the memory manager should swap out the frame that won't be used again or the one that will be used furthest from now. After performing the swaps, the memory manager sends the memory address back to the read function. The read function then reads the value directly from memory. Seeing this process was pretty awesome and showed how clever operating systems can be.

4.0 SUMMARY

The simulation successfully demonstrates the tasks that are preformed by essential operating system components. Seeing all the elements communicating with each other and operating on queue was very refreshing. This basic model just goes to show how complex an operating system is.

A cool idea to add to the simulation would be inter-process communication. The idea would be to allow two or more processes to talk to each other using messages queues or sockets.

5.0 APPENDIX

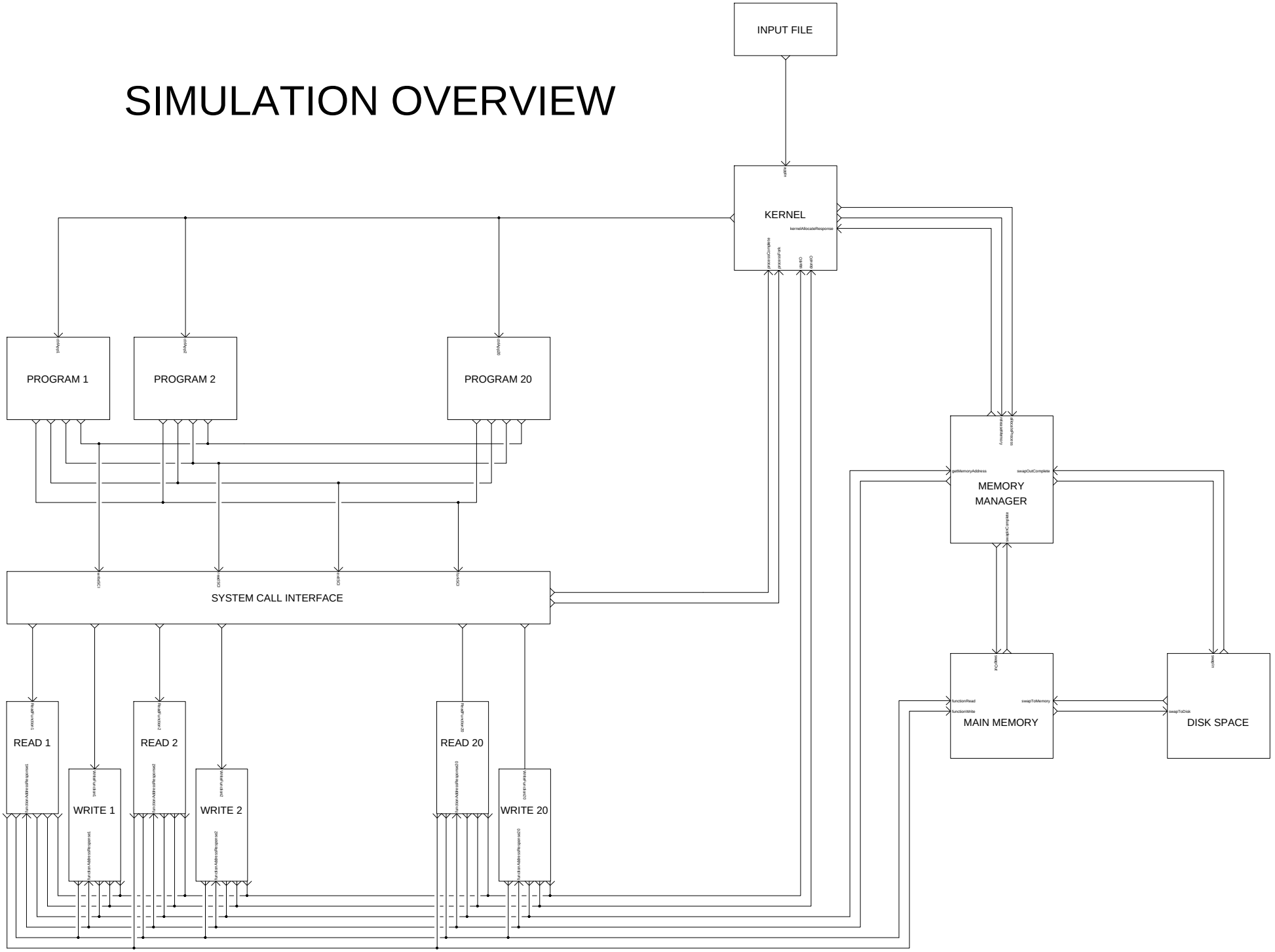
Figure 1: Simulation overview.

This figure provides an overview of how all the models are interconnected. The lines connecting the models are the paths that messages take. On receiving end of each message line is the name of the message port. There are little arrows on each end of the message lines to denote if the message is input or output for the model it connects to.

Figure 2: Virtual memory.

This figure shows the main memory and a portion of disk storage. The disk storage has allocated space for three processes of varying size. It can be seen that each process has its own logical address space that begins at index zero. Processes address their data using these addresses. The memory manager keeps track of the disk address that each logical address is mapped to. In the figure it can be seen that each frame of the main memory contains a different frame from memory. These represent frames from disk that have been swapped in. Without virtual memory and swapping, the memory is only large enough to support one of those processes.

SIMULATION OVERVIEW



VIRTUAL MEMORY