**CS-3243 Operating System Fall’11 Prof. Patrick Bobbie**

Cory Richerson

Gabriel Gilmour

Robert Harris

Resubmission..talked with Professor Bobbie Monday after class 11/7 where he allowed us to add our core dump and waiting time data within the project and our documentation.

**Introduction**

Operating systems are complex and their concepts difficult to understand. This project was designed to help us gain further knowledge of operating systems, by forcing us to think critically about the design and implementation techniques. This report will cover our approach to the system architecture and design, the components we implemented for the project, our observations on threading, and our conclusion.

Object-Oriented techniques were used for the system architecture and design to implement the modules that make up the system. This allowed a nearly one-to-one mapping of the physical and virtual elements into the design.

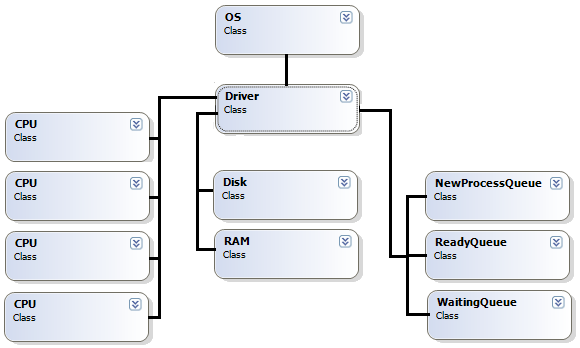
The code was written in C# while using Microsoft Visual Studio 2010 to compile and manage the project.

**System Architecture and Design (Configuration)**

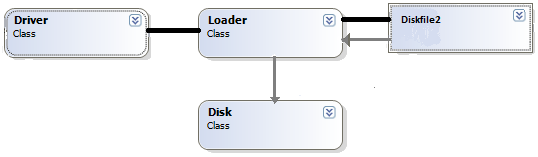
After reviewing the project specification document, we wanted to decide on our operating system structure. We ruled out the simple structure approach, used by MS-DOS, due to the complexity in implementing and maintaining its structure. We then decided against the layered approach because it required such careful planning and critical thinking, that it seemed one component in the wrong layer would cause the operating system as a whole to come crashing down. The microkernel approach seemed promising, but the performance problems were undesirable; so we decided to go with the modular approach. We chose the modular approach because it uses object-oriented programming techniques, which every group member is familiar with, to create the kernel.

We divided the operating system into components outlined by the project specification block diagram. There were 6 main components used: the CPU, driver, loader, PCB, long-term scheduler, and short-term scheduler. Once the components were decided on, it was just a matter of choosing a programming language. We ended up choosing C# because we were all comfortable and familiar with the language. The main components we used will be discussed further in the following paragraphs.

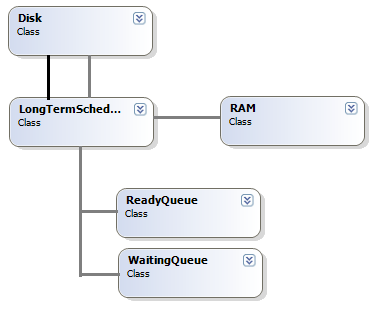
Our operating system starts with our OS class file, which basically just starts the operating system by declaring an instance, called kernel, of our Driver class. Our kernel, in this case the Driver class, is considerably small since most of the code resides in the other classes. The kernel instantiates the new process queue, the ready queue, the waiting queue, disk and ram. The kernel also uses an integer value of 4 to control the number of CPUs generated for threading. Once the kernel has everything it needs, it calls the run method for the loader.



The loader is how we read in the text files into our virtual disk. After stripping off the job cards/data cards, we created a method to convert the input data to an unsigned integer in our SystemCaller class to be stored in the PCB (the method is called ConvertInputDataToUInt).



The long-term scheduler is responsible for taking the data from disk and writing it to memory. Our long-term scheduler first gets a batch of 15 jobs and then sorts it. Since this phase of the project didn’t require any sorting algorithms, besides FIFO (first in first out), we merged the long and short-term schedulers. After the long-term scheduler sorts the jobs, the jobs are inserted into our virtual memory. Once the jobs are in virtual memory, the long-term scheduler puts the new processes into the waiting queue and the waiting processes into the ready queue.



The CPU was the probably the most complex and difficult component of the whole project. As stated in the instruction format document, there are 16 registers in use throughout the program execution. The CPU first checks for new processes to execute by checking the ready queue. Once the CPU has a process, we use a while loop in conjunction with a Boolean method called HasProcess. While HasProcess is set to true, meaning while the CPU has a process, the while loop is set to execute its three methods until there aren’t any more processes left in the ready queue. The first method of the while loop is the Fetch method. The Fetch method just fetches the process info from the PCB. The Decode method then formats the instructions for the CPU to use in the execute method. The Execute method, using a switch statement, performs the appropriate operations based on the instructions from the Decode method. Once all the processes have been run, the CPUs while loop exits.

The PCB is, without a doubt, one of the most important parts of the whole operating system project, since it is used every step during runtime. Without the PCB, it is easy to see how difficult it would be to read in data and keep track of a process’ information. Our PCB stores the process state, process ID, priority, job length, the program counter, the memory address and the registers. The process state is used to determine when the process is ready to be moved to another queue or to let the CPU know a process is ready to be executed. The process ID, priority, and the job length are all read into the PCB when the loader is called. The program counter is necessary because it keeps track of the next instruction to be fetched. The memory address is important verify that we allocated the right amount of memory for each process. The registers are needed to store the instructions from the jobs for them to be processed.

Threading was an interesting concept for us to tackle, and it is important to note that we don’t have a memory management unit (MMU) yet. As discussed in class, a MMU is a unit to help regulate and control access to shared resources. The reason we don’t have an MMU is because the locks that control access to the shared resources are handled in the various classes. As stated earlier, the Driver class has an integer value to control the number of CPUs generated. Once the CPUs are created, a thread is created for each CPU and executed. When the CPU checks the ready queue for a new process, it locks the ready queue so other CPUs can’t access it at the same time. With the ready queue locked, the CPU then checks the cache, locks the cache and reads the process’ data stored in the PCB from memory. The process again repeats until there aren’t any more processes to run. For the second phase of the project we will be adding a MMU to make the project easier to handle.

Once all those components were up and running we ran the jobs. The results are at the end of this report.

**Code and Compilation**

**CPU:**

**CPU.cs** - An enumeration type is used to define the Instructions and Instructions Types and are used by the Decode and Execute methods for execution. The kernel, disk, ram, ready queue, pcb, and current process to be executed are linked to the cpu. The thread, timer, ready queue, and cache lock objects are declared as well as the methods needed to Start, Pause, and Resume the CPU. The cache is declared which is used to hold the instructions and buffers during execution and is filled using the Fill Cache method . The primary logic for execution of the CPU is within the Run method, this is where the Fetch, Decode, and Execute methods are called while the CPU has a process. The Fetch method assignes the current insturction and increments the program counter. The Decode method analyzes the necessary hex characters to assign the approporiate insturction and instruction type as well as setting the source and destination registers. The Execute method uses the instruction and instruction type to perform the requested operations. Once execution is complete the Process PCB is saved and the process is put back into RAM.

**Kernel:**

**Driver.cs** – This is where most of the ‘hardware’ is declared, the number of cpu’s is set, and the execution of the OS is performed. The RunOS method uses the number of cpu’s to create them and put them into a List. The Loader and Long Term Scheduler are called before the processes are executed in two batches.

**Loader.cs** - This class is used to read the data file to disk as instructions and data, while also creating the processes and putting them into the New Process Queue. The Kernel, Disk, RAM, and New Process Queue are linked; variables are declared for use. The Run method calls the ReadProgramFile method which reads in the data file line-by –line to either: Interpreting the Card line, write instruction/data to disk, or add Process to the NPQ.

**OS.cs** – This class creates a Kernel object and calls the RunOS method starting the OS. This has to be the smallest Operating system ever created, only 1kb!

**SystemCaller.cs** – this class is used to convert data to various formats, display / output the contents of the disk and RAM, and send information to the console for observation.

**Processes:**

**PCB.cs** – This class contains the registers and data needed to handle the processes. Enumeration types were also used here to represent the various process states. Data includes, but not limited to the following: process state, waiting time, I/O count, process ID, priority, instruction length, buffer sizes, data addresses, job lengths …

**Process.cs** – Encapsulates a Process Control Block, adding a get and set method, and assigning the ID.

**Queues:**

**NewProcessQueue.cs** – Encapsulates a Queue.

**ReadyQueus.cs** – Encapsulates a Queue.

**WaitingQueue.cs** - Encapsulates a Queue.

**Scheduling:**

**Dispatcher.cs** – This class assigns a process to a CPU from the Ready Queue and sets the CPU PCB. The class constructor links the kernel and Ready Queue.

**LongTermScheduler.cs** – This class gets the batch, inserts it into memory and manages the queues. The class constructor links the hardware using the passed in kernel parameter. The Run method calls the various methods to move the processes into the various queues.

**Shared Memory:**

**Disk.cs** – An array of size 2048 to hold all of the instructions and data. There are three methods to Write to the disk, Read from the disk, and Get the Disk Size.

**RAM.cs** - An array of size 1024 to hold a batch of the instructions and data. It has a Locked attribute which has a get and set method. There are three other methods to Write Data to, Read Data from, and Get the Memory Size.

**User Guide**

Two files need to be changed to correctly read in DataFile2.txt and output the core dump files. The two files are located in the ‘Kernel Stuff’ folder. The Loader class needs one change, while the SystemCaller class requires two changes (highlighted below). Once the project is run, the output files will be located in the OS\_PROJECT file and will be overwritten with each successive execution.

**Loader Class file line 44:**

**StreamReader file = new StreamReader(@"C:\Users\Me\Desktop\OS\_PROJECT\DataFile2.txt");**

**SystemCaller Class file line 60:**

**StreamWriter writer = new StreamWriter(@"C:\Users\Me\Desktop\OS\_PROJECT\CoreDumpBatch" + batchNumber + ".txt", false);**

**SystemCaller Class file line 92:**

**StreamWriter writer = new StreamWriter(@"C:\Users\Me\Desktop\OS\_PROJECT\CoreDumpProcsBatch" + batchNumber + ".txt", false);**

**Data Collection and Analysis**

We collected several sets of data for both our single-threaded CPU system and our multi-threaded CPU system: I/O count, output to memory/disk, completion times, and waiting times. I/O count and output are virtually irrelevant for the sake of data analysis, unless of course they were to not match, which is not the case.

We found it, generally, hard to compare the two system setups given two reasons we have come up with: the lack of an extensive data set and the actual virtualization of the system itself. Although we manage resources and threads within our own virtualized OS, our virtualized OS is still being scheduled/managed by the .NET framework and physical computer system, and not directly by us. This threw off our result set from our expectations.

We expected the multi-threaded virtualized OS to be, in general, a large improvement over the single-threaded CPU system. We expected the completion times for each process to stay the same across the board, and for the multi-threaded system to outclass the single-threaded system in terms of waiting times. This was not exactly the case.

For completion times, the multi-threaded CPU system performed as expected—that is, the completion times for each process were low, ranging from less than 1 ms to about 5 ms. However, the single-threaded CPU did not perform the same as we expected. Each completion time, from the beginning process of each batch, scaled linearly, beginning from about 4 ms up to 15 ms. This did not make sense to us, as the same CPU structure should run each job the same, and thus, achieve the same completion times. Also worth noting, despite the single-threaded CPU system having much higher completion times per process, it was much faster in actually completing all jobs in each batch. We can only deduce that these occurances are somehow due to our virtual OS threads competing with other threads on the physical system.

Similar strange results were found when comparing waiting times between the two systems and against our expectations. We expected our multi-threaded system to destroy our single-threaded system in terms of waiting times per process, but once again, our dreams were shattered. The multi-threaded system (on per process terms) sometimes had initial waiting times that were far below the single-threaded system, but would often have times that far exceeded it. Frequently the first four jobs would have next to no waiting time, whereas the last four would generally wait nearly 30% longer than their single threaded cousins.

**Conclusion**

After working on this operating system project, we all have a better understanding of how an operating system works and the complexities that come with it. Although phase one of this project is finished, we realize there are many different ways to improve our operating system. The most obvious way to improve our project would be to add preemption by way of a scheduling algorithm such as the shortest job first algorithm, or the priority algorithm that is required in phase two. Either of these scheduling algorithms would help with our completion times and would not take much on our part to implement. The addition of a MMU would also benefit our project, as mentioned earlier, and will be necessary for the completion of phase two.

**Single Threaded Results: I/O Count and Output**

|  |  |  |
| --- | --- | --- |
| Job Number | Number of I/O operations | Output |
| 1 | 12 | 228 |
| 2 | 12 | 85 |
| 3 | 12 | 56 |
| 4 | 12 | 0,1,2,3,5,8,13,21,34,55,89 |
| 5 | 12 | 170 |
| 6 | 12 | 79 |
| 7 | 12 | 0,1,1,1,1,1,1,1,1,1,1 |
| 8 | 12 | 0,1,2,3,5,8,13,21,34,55,89 |
| 9 | 12 | 56 |
| 10 | 12 | 85 |
| 11 | 12 | 868 |
| 12 | 12 | 225 |
| 13 | 12 | 56 |
| 14 | 10 | 0,1,2,3,5,8,13,21,34 |
| 15 | 11 | 53 |
| 16 | 5 | 0,1,2,3 |
| 17 | 12 | 177 |
| 18 | 12 | 56 |
| 19 | 10 | 69 |
| 20 | 12 | 56 |
| 21 | 13 | 0,1,2,3,5,8,13,21,34,55,89,144 |
| 22 | 12 | 85 |
| 23 | 11 | 143 |
| 24 | 12 | 56 |
| 25 | 7 | 127 |
| 26 | 8 | 0,1,2,3,5,8,13 |
| 27 | 12 | 225 |
| 28 | 12 | 225 |
| 29 | 12 | 80 |
| 30 | 11 | 0,1,2,3,5,8,13,21,34,55 |

**Multi-threaded Results: I/O Count and Output**

|  |  |  |
| --- | --- | --- |
| Job Number | Number of I/O operations | Output |
| 1 | 12 | 228 |
| 2 | 12 | 85 |
| 3 | 12 | 56 |
| 4 | 12 | 0,1,2,3,5,8,13,21,34,55,89 |
| 5 | 12 | 170 |
| 6 | 12 | 79 |
| 7 | 12 | 0,1,1,1,1,1,1,1,1,1,1 |
| 8 | 12 | 0,1,2,3,5,8,13,21,34,55,89 |
| 9 | 12 | 56 |
| 10 | 12 | 85 |
| 11 | 12 | 868 |
| 12 | 12 | 225 |
| 13 | 12 | 56 |
| 14 | 10 | 0,1,2,3,5,8,13,21,34 |
| 15 | 11 | 53 |
| 16 | 5 | 0,1,2,3 |
| 17 | 12 | 177 |
| 18 | 12 | 56 |
| 19 | 10 | 69 |
| 20 | 12 | 56 |
| 21 | 13 | 0,1,2,3,5,8,13,21,34,55,89,144 |
| 22 | 12 | 85 |
| 23 | 11 | 143 |
| 24 | 12 | 56 |
| 25 | 7 | 127 |
| 26 | 8 | 0,1,2,3,5,8,13 |
| 27 | 12 | 225 |
| 28 | 12 | 225 |
| 29 | 12 | 80 |
| 30 | 11 | 0,1,2,3,5,8,13,21,34,55 |

**Multi-threaded Average Timings (7 runs):**

**Single-Threaded Results (7 runs):**