CIS 415 Operating Systems

Assignment 2 Report Collection

Submitted to:
Prof. Allen Malony

Author:
Taylor Madden

Introduction

In this project we were expected to implement 4 versions of a master control program (MCP). Each version builds upon its predecessor. In the end the goal of MCP was to run and schedule a workload of programs on the system. In part1, we focus on launching and running processes (ignoring the scheduling component in that we just run all the processes together). After all processes are up and running our MCP must wait for all processes to terminate and exit from main. Part 2 adds in some of the necessary logic for scheduling by stopping all forked processes before executing them by implementing a mechanism to signal processes to stop and to continue. Part 3 completes the scheduling functionality by implementing a scheduler that allows a process to run for a specified time quantum. Finally, part 4 will analyze the process information gained from executing part3 and present the information cleanly in the console each time a time quantum is completed.

Background

Firstly, the pseudocode provided for part 1 were the basic steps for launching processes and waiting for their termination which is, if it you reduce everything else down, the goal of part1. Therefore, I based my main function off of this pseudocode and worked on functionality to get the workload properly from stdin or command line (the read step in pseudocode), launch the workload using fork and execvp and wait for termination of all processes. I also got some background from reading the Process Management section of the Burroughs MCP wikipedia page (https://en.wikipedia.org/wiki/Burroughs_MCP), to see what that implementation may have looked like. The article says that jobs (or processes) are managed by moving in and out of job queues. This inspired me to implement my MCP with a queue struct to store processes (more about that in Implementation section). Finally, I know a lot about processes from studying the midterm, drawing from my previous courses, and interacting with them throughout projects. Knowing how processes work, change states, and are created helped everything click into place faster than it would have if no knowledge had been there beforehand.

Implementation

In the beginning, I just had a main function along with several helpers that performed actions like getting the length of an argument or command, reading a line, etc. I found was having a lot of trouble understanding at which points was my code failing and why. I had the same issue with the last project and I decided to do something about it so I wouldn't have the same headache. For that reason along with the above mentioned inspiration ala wikipedia, I decided to break things down and store my processes in a Queue structure (insp. From https://github.com/rafaeltardivo/C-Queue/blob/master/queue.c). I used a linked list implementation of a queue because I found that to be simpler in C. Also, I decided to make a structure for processes to make my solution more intuitive and easier for me to understand when implementing the following versions of MCP. Each process has a command, arguments, pid, status, number of arguments,as well as pointer to the next process in the queue. I also created separate functions for getting the workload (getProcesses), loading processes into memory/allocating space(allocateAssignProcesses), running all processes (forkAll), and deallocating memory (deallocateEndProcesses) which helped a lot with valgrind memory leaks and working out all the kinks in reading stdin/file which ended up being where my initial main had failed. To sum it up, MCP v1.0 calls main, initializes our process queue, gets the workload from a file from stdin, uses the workload to create

processes/load them into memory, puts the processes into our process queue, forks/executes all processes, waits for those to terminate, deallocates memory from processes and exits.

For part2, I implement signalProcesses, stopProcesses, continueProcesses, and waitProcesses to provide a mechanism to stop all forked processes and continue. startProcesses sends a signal to all children (which were returned by the call to forkAll()) to wake all children using kill(pid[i],USR1). After that I execute the processes with an execvp and call stopProcesses which sends SIGSTOP to suspend all children using kill again. After they are all stopped, I called continueProcess which again uses kill to sends SIGCONT to all child processes. After they are all continued we wait for all processes to terminate and exit (which I put into waitProcesses() which copies code that was previously used to wait in forkPrograms). After exiting, I deallocate my process queue and list of PIDs.

For part3, I replace all the signaling functions I created in part2 with the capability to handle signals given an alarm that goes off every specified timeslice. Essentially we initialize our queue, get our timeslice from stdin when executing output of gcc, set the signal handlers (I wrote additional static functions usr1Sig, alarmSig, and childSig to handle SIGUSR1, SIGALRM, and SIGCHLD signals in different ways), start our timer, and run processes in the specified timeslot. Of note in main is that I now start the first process in my process queue which is the process that runs the time quantum and free it after all the execution of the processes remaining in the queue.

For part4, I kept functionality from part3 the same but added in the function displayKnowledge which collects information that can be gained per timeslot interval. This part fell into place pretty quickly although I could have been more detailed in the metrics I displayed.

Console Output Example For ./p4 -ts=3 afile.txt

```
PID:                6665
COMMAND EXECUTING:  ./p4
BYTES READ:         0
BYTES WRITTEN:      0

PID:                6666
COMMAND EXECUTING:  ./p4
BYTES READ:         0
BYTES WRITTEN:      0

PID:                6667
COMMAND EXECUTING:  ./p4
BYTES READ:         0
BYTES WRITTEN:      0
execvp(program,args): No such file or directory

PID:                6661
COMMAND EXECUTING:  ./cpubound
BYTES READ:         1956
BYTES WRITTEN:      0

PID:                6662
COMMAND EXECUTING:  ./cpubound
BYTES READ:         1956
BYTES WRITTEN:      0

PID:                6663
COMMAND EXECUTING:  ./cpubound
BYTES READ:         1956
BYTES WRITTEN:      0
```

When testing part1 and par2 please still enter a timeslot to get intended output (time wont effect output for part1 or part2.)

Performance Results and Discussion

To the best of my knowledge, part 1, 2, 3, and 4 should be completed according to the specifications and should have no valgrind memory leaks. It is however being turned in on the third day after the original deadline so I expect a 30% penalty.

Conclusion

I spent at least 4 hours of almost everyday since I turned in project 1 working on this and still turned it in late! Oh well, I learned a lot, especially about breaking down an overwhelming program into various structures and functions! I am really really proud that I didn't give up or succumb to the pressure of the deadline. I think it's made me a much improved C programmer, so that's great. I also feel like it took away any semblance of work-life balance I had in my life, so take from that what you will! (Not to be misread--I'm sure there are plenty of students and people who work much harder than I do!) Overall, I'm glad that I could prove to myself that I can complete a project in C of this scale. I will have to get started tomorrow on project 3 if I want any hope of finishing in time! Hooray?