CIS 415 Operating Systems

Project 2 Report Collection

Submitted to:

Prof. Allen Malony

Author:

*Melanie Wiegand*

*mwiegan6*

*951976356*

**Report**

**Introduction**

*This project involved creating a Master Control Program step by step to organize and schedule processes. The first step (Part 1) was to implement a program which reads a set of commands from a specified input file and forks a separate child process to execute each command, then waits for each child process to terminate before exiting. Next, Part 2 provided functionality for allowing the MCP to pause all child processes before execution and later resume them using the SIGUSR1, SIGSTOP, and SIGCONT signals. Part 3 then implemented a round-robin system for scheduling the child processes, where each process is given a certain amount of time to run before the MCP forcibly pauses it and runs the next one. This was done using the alarm() system call, which signals SIGALRM after the specified period (quantum). In this way, each process gets a turn on the processor until all tasks are complete. For Part 4, a display table was added to showcase some relevant information about each process as they ran, such as the current state, time spent in kernel mode, and priority. Finally, Part 5 implemented a system for determining how long each process should spend before its turn ended; I chose to give 2 seconds to CPU bound processes and 1 second to I/O bound processes (as determined by the time spent in kernel mode compared with user mode.*

*The key aspects of this project were parsing the input file, creating and managing multiple child processes using the signal commands, implementing a round-robin style scheduling scheme, analyzing various relevant process statistics, categorizing processes as CPU bound or I/O bound, preventing memory leaks, and ensuring all errors were handled gracefully.*

**Background**

*This project really helped me to gain a deeper understanding of the concepts we have been discussing in class. It was really cool to see all of these ideas of process management and scheduling come to life in the form of the MCP. I was able to utilize the information from the lectures regarding process control, signaling, system calls, etc. to implement the project. I was previously fairly unfamiliar with how processes actually run and communicate with the OS, and actually implementing it myself really helped me to get a deeper understanding of the core concepts. I learned a lot about process handling and specifically the round-robin system and how it can be applied with selective quanta to maximize performance.*

*I used the string\_parser file from Project 1 to implement the file reading for this project. I essentially just copied my filemode code over from Project 1, which was nice because I could devote more time to the other aspects of the project instead of worrying about file reading. I did notice that for Project 2, we did not need to tokenize the inputs with the semicolon delimiter, since the input file had exactly one command per line. I kept the code in Part 1, but for Part 2 and onward, I decided it was cluttering up my code and removed it.*

*I continued using the ix-dev system (via Windows Powershell) to test my programs, utilizing a GitHub repository to communicate between my laptop files and my ix-dev account. I really like this method because it lets me work in VSCode, which I much prefer to coding directly in the terminal. I was able to quickly push files to the repo and pull them from the terminal, which gave me an easy way to test my code as I was going.*

**Implementation**

*I appreciated that each section of the project built off of the previous sections, so I could copy my code over instead of building each part from scratch. This made the implementation significantly more streamlined for me. I especially liked that the signals were introduced in parts 1 and 2, and then we had to figure out the logic of where to actually use them in the later sections.*

*By far the most interesting aspect of this project for me was in part 4 when I had to figure out how to get the relevant attributes of each process in order to display them in the table. I ended up printing the proc/<pid>/stat path into a file and then using fscanf to read through each of the attributes, setting the relevant ones to variable for future use. This also involved using dummy variables to skip through the attributes I wasn’t using, which may have been inefficient but seemed to work best for my method. There were a few tricky parts, like how I had to use “(%[^)])” to represent the comm attribute, since it is enclosed by parentheses and can contain spaces.*

*For Part 5, I realized that I needed to get the stats before I was ready to print them, since I needed to pass the user and kernel times into main earlier to accurately update my CPU bound/IO bound classifications. I ended up creating a struct, “pinfo” that stored each of the attributes, and used an array of these throughout my main function instead of my original PID array. I then used separate get\_stats() and print\_stats() functions.*

// putting all process info into a struct to pass out of the get\_stats() func

**typedef struct**

**{**

**pid\_t** pid;

**char** state;

**unsigned long** utime;

**unsigned long** ktime;

**long** priority;

**long** threads;

**unsigned long** stack\_addr;

**int** quantum;

**char**\* type;

**int** done**;**

**}** pinfo;

*This did cause a few issues that I had to think about in more depth; for example, I had to add a clause in my get\_stats() function to check whether the state was currently running or not, since this change caused some inconsistencies in my table. Additionally, a significant amount of my time spent on this project was dedicated to figuring out how to make the display table in parts 4 and 5 look really good. I made so many small adjustments and I kept having small spacing issues or bugs with a few of the columns. It took a very long time to fix everything while keeping my code functional.*

**Performance Results and Discussion**

*For each section of this project, I based the outputs off of those from the example executables provided by Abie in the class Slack page. I found these very helpful for judging whether my programs were running as expected, and I tried to match the syntax fairly closely. I believe my outputs for parts 1-4 are all very similar to the provided executables, as shown (my output first, example output second):*

***Part 1:***

*A black background with white text

AI-generated content may be incorrect.*

*A screen shot of a computer

AI-generated content may be incorrect.*

* *Processes are launched and immediately execute. Essentially the same as expected output.*

***Part 2:***

*A black screen with white text

AI-generated content may be incorrect.*

*A computer screen with white text

AI-generated content may be incorrect.*

* *SIGUSR1, SIGSTOP, and SIGCONT signals are sent before the processes execute.*

***Part 3:***

*A screenshot of a computer program

AI-generated content may be incorrect.*  *A screenshot of a computer program

AI-generated content may be incorrect.*

* *Only the last few lines are shown; the actual outputs are longer. The processes run one after another in order, each running for 1 second until the SIGALRM goes off and the MCP switches to the next process. After a process finishes, it is no longer part of the rotation (these images show only the last two processes alternating since the others have finished).*

***Part 4:***

*A screen shot of a computer

AI-generated content may be incorrect.A black screen with white text

AI-generated content may be incorrect.*

* *I chose all the same attributes to display since I was basing it off of the example output. So, the programs run essentially identically (I think my table looks a little better ;P), with the active program getting an “R” and the others having a “T” for the state column. The user and kernel times update with every cycle.*

***Part 5:***

*A screen shot of a computer

AI-generated content may be incorrect.*

*A black screen with white text

AI-generated content may be incorrect.*

* *Part 5 is where my output differs the most from the expected. I chose to classify each process as either I/O bound or CPU bound and give them different quantums accordingly. The example output seems to change the quantum based solely on the CPU usage. So, the tables look a little different.*

*I also made sure to run Valgrind on each part of my project to check for memory leaks and errors. For each section, I had no leaks and no errors, but did have some memory blocks classified as “still reachable.” I encountered this during the first project as well and was told it was not an issue. Valgrind logs are attached in the project submission.*

**Conclusion**

*This project helped me gain a much better understanding of how operating systems deal with multiple processes at the same time. I learned a lot about how to manage and schedule processes using the signal commands. I also learned more about what a process consists of and how they communicate with one another and with the OS. I found it super interesting to read through the proc/pid/stat manual page (*[*https://man7.org/linux/man-pages/man5/proc\_pid\_stat.5.html*](https://man7.org/linux/man-pages/man5/proc_pid_stat.5.html)*) and look at all the different attributes of the processes that are available. I also really enjoyed the process of building up a working operating system piece by piece, and I found the steps to be reasonably manageable. I thought it was really interesting how we could use a variable quantum to give some processes more time to run than others, and I am curious about what attributes I could have chosen to base this on that would have made the OS more efficient. This project deepened my understanding of how processes interact with the user and kernel spaces, and how it is possible to implement process scheduling using signaling and standard system calls.*