CIS 415 Operating Systems

Project <2> Report Collection

Submitted to:

Prof. Allen Malony

Author:

*<Andrew Chan>*

*<UO ID - 951908437>*

*<Duck ID - achan10>*

**Report**

**Introduction**

*In this project, we are making a MCP (Master Control Program) which is a simulation of a process scheduler. This is inspired by the “Ghost in the shell” idea by looking at each process as an independent entity that is managed by a central controller. The MCP takes in a file with the -f flag and reads each line as a command to then execute each of them as a separate subprocess. It then uses the round robin scheduling algorithm to allocate CPU time amongst them.*

*There were 4 parts plus an extra credit 5th part that all built off each other which created more complexity. At first the MCP is only responsible for creating and waiting processes, but then as the parts went on we added complexity with things such as time-slicing, resource monitoring, and even dynamic time slice adjustments based on the behavior of each process. This project looked at providing experience with important operating system concepts such as process management, signal handling, and scheduling algorithms. The following report will look at the challenges, development, and the testing that happened throughout each part of the project.*

**Background**

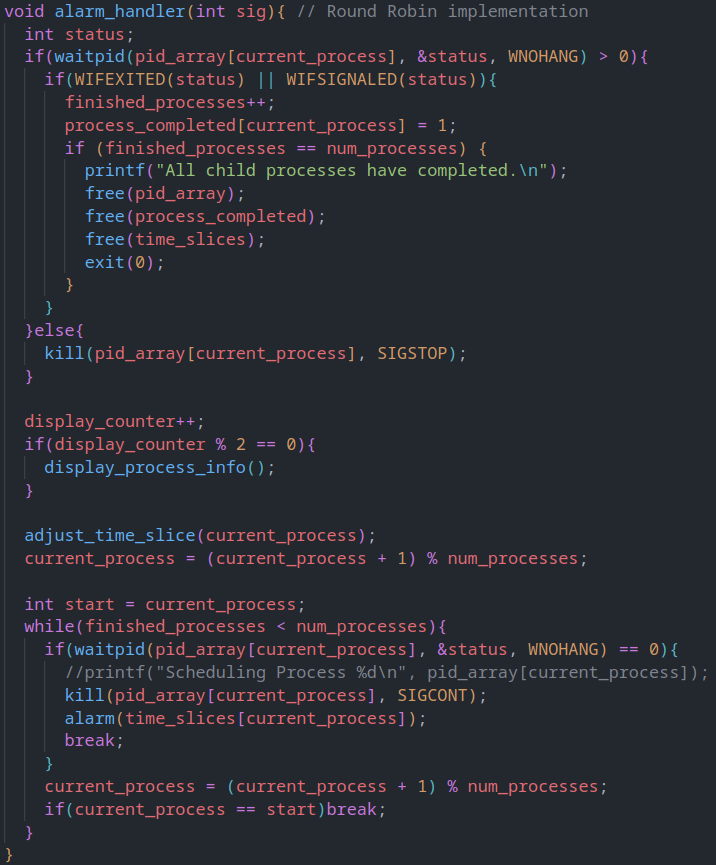
*This project uses the Round Robin scheduling algorithm to manage and handle the pool of subprocesses. This algorithm is a simple way that operating systems use to give CPU time to each process in a set interval otherwise known as time slices. When a process’s time slice is over, the OS suspends it and does a context switch over to the next process that is waiting in the queue. This provides fair and predictive outcomes since each process gets an equal opportunity to execute while not taking into consideration priority between the processes unless further logic is implemented.*

*When implementing the MCP, I used multiple Linux system calls such as fork(), exec(), and waitpid(). These system calls were meant to manage the processes such as fork() creating a new process by duplicating the parent. Whereas exec() replaced the parent’s memory with a new program. I also combined these calls with signals such as signwait(), SIGUSR1, and SIGSTOP to allow for very precise control over not only the timing, but also the execution of each of the subprocesses. This further refines the scheduling as I then added a feature in part 5 that adjusts the time slices dynamically by looking at the CPU-boundness and I/O-boundness so to speak for each of the processes. This method is better able to enhance the performance by allowing the CPU-bound processes more time to execute while reducing time slices for I/O-bound processes that usually spend more time in the waiting state. By implementing these methods I was further able to understand these calls and concepts, more specifically, the signals and process control structure. This was a very important part in being able to create a fully responsive scheduler in this project.*

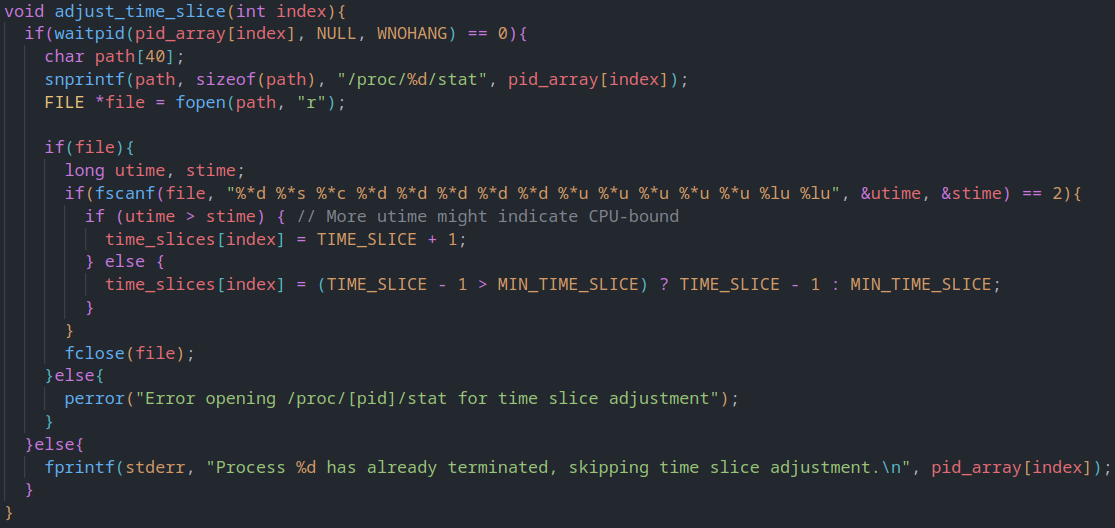
**Implementation**

*When it came to the implementation of this project, the MCP involved managing multiple subprocesses using the RR (round-robin) scheduling algorithm, along with the additional dynamic adjustments. Since this project is divided into 5 parts, I was able to show the process of development in each state.*

*In part 1, I implemented the basic functionality for creating subprocesses based on the commands that were given through the input file. I used methods like fork() and exec() to create separate processes for each command. Which then waits for each process using waitpid(). After all of this has finished it then exits. Part 2 added more control by introducing signals. This was achieved by assigning each child process a wait for a SIGUSR1 before calling exec(). This allows the MCP to synchronize the launch of all processes and is able to control them using signals such as SIGSTOP and SIGCONT to allow the starting and stopping of the processes. Part 3 was the implementation of the RR scheduling algorithm with a default fixed time slice of 1 seconds. The alarm() system call sets up a timer that activates the SIGALRM signal after a specific interval which then calls my self defined alarm\_handler() function. This alarm handler function suspends the current process with SIGSTOP, and then moves onto the next process in the list. Then it continues with a SIGCONT which then effectively cycles through all of the processes. This idea is context switching where we have a state stop, and a state load throughout all of the cycles. In part 4, I further enhanced the scheduler by reading the system resource data by using each process’s /proc data. I then display it using a table format which includes the user and system CPU time. It also showed things like the virtual memory, and the nice. This part required the parsing of the /proc/<pid>/stat file which gave a lot of really cool and interesting insight to the resources within the Linux process stats. And finally for the extra credit part 5, I implemented a feature that dynamically adjusts each of the process’s time slice depending on the CPU-boundedness and I/O-boundedness. I created the adjust\_time\_slice() function that reads the user and the system CPU times for each of the processes from the /proc file. Then if the user time is higher than the system time, the process identifies this as CPU-bound and its correlating time slice is increased to allow more uninterrupted execution since it doesn’t have to wait for things like I/O. And of course conversely, if the system time is higher than the time slice gets reduced. One thing I made sure to do was to make sure that the time didn’t go below 1 since that means that the process wouldn’t even get any CPU time which will cause starvation. Below are implementations that I created that I found particularly cool and are proud of.*

*The round robin scheduling:*

*The dynamic time slice adjustment:*

**

**Performance Results and Discussion**

*I believe my code runs well. After extensive testing I don’t believe that there are any memory leaks or errors in the code. That being said however, I believe that all code has mistakes or bugs to some extent so I wouldn’t be surprised if I wasn’t smart or extensive enough with my testing to catch any discrepancies. After analyzing that part 4 times and system resource data they all make sense, and since they all build off of each other, I believe that they are all right. Apart from part 5, there is 1 second given to each of the processes during the round robin implementation. And on top of this, my output also matches the example output that was given to us. Overall the MCP achieved the intended functionality and was able to adapt the scheduling time slices based on the subprocess behavior.*

**Conclusion**

*In conclusion I liked this project and learned a lot. I think it was really cool seeing this project progress throughout the multiple parts and it was very insightful to how a real CPU scheduler works. One of the most important things I think was the round robin implementation since it is an essential algorithm that CPU schedulers use. And on top of that part 5 was really good too since it gave a lot of insight on how the OS balances the CPU-bound tasks and I/O-bound tasks. I also liked parsing the /proc data since I didn’t even know that was a thing before this project and it was cool looking at all the different types of information.*

*Overall I believe that this project really strengthened my ability to develop not only in C, but also being able to piece through a very difficult task and take it part by part. It was very challenging overall but at the same time very rewarding. I also liked the idea of the given example executables for each part for us to compare our own code to.*