CS 415 Operating Systems
Fall 2025

Project #2 Report Collection

Author:

Rayna Patel

raynap

952017511

Report

Introduction

The goal of this project was to build a Master Control Program (MCP), a specialized system capable of managing the execution of multiple processes. This project focuses on concurrency, process control, and scheduling. The core functionalities I implemented included launching concurrent processes from an input file, controlling their execution states using POSIX signals, implementing a Round Robin scheduler using alarms, and monitoring system resources by working with the Linux virtual file system. Additionally, I implemented a "smart" scheduler for extra credit that will dynamically adjust time slices based on a process' behavior.

Background

This project required a deep understanding of the process lifecycle and asynchronous signaling in Linux. To launch processes, I utilized the fork() system call to create child processes and the execvp() system call to overlay them with new workload programs. A significant portion of the project relied on inter-process communication via signals. I used SIGUSR1 to synchronize process start times, SIGSTOP and SIGCONT to manage flow, and SIGALRM to trigger scheduling events.

For Part 3, I implemented a Round Robin scheduling algorithm. This required managing a queue of PIDs and handling asynchronous interrupts. For Part 4, I interfaced directly with the kernel data structures via the /proc directory. This involved parsing raw text files to extract real-time metrics such as CPU time (both user and system) and memory usage. Finally, for Part 5, I utilized these metrics to derive heuristics about process behavior (CPU bound vs I/O bound) to optimize CPU scheduling.

Implementation

Part 1 involved parsing the input.txt file and launching all commands concurrently. I used strtok to tokenize arguments and a loop to fork() each command immediately. The parent process then waited for all children to terminate using a loop of wait().

Part 2 introduced process control. In the child process, I implemented sigwait() (preceded by sigprocmask to block the signal) to pause execution immediately after forking. In the parent process, I implemented logic to send SIGUSR1 to wake all children simultaneously, followed by SIGSTOP and SIGCONT to demonstrate full control over process suspension and resumption.

Part 3 turned the MCP into a Round Robin scheduler. I implemented a custom signal handler for SIGALRM. This handler manages the context switch. It checks for terminated processes using waitpid with the WNOHANG flag, stops the currently running process using SIGSTOP, advances the index in the global PID array, and resumes the next process with SIGCONT (or SIGUSR1 if it has not been started before).

Part 4 added monitoring capabilities. Inside the scheduling loop, I added a function to read the /proc stat file for every active process. I parsed the specific columns for state, user time, system time, and virtual memory, formatting them into a table that continuously updates.

Part 5 enhanced the scheduler to be dynamic. I implemented a heuristic that compares a process's User Time (utime) against its System Time (stime). If utime is greater than stime, the process is classified as CPU-bound and granted a longer time slice (2 seconds). Otherwise, it is classified as I/O-bound and given a standard time

slice (1 second). I updated the dashboard to include two new columns, "Type" and "Slice," and modified the alarm() call to use these dynamic values.

Performance Results and Discussion

I encountered specific challenges due to the differences between macOS and Linux. While Parts 1 through 3 (process creation and signaling) largely functioned similarly on both systems, Part 4 failed on macOS because it lacks the /proc virtual file system found in Linux. Additionally, the helper programs (iobound and cpubound) compiled on macOS could not be executed when I moved the code to the Linux environment due to binary format differences. To resolve this, I transferred my source code to the Linux environment and recompiled all helper executables and the MCP using the Makefile. This step was crucial for the final validation of the Round Robin scheduler and the resource monitor.

Part 1 Output: The output below demonstrates the concurrent launching of processes. The output is "jumbled," indicating that processes were running simultaneously rather than sequentially.

Part 2 Output: The output below shows the signal control sequence. The "Starting," "Stopping," and "Continuing" messages confirm that the MCP successfully synchronized the child processes using SIGUSR1, SIGSTOP, and SIGCONT.

Part 1:

```
(base) raynapatel@dyn-10-108-80-151 project2 % ./part1 -f input.txt
Execvp: No such file or directory
test_script.sh                  Makefile
Project 2 - Description.pdf     iobound.c
part5.c                         iobound
part4.c                         input.txt
part3.c                         cpubound.c
part2.c                         cpubound
part1.dSYM                      ..
part1.c                         .
part1
Process: 89883 - Begining calculation.
Process: 89882 - Begining to write to file.
Process: 89883 - Finished.
Process: 89882 - Finished.
```

Part 2:

```
(base) raynapatel@dyn-10-108-80-151 project2 % ./part2 -f input.txt
Starting processes...
Processes running: Sleeping for 2 seconds...
Execvp: No such file or directory
Process: 90739 - Begining to write to file.
Process: 90740 - Begining calculation.
test_script.sh                  part1
Project 2 - Description.pdf     Makefile
part5.c                         iobound.c
part4.c                         iobound
part3.c                         input.txt
part2.dSYM                      cpubound.c
part2.c                         cpubound
part2                           ..
part1.dSYM                      .
part1.c
Stopping processes...
Processes stopped. Sleeping for 2 seconds...
Continuing processes...
Process: 90739 - Finished.
Process: 90740 - Finished.
```

Part 3 Output: The output below demonstrates the Round Robin scheduler. The interleaved output of the iobound and cpubound processes proves that the MCP is successfully time-slicing the CPU execution between processes.

Part 4 Output: The screenshot below shows the monitoring Dashboard running in the Linux environment. The table displays the correct PIDs, process states (Showing T for stopped and R for running), and resource usage statistics read from /proc.

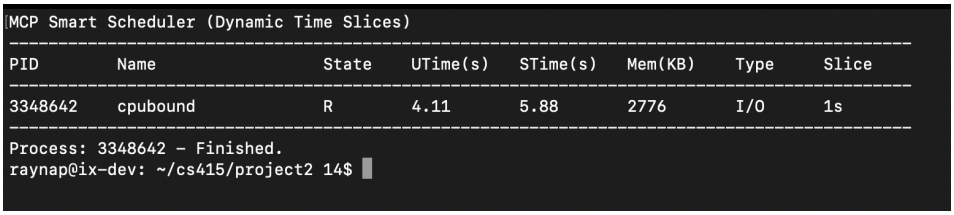
Part 3:

```
(base) raynapatel@dyn-10-108-122-198 project2 % ./part3 -f input.txt
test_script.sh                  part2.dSYM                      iobound.c
Project 2 - Description.pdf     part2.c                         iobound
part5.c                         part2                           input.txt
part4.c                         part1.dSYM                      cpubound.c
part3.dSYM                      part1.c                         cpubound
part3.c                         part1                           ..
part3                           Makefile                        .
Execvp: No such file or directory
Process: 34069 - Begining to write to file.
Process: 34070 - Begining calculation.
Process: 34069 - Finished.
Process: 34070 - Finished.
```

Part 4:

```
MCP Monitor (Round Robin Scheduler)
--------------------------------------------------------------------------------
PID        Name                 State      UTime(s)   STime(s)   Mem(KB)
--------------------------------------------------------------------------------
Process: 3195700 - Finished.
3195699    iobound              T          7.54       2.45       2776
3195700    cpubound             R          4.13       5.87       0
--------------------------------------------------------------------------------
Process: 3195699 - Finished.
```

Part 5 Output: The screenshot below demonstrates the "Smart Scheduler." The dashboard now includes Type and Slice columns. The scheduler correctly identified the cpubound process as having higher system time usage (due to frequent clock() calls) and classified it as I/O bound with a 1-second slice, dynamically adjusting the alarm() interval.



Memory Safety I utilized Valgrind to ensure memory safety. I verified that all file pointers were closed and dynamic memory allocated by getline was freed. The final Valgrind report showed zero memory leaks and zero errors (all logs are in the valgrind_log folder).

Conclusion

This project provided a full look into the responsibilities of an operating system kernel. Moving from basic process to creating a fully functional, monitoring scheduler highlighted the details of concurrency and asynchronous events and allowed me to understand the complexities behind these processes. I learned the importance of using sigprocmask to prevent race conditions where signals might arrive before a process is ready to handle them.

Furthermore, the issue with the /proc directory reinforced the differences between Unix-like systems (macOS vs. Linux) and the importance of testing in the target environment. Implementing the extra credit portion solidified my understanding of how operating systems can use runtime statistics to make scheduling decisions that optimize overall system throughput.