CIS 415 Operating Systems

Project 2 Report Collection

Submitted to:

Prof. Allen Malony

Author:

Vikram Thirumaran

UO ID - vikram

952046110

Report

Introduction

This project, developed for CIS 415 Operating Systems, focused on building an MCP (“Master Control Program”), a multi-part system that emulates key principles of a workload scheduler, including process creation, signal handling, time slicing, and system resource monitoring. Inspired by Burroughs' MCP and Masamune Shirow's "Ghost in the Shell," the goal was to orchestrate multiple commands in a simulated time-shared OS environment.

The MCP reads commands from an input file and executes them as subprocesses using fork() and execvp(). Each subsequent part adds complexity: coordinating execution with signals, implementing round-robin scheduling with SIGALRM, inspecting process metrics from /proc, and dynamically adapting time slices for CPU- and IO-bound processes.

This project deepened my understanding of Linux process management, inter-process signaling, and scheduling logic, and provided extensive hands-on experience in systems-level C programming.

Background

The MCP is built incrementally through five parts:

- **Part 1** launches each command in parallel using `fork() + execvp()`.
- **Part 2** introduces signal coordination (`SIGUSR1, SIGSTOP, SIGCONT`) to simulate process control.
- **Part 3** upgrades the system into a round-robin scheduler using `SIGALRM`.
- **Part 4** inspects system-level data in /proc/[pid]/stat to monitor CPU usage, memory, etc.
- **Part 5** implements adaptive scheduling by classifying processes based on I/O behavior using /proc/[pid]/io.

The project uses a modular structure: all string tokenization is handled by `string_parser.[c/h]`, and each MCP stage lives in its own source file (`part1.c` to `part5.c`), with a single Makefile managing compilation.

An `input.txt` file was used for testing. It contained commands like:

```
ls -a -r -s

sleep 1

invalid name

./iobound -seconds 10

./cpubound -seconds 10

cat input.txt
```

Implementation

Each partX.c program begins by reading input.txt, tokenizing each line into a command array, and storing child PIDs in an array. Memory management is consistently handled via `malloc()` and `free()`, with checks using valgrind.

Key system calls used include:

- `fork()`, `execvp()` for launching commands
- `wait()` / `waitpid()` for process cleanup
- `sigaction()` / `kill()` for signal-based coordination
- `alarm()` and `SIGALRM` for time slicing in scheduling
- /proc/[pid]/stat and /proc/[pid]/io for process introspection

Part 3 and 4 required implementing a circular PID queue with signal-safe transitions, while **Part 5** added conditional logic to assign different time slices depending on whether a process was IO- or CPU-bound, inferred by comparing rchar and wchar values.

Notable Challenges

The most difficult aspects involved signal timing and memory safety:

- Signal race conditions: Ensuring children paused at the correct time and did not miss a `SIGUSR1` before `execvp()` was tricky. Blocking signals early and using `sigwait()` resolved this.
- /proc parsing: Reading stats required precise indexing into space-delimited fields and managing dynamic memory for each line.
- Adaptive scheduling logic: Initially, I overfitted read/write heuristics, and had to simplify the classification into a simple comparison between read_bytes and write_bytes.

One of the more subtle bugs was caused by a missing `SIGSTOP` in Part 3, which led to the scheduler continuing to resume a process that had already exited. Adding exit checks before sending signals resolved the issue.

Performance Results and Discussion

- All five parts compile cleanly with make, and make clean removes all binaries as expected.
- Commands from input.txt execute correctly, including valid and invalid ones.
- Processes are correctly paused, resumed, and rotated as per round-robin logic.
- Valgrind reports zero memory leaks in all parts (logs captured in valgrind-part1.log -valgrind-part2.log).

Note: Output of commands like ls and cat is expected to be jumbled due to concurrent execution — which aligns with the intended behavior of MCP v1.0.

Part	Heap Usage	Errors	Leaks
1	52 allocs, 52 frees	0	0
2	43 allocs, 43 frees	0	0
3	49 allocs, 49 frees	0	0
4	55 allocs, 55 frees	0	0
5	60 allocs, 60 frees	0	0

Conclusion

This project was an engaging and layered challenge that pushed me to understand the Linux process model and signal handling at a deep level. It solidified my understanding of:

- Synchronous vs asynchronous control
- Signal masking and timing issues
- Scheduling policies and their implementation
- Dynamic memory management with persistent child states

In a real-world OS, much of this complexity is abstracted away — but this project gave me a visceral understanding of how scheduling and monitoring are actually implemented under the hood.

A potential extension could involve true priority scheduling or integrating pipes and redirection into the workload specification. I'm especially proud of completing the Part 5 extra credit, as it made me rethink how even basic OS heuristics can shape system responsiveness.