CIS 415 Operating Systems

Project 2 Report Collection

Submitted to:

Prof. Allen Malony

Author:

Itay Mevorach

itaym

951918487

Report

Introduction

In this project, I implemented a Master Control Program (MCP) which primarily is used to launch a series of sub-process that will execute commands in an input file. The Master Control Program reads a list of commands, from the input file, and runs the process that will execute the read command. The Master Control Program will attempt to run processes in a time-sliced manner to achieve programming concurrency. Additionally, the Master Control Program has the functionality of keeping track of each process, monitoring how many and which resources each process is using, in real time.

While the Master Control Program is an objectively primitive operating system in comparison to modern systems. It is the first operating system to manage multiple workloads simultaneously through multiple processors and features the first implementation of virtual memory. At the time, this was an incredible innovation, which allowed for an increase in processing power and eventually led to the development of modern processors, with many cores and threads.

Background

In developing this Master Control Program, I drew upon my knowledge of Unix-like systems and the intricacies of process management within them. The implementation heavily relies on fundamental system calls such as `fork`, `execvp`, and `waitpid`, which allow for the creation, execution, and monitoring of multiple child processes. To facilitate communication and synchronization between the parent and child processes, I utilized signal handling mechanisms, particularly SIGUSR1 and SIGALRM in part 4, to control the execution flow and handle process suspension and continuation. These choices were made to ensure efficient and synchronized process execution, enabling the program to manage multiple tasks concurrently and systematically.

During the development process, extensive research into the workings of the `/proc` directory proved to be particularly challenging. Leveraging this system-specific directory, I was able to extract crucial process-related data, including process IDs, memory utilization, and execution statuses. The data extraction mechanism provides insights into the behavior and resource consumption patterns of the child processes, facilitating efficient monitoring and management during the execution phase. Furthermore, I recognized the significance of dynamic memory allocation to accommodate a varying number of child processes, ensuring the program's scalability to diverse workload demands. I decided to create processes based on, essentially, how many lines were in the input, effectively creating a process for each command. Overall, the project's background research and the understanding of system calls, process management, and signal handling played a pivotal role in shaping the program's architecture and functionalities.

Implementation

This code snippet shows my implementation of the top_proccess_data function, which is a helper function responsible for accessing and reading specific process information from the /proc directory in a Linux system. The function constructs the path to the corresponding status file of a process by using the process ID. The function then opens the file in "read" mode and extracts relevant process details. The implementation is below.

```
void top_proccess_data(const pid_t id) {
    char buf[BUFSIZE];
    char line[256];

    sprintf(buf, "/proc/%d/status", id);

    FILE* file = fopen(buf, "r");

    if (file) {
        while (fgets(line, 256, file)) {
            if (strncmp(line, "Name:", 5) == 0)
                printf("%s", line);

            if (strncmp(line, "State:", 6) == 0)
                printf("%s", line);

            if (strncmp(line, "Pid:", 4) == 0)
                printf("%s", line);

            // Additional similar lines for other process information
```

The code includes a series of conditional checks using strncmp function from the C standard library to identify specific keywords, such as "Name," "State," "Pid," "VmPeak," and more process-related information from the /proc directory. When a line is found with a matching keyword, the function prints that line, effectively displaying relevant process information.

The snippet demonstrates file I/O operations in C, including file opening, reading, and conditionals based on specific line content. The top_process_data function utilizes C's file-handling capabilities to access and extract necessary information from the status file of a particular process. Thus, top_process_data enables real-time monitoring and display of crucial process details during program execution.

While it is true that rest of the implementation of this project is the main driver of the program. I believe that this was the most interest learning experience, since the /proc directory was a completely foreign concept to me before researching for the project. And therefore, I decided to discuss top_process_data in this section.

Performance Results and Discussion

In terms of performance, my implementation has demonstrated efficient handling and management of multiple child processes, showcasing its capability to execute various tasks concurrently. The implementation leverages system calls and process management functionalities in a way that ensures smooth and effective multitasking. By utilizing features such as forking, signaling, and process control, I've enabled the system to execute tasks in parallel, thereby maximizing resource utilization and minimizing idle time.

One of the key performance metrics I observed was the effective utilization of CPU resources. Using the SIGSTOP and SIGCONT signals, I control the execution of child processes, allowing them to pause and resume as necessary. This strategy proved crucial in preventing unnecessary CPU usage, ensuring that processes were only active when needed and idle when waiting for further instructions.

Finally, my project exhibited reliable process monitoring capabilities, allowing for the real-time tracking of critical process information. By accessing and parsing data from the /proc directory, I could display essential process details such as name, state, ID, and memory usage. This feature enhanced the project's performance by providing insightful information about the ongoing processes, facilitating effective resource management and troubleshooting.

Conclusion

Overall, the project's performance was robust, as evidenced by the successful execution of multiple processes with synchronized control and real-time monitoring. The implementation of efficient process handling techniques, coupled with insightful data retrieval from the system, contributed to the project's strong performance, enabling it to effectively manage and execute concurrent tasks while ensuring optimal resource utilization. In conclusion, my implementation of this project allows for concurrency in running processes using key system calls such as 'fork' and 'execvp' which enable child processes. And the usage of the /proc directory allows my implementation to send feedback to the user on utilization of CPU resources.