CIS 415 Operating Systems

Project 2 Report Collection

Submitted to:

Prof. Allen Malony

Author:

Josh Jilot

UO ID - jjilot

951863225

Report

Introduction

In this project, I was tasked with iterating upon an MCP – Master Control Program – which controls a number of sub processes given through an input file. The first iteration only begins the processes and then sees them to completion, but by the final part the MCP has implemented round robin scheduling to give each process two seconds of execution until they all finish. This project was much simpler than the previous one thanks to the iterative design, but it still required a solid understanding of process scheduling and forking. I found that I was struggling to make headway at first, but after studying up for the midterm I was more familiar with the concepts required to make progress and I was able to come up with clever solutions much quicker.

Background

The most important parts of this project to understand were the basics of how processes function after being forked, overwritten (with execvp()), or sent a signal. Before this project, I had never worked with processes in such a direct way and I had certainly never used functions like fork(), waitpid(), or sigwait(). Learning about processes and their functions was, itself, a very large part of completing this assignment, and it was greatly aided by my studying for the midterm. After I had established a general understanding of how processes are created and scheduled I was much more easily able to conceptualize how I wanted my programs to run. This general understanding included things like how child processes can be detected by checking the return value of fork(), what happens when a process is blocked, and how signal handling functions like sigset(), signal(), and alarm() work.

After gaining a basic understanding of the essentials for this project, it was time to learn some specifics. I learned very quickly that iterating through waitpid() calls on each child process would allow me to easily wait for every process to finish before termination. However, I discovered almost as quickly that using a technique like this would block my parent process in the meantime – making it useless for scheduling. This forced me to dig deeper and learn about certain flags I could implement to disable the blocking of functions like this. I continued to learn new things (of which I am being vague because I will be very specific in the Implementation section) in each iteration, like how the signal() function can be used to detect a signal and call an appropriate function, which is extremely useful for handling child processes and their various activities. The last major aspect of the project I had to learn about was the /proc directory and how it was formatted. Fortunately, this was as simple as looking up the list of stats available in the proc/stat file and only accessing the ones I needed. Once I had brushed up on all of these basics, I was ready to use them in tandem with my general knowledge of process scheduling to implement each part of the project.

Implementation

For each part of this project I found myself learning about new functions and techniques I had never used before. In part 1, I used the file line tokenizing functions from Project 1. By including string_parser.c in my main file I was able to iterate through the lines of the file with a while loop and tokenize the results. In Project 1, I immediately sent these tokens to a new function that handled calling the associated unix function, but this time I chose to copy them and store them in an array, giving me an array of tokenized lines in the file. Then, it was simple to iterate through the array and begin each process, waiting until they were all complete to terminate the parent process. For part 2, I created a signaling function that would send a desired signal to each process. This allowed me to send the three signals to each process as desired and terminate the main process.

Part 3 was when the scheduling began, which added a whole new layer to the project. To implement round robin scheduling, I first needed to make a parallel array to my process array that would track the status of each process. I iterated through this with a while loop so that my program would know when to terminate without checking waitpid() (and in turn blocking itself). This is demonstrated below, along with a commented out line utilizing the sleep() function to avoid unwanted output interleaving that I removed due to a ban on sleep() usage in the rubric. The processes would be marked as terminated by my scheduling functions, which simply iterate

```
// continue scheduling and waiting for child processes (parent only)
while (1) {
    int all_terminated = 1;
    for (int i = 0; i < size; i++) {
        if (process_alive[i]) {
            all_terminated = 0;
            break;
        }
    }
    if (all_terminated) { break; }
    // sleep(1)   // this line prevents breaking up ls output but rubric says no sleep()
}
```

through the list of child processes and send an alarm signal every two seconds. Each time this signal was sent, the next active process in the array was given the SIGCONT signal while the currently running process was given the SIGSTOP signal.

Additionally, to handle a child terminating, I added a function that would be called whenever the SIGCHLD signal was received. This meant that all I needed to do was check the list of processes for a newly terminated child and mark it as terminated. The instructions mentioned that this could not be done with waitpid(), but I found that the WNOHANG flag would allow me to check the status of the child process without blocking. However, just to be safe, I implemented it using waitid() and the same flag as shown below. This left only part

```
void handle_sigchld(int sig) {
    siginfo_t siginfo;

    // WNOHANG to avoid blocking, WEXITED to check for terminated children only
    // no waitpid()
    while (waitid(P_ALL, 0, &siginfo, WNOHANG | WEXITED) == 0) {
        // break loop if none have exited
        if (siginfo.si_pid == 0) break;

        // process the terminated child by PID
        for (int i = 0; i < size; i++) {
            if (pid_arr[i] == siginfo.si_pid) {
                process_alive[i] = 0;
                break;
            }
        }
    }
}
```

four, where I created a function to loop through all processes currently running and print their proc/stat info. I chose to print the same info as the example executable: user time, system time, total time, PID, nice value, and virtual memory size. This was fairly simple, as I just used fscanf() to ignore the fields I didn't need and print the ones I did.

Performance Results and Discussion

All output screenshots are included in the “captures” folder within my submitted .zip file. Each part worked as required with no memory leaks. I formatted my outputs to closely match the example executables as well, for easier readability. The memory checks can also be found in the “valgrind_logs” folder within my submitted .zip file. While I did not have any major issues, I did notice that the example executable displays an execvp() error along with a free() error, while mine only displays the execvp() error. I also found that the output of the ls command would sometimes get split up by one of the subsequent processes starting. I solved this with a sleep(1) call, but removed the solution due to the ban on sleep() for scheduling purposes. I do not believe these issues are major enough to merit point deductions and just show slight differences in implementation.

Conclusion

Overall, I actually found myself enjoying this project. It allowed me to tie in the information I have learned in class to an actual assignment, solidifying my knowledge of the concepts greatly. The iterative approach to the project meant that I felt a sense of accomplishment after each section and was able to refine previous parts as I tested new techniques later on. I felt that this project was much more straightforward and relative to the overall class compared to the first one, and I look forward to similar projects in the future.