CIS 415 Operating Systems

Assignment 2 Report Collection

Submitted to:

Prof. Allen Malony

Author:

Ryan Gurnick

Report

Introduction

In this project we are developing a master control program whose sole purpose is to take the input file and schedule the execution of the given binaries. This project has multiple parts in which we are adding in additional functionality. In part 1 we mainly just work on writing the code that will launch the master control program, load data in, and then execute all processes in order on their own forked processes. Part2 is attempting to start adding some scheduling mechanics so that each process is told when to operate by the parent using various new signals that are implemented to stop and continue each processes execution. Part3 we start adding in some basic scheduling to allow processes to switch into and out of an execution state. Part4 we add in some addition information using the /proc statistics information.

The goal is to take a file as input and use each line as a separate process for execution. Throughout this project we will add more and more functionality to this master control program which will allow it to take more and more control over which processes will run and when. The instructions where a bit vague with certain parts of this project which lead to some ambiguity in my implementation and possible lack of functionality.

Background

Throughout this project we have been using a variety of system calls that allow this MCP to be possible. The main one in fact is the fork() sys call that allows the running process to split and end up with a child. Additionally we have reused some previous system calls which are beloved like getline(), strtok(), fopen(), execvp(), fclose(), free(), waitpid(). We used getline() to parse the input from the input file after using fopen() to get the contents. From there we use strtok() to break up the line by spaces for the execvp() system call to execute based off of. Then we fclose() the input file and waitpid() for the processes to finish before free()-ing and fclose()-ing the files. (This is the basic structure that this program takes).

In part one you can really see that basis start to take shape, from there part two will add on some sigaction() calls along with a varity of others to send signals, such as kill(). In here we will use the sigaction structure to provide a signal to various processes. This allows for a SIGUSR1 signal to send before starting the process, a SIGSTOP to stop the process, and a SIGCONT to continue the process. In part three we use alarm() to try and tell the running process to only run for a certain amount of time in this case 1 second. (Note: I don't think that I implemented this entirely correctly as I read that the sig_handler should not do more than a constant operation) In part 4, I use part 3 to build off of and use more of the same system calls to open /proc/<pid>/ status and provide the first ten lines to show how the processes status is at the end.

Implementation

Let's get into the implementation of this application. I will talk in generalities about as much of this as I understand. (Note: the code also has "okay-ish" comments in it to help navigate it, in places where my understanding is lacking you may see the same for the comments)

Figure 1: Some Algorithm implementation

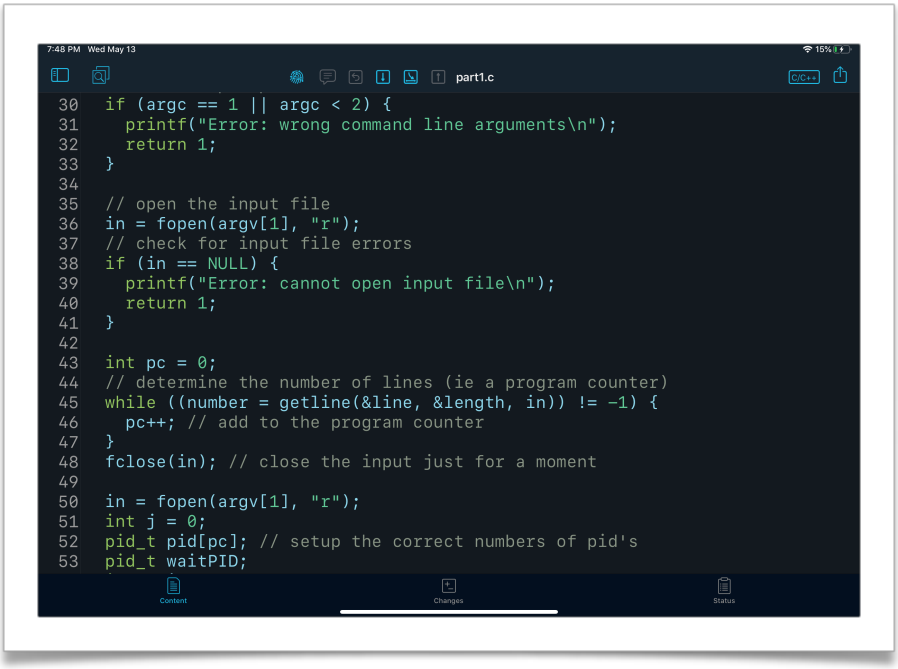

Figure 1: *The code on line 35-48 is being shown to display the implementation for getting the input file. This will load the file given via a command line argument. If there is an issue opening the file, the user will see a generic error. Next we will keep track of the number of lines which will form the basis for the process counter, each line is going to be a single process. On line 43 we setup the variable pc which will act as an accumulator for the while loop below with the intent of storing the number of lines/the process count (which are the same). On line 48 we close the file to easily move the line pointer back to the beginning of the file so that we don't need to store it, in my implementation the input file is just loaded twice rather than storing it entirely into memory somewhere.*

Figure 2: *This code is to show the way that processes are started. This is pretty similar to the sample code that was provided aside from the while loop that goes through each line of the input file. For each line we will zero out the arguments array for use in the future. Then we will tokenize the line by spaces and store it into the arguments array. Additionally we strip out the new line character so that we don't have any issues. From there fork() the main process and if its the child then we will run an execvp() to run the command provided from the input. Following that we will free memory if there are errors and then exit(). Finally after all process have been forked we will wait for all of the processes to complete.*

```
16 void handler(int signal, siginfo_t *info, void *context) {
17     printf("Child Process: %i - Received Signal: %d\n", getpid(), signal);
18 }
19
20 void signaler(pid_t *pid, int signal) {
21   int i = 0;
22   for (i; i < ProgramCount; i++) {
23     printf("Received Signal %d to pid %d\n", signal, pid[i]);
24         kill(pid[i], signal);
25   }
26 }
```

Figure 3: *Here is a basic version of the handlers used in part 2 of the project to handle the receiving of signals as well as a signaler function that will go through all programs and send the signal using the kill() system call. This same basic structure continues to be used for the signaler all the way to part 4.*

In part 3-4 the handler gets some extra functionality (which I don't know that it should, but its the only place that seems logical to me) to handle the processing of the stop signal and continue signal. This will allow the children to stop and continue based on received signals.

Implementation Note: *I would normally add more for this implementation section but this document is already getting rather long and I don't have a great understanding of part 3 and 4 so the code should speak for itself. I have used a hodgepodge of information from the book, slides, and the internet without much success in understanding the content to the fullest extent.*

Performance Results and Discussion

There were not formalized results that where provided for this assignment. Overall when running valgrind to check for memory issues there should not be any for the parent or children processes. There are some minor memory issues that have not been tracked down at this point dealing with uninitialized variables in child processes. When running valgrind there was some information about which files where causing this issue however it did not provide information as to which lines where causing this issue. GDB was run to try and determine what was causing this but this did not become clear.

Since in the instructions it is not clear what type of output is desired, I tried to follow a general rule of thumb ensuring that you can see the signals as printed in the code. For Part 1, & Part 2 you can clearly see the executed processes output taking place in between the signals. In Part 3 & Part 4 you will not see the output

itself for some reason. In Part 4 there will be a part of the MCP that runs that will clear the screen using the system("clear") method and then it will show the processes status. Mainly this just shows a couple lines of the most useful information at the top, but it makes screenshooting the output difficult.

Conclusion

The instructions for this assignment need to be changed a huge amount. I missed a single lab (lab 6, which for the record is my fault) which may have provided some context on this entire assignment. However throughout the three weeks i've been working on this, we have not discussed nearly enough content regarding the successful completion of this assignment. For lab 5, which supposedly covered this we where expected to fork a process, which helped in general but that code was provided in the instructions. Overall, I feel like I was not able to successfully complete this because the reading, lectures and lab did not provided adequate information about the topics required to fully complete this assignment.