CIS 415 Operating Systems

Assignment 2 Report Collection

Submitted to:

Prof. Allen Malony

Author:

Noah Tigner

Report

Introduction

This project was an implementation of an MCP (Master Control Program). The MCP is in charge of scheduling processes to run concurrently. The MCP reads in commands from a text file and executes them. Round Robin scheduling is done to determine which process executes next (for a set slice of time). The MCP is also capable of listing information about each workload process that is executing or waiting.

Background

This program made extensive use of signals and signal handlers. SIGCONT and SIGSTOP were used to resume and stop processes. SIGUSR1 was implemented to allow a waiting process to continue before executing (ie before entering the critical section). SIGALRM and alarm were used to make the scheduler stop the running workload process and resume another process (in a Round Robin fashion).

For getting info about processes, the proc filesystem is read. This mimics the Unix top command. Specifically, for a given process, info is read from /proc/<pid>/stat. Only relevant info is printed to the screen.

Implementation

Part 1 of this project implements the core functionality of forking and executing commands given by lines in a text file. Part 2 adds in the use of signals and signal handlers, which are used to start, resume, and pause workload processes. Part 3 introduced the round Robin scheduling algorithm. Scheduling events are triggered by the alert() function, which is called at a fixed quantum. Part 4 includes the functionality of printing process info. This is done at a fixed interval for each workload process that has yet to terminate.

Performance Results and Discussion

As far as I can tell, the project performs all of the required functionality. Print statements are used to let the user know what is going on behind the scenes (ie a process received a signal, etc). Using valgrind, there are no leaks possible, and no errors occur. The only thing I have noticed is that in part 4, when running the cpubound and iobound tasks, when the process info is shown, both are sometimes listed as running, even though the round robin algorithm ensures only one of the two is actually running at a time. I think this is because the process switching is done very quickly, and the printing of process info is relatively slow (it relies on reading from proc and printing which is relatively slow), and the printing is not always up to date. Another interesting thing is that (at least on my VM), proc lists the kernel and user jiffies of some processes as 0, which doesn't seem correct but is reflective of the data held in proc.

Conclusion

This project introduced me to fork() and exec(), signals and signal handlers, process scheduling, and reading and analyzing info from proc. I learned a lot about concurrency and scheduling.

Code

```
# define _GNU_SOURCE
# include <stdio.h>
# include <stdlib.h>
# include <string.h>
# include <unistd.h>
# include <dirent.h>
# include <stdbool.h>
# include <errno.h>
# include <sys/types.h>
# include <sys/wait.h>
# include "header.h"

// Global
pid_t pidPool[10];
int running = 0;

// For a given process, read and print useful info from proc
void processInfo(int pid) {
    char path[64];
    snprintf(path, sizeof(path), "/proc/%u/stat", pid);

    FILE *procFile = fopen(path, "r");

    size_t size = 0;
    char *line = 0;

    do {
        ssize_t lineSize = getline(&line, &size, procFile);
        if (lineSize < 0){
            break;
        }

        char *split = strtok(line, " ");
        int j = 0;
        while(split != NULL) {
            if(j == 0) {
                printf("Process %s\n", split);
            }
            else if(j == 1) {
                printf("   -Name:                  %s\n", split);
            }
            else if(j == 2) {
                printf("   -State:                 %s\n", split);
            }
            else if(j == 5) {
                printf("   -Parent PID:            %s\n", split);
            }
            else if(j == 10) {
```

```
                printf("   -Flags:                  %s\n", split);
            }
            else if(j == 15) {
                printf("   -User Mode Jiffies:      %s\n", split);
            }
            else if(j == 16) {
                printf("   -Kernel Mode Jiffies:    %s\n", split);
            }
            else if(j == 20) {
                printf("   -Priority:               %s\n", split);
            }
            else if(j == 23) {
                printf("   -Start Time:             %s\n", split);
            }
            else if(j == 24) {
                printf("   -Virtual Memory Size:    %s\n", split);
            }

            split = strtok(NULL, " ");
            j++;
        }
    }
    while(!feof(procFile));
    fclose(procFile);
    free(line);

}

// Works like Round Robin
void alarmHandler(int num) {
    int count = 0;
    while(pidPool[count] != -999) {
        count++;
    }

    int index = 0;
    while(index < count) {
        kill(pidPool[index], SIGSTOP);
        index++;
    }

    int stat;
    int i = 1;
    while( i < count+1) {
        running = (running + 1) % count;

        // Don't try to continue a terminated process
        if(waitpid(pidPool[running], &stat, WNOHANG) == 0 && pidPool[running] != 0) {
            printf("Sending %d SIGCONT\n", pidPool[running]);

            kill(pidPool[running], SIGCONT);
```

```
            break;
        }
        i++;
    }
}

void parseFile(char *buffer, size_t bufferSize, char *in) {

    char whitespace[2] = " ";

    FILE *infile = fopen(in, "r");
    if(infile == NULL) {
        write(1, "Input file missing\n", 20);
        free(buffer);
        exit(1);
    }

    size_t input;

    for(int i = 0; i < 10; i++) {
        pidPool[i] = -999;
    }

    size_t numLines = 10;
    char **argsList[10];
    int lineNum = 0;

    struct sigaction al = {0};
    al.sa_handler = &alarmHandler;
    sigaction(SIGALRM, &al, NULL);

    while(((input = getline(&buffer, &bufferSize, infile)) != -1) && lineNum < numLines) {

        int argNum = 1;

        for(int i = 0; i < strlen(buffer); i++) {
            if(buffer[i] ==  ' ') {
                argNum++;
            }
        }

        char *args[argNum+1];
        char *arg = strtok(buffer, whitespace);

        int i = 0;
        while(arg != NULL && i < argNum) {
            if(arg[strlen(arg)-1] == '\n') {
                arg[strlen(arg)-1] = '\0';
            }
            args[i] = arg;  // strdup or stdcpy??
```

```
        arg = strtok(NULL, whitespace);
        i++;
    }
    args[argNum] = NULL;

    pidPool[lineNum] = fork();

    if(pidPool[lineNum] == 0) {
        // Child

        printf("Process: %ld - Suspended\n", (long)getpid());


        int success = execvp(args[0], args);
        if(success == -1) {
            perror("Error");
        }
        exit(-1);

    }
    else if(pidPool[lineNum] < 0) {
        printf("Fail\n");
    }

    lineNum++;
}

fclose(infile);
free(buffer);

sleep(2);

int counterToTop = 0;
bool stillGoing = true;
while(stillGoing == true) {

    // checks if theres still processes
    stillGoing = false;
    int j = 0;
    while(pidPool[j] != -999) {
        int response = kill(pidPool[j], 0);
        if(response != -1) {
            stillGoing = true;
            break;
        }
        j++;
    }

    // periodically print out process info
    if(counterToTop%6 == 0 && stillGoing == true) {
        printf("*****************************************************************\n");
```

```
            j = 0;
            while(pidPool[j] != -999) {
                int response = kill(pidPool[j], 0);
                int stat;
                if(response != -1 && waitpid(pidPool[j], &stat, WNOHANG) == 0) {
                    processInfo(pidPool[j]);
                }
                j++;
            }
            printf("*********************************************************************\n");
        }
        counterToTop++;


        int quantum = 5;

        // Trigger the scheduler
        alarm(quantum);
        sleep(quantum);
    }

    printf("Process: %ld - Joined\n", (long)getpid());
    int stat = 0;
    wait(&stat);
    if (WIFEXITED(stat)) {
        printf("Exit status: %d\n", WEXITSTATUS(stat));
    }
}


int main(int argc, char *argv[]) {

    FILE *infile;

    // Allocate memory for the buffer
    char *buffer;
    size_t bufferSize = 32;
    size_t input;
    buffer = (char*)malloc(bufferSize * sizeof(char));

    if(argc == 2) {

        parseFile(buffer, bufferSize, argv[1]);

        return 0;
    }
    else {
        write(1, "Argument for input file missing\n", 33);
        free(buffer);
        exit(1);
```

```
    }
    return 0;
}
```