CIS 415 Operating Systems

Assignment <2> Report Collection

Submitted to:

Prof. Allen Malony

Author:

Irfan Filipovic

Report

Introduction

Through my journey as a Computer and Information Science undergraduate at the University of Oregon, I have used a command line interface on multiple operating systems in order to execute commands. To execute multiple commands and operations in a sequence a process may be made for each. These processes then must be managed to run and terminate in an efficient manner. The main process, for example created when you open initial command line interface, will manage the scheduling of these processes.

This project aims at implementing "MCP Ghost in the Shell" which is the main process that will schedule these processes. The sequence of commands to the MCP will be given in an input file, commands such as Linux system call 'ls', will be present as well as programs provided by Professor Malony. The project consists of four parts, each part is a more complex management of processes. The first part will consist of creating processes and executing, while the last will require scheduling, context switching, and providing data on running processes.

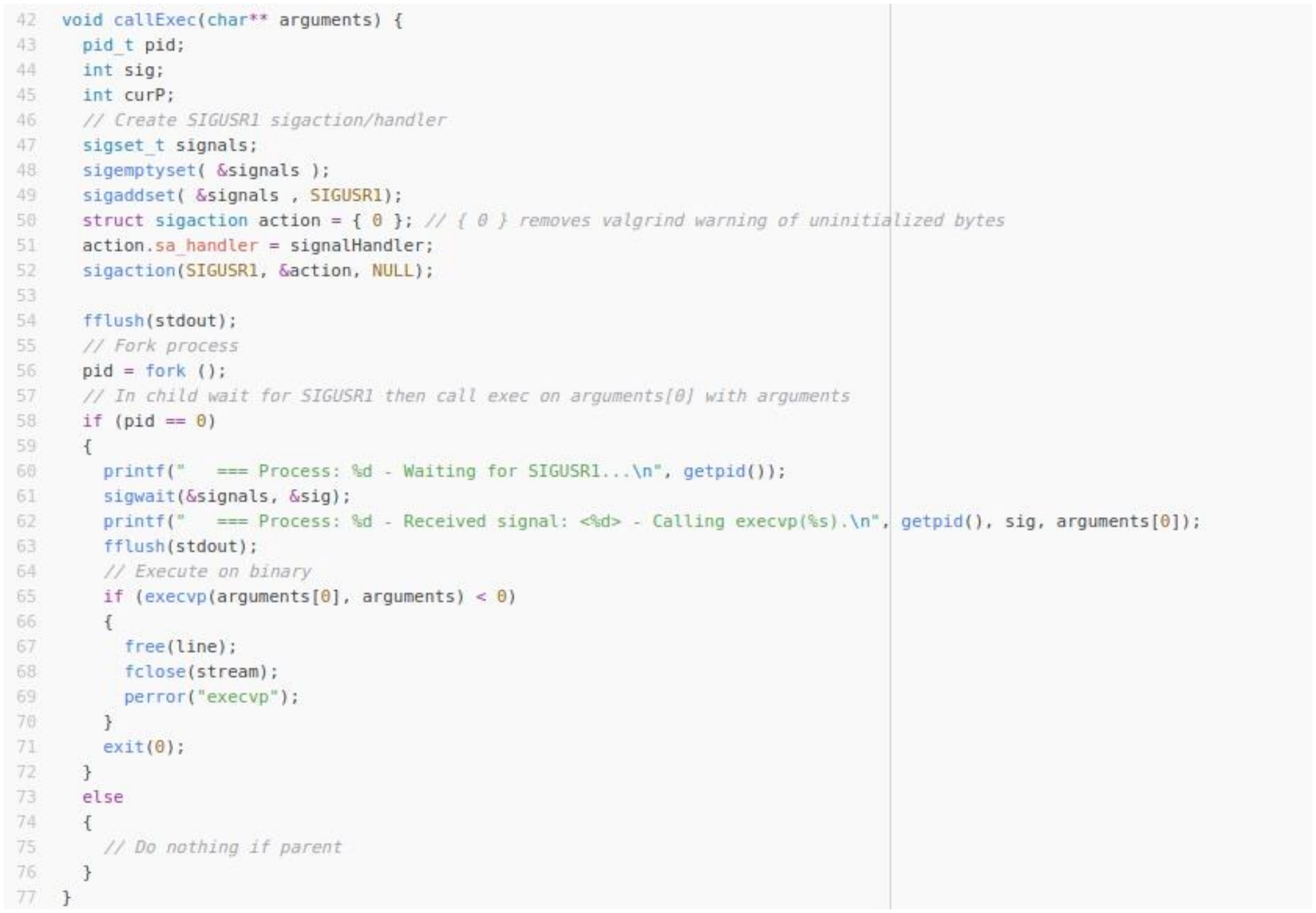
Background

Linux system call decision making was conducted by searching section 2 of the Linux man pages. To implement input file parsing code was repurposed from project one, as the file input mode was consistent with parsing data based on delimiters. Creating and waiting for processes was displayed in lab 4, sending signals and creating handlers was displayed in lab 5, and in lab 6 implementing scheduling and alternating between processes was conducted.

The only functionality which was difficult to implement was gaining access to the '/proc' files. To implement all process management, I reused code from labs and followed guidance provided in the documentation. The Linux manual page provided the understanding needed to implement parsing strings and integers together, and the Linux man pages at die.net (https://linux.die.net/man/2/) provided some help as well.

Implementation

Each part of the project incorporated the initial process creation and wait in part 1, while parts 2 through 3 added more functions. To run processes concurrently processes must be stored in array, and then called therefore after part 1 there was a function to accept a list of arguments and instantiate a process that would wait until 'SIGUSR1' was sent and then operate on command in argument[0] and the arguments for the call following.

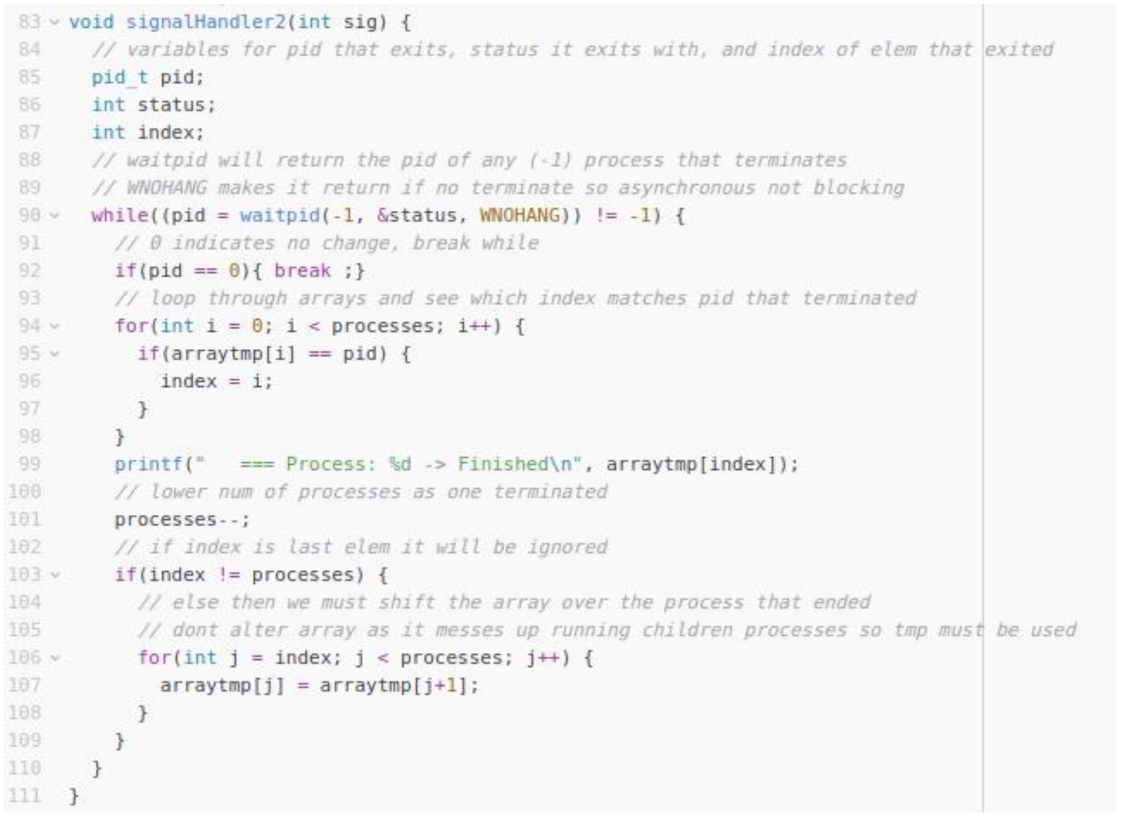


```c
void callExec(char** arguments) {
  pid_t pid;
  int sig;
  int curP;
  // Create SIGUSR1 sigaction/handler
  sigset_t signals;
  sigemptyset( &signals );
  sigaddset( &signals , SIGUSR1);
  struct sigaction action = { 0 }; // { 0 } removes valgrind warning of uninitialized bytes
  action.sa_handler = signalHandler;
  sigaction(SIGUSR1, &action, NULL);

  fflush(stdout);
  // Fork process
  pid = fork ();
  // In child wait for SIGUSR1 then call exec on arguments[0] with arguments
  if (pid == 0)
  {
    printf("   === Process: %d - Waiting for SIGUSR1...\n", getpid());
    sigwait(&signals, &sig);
    printf("   === Process: %d - Received signal: <%d> - Calling execvp(%s).\n", getpid(), sig, arguments[0]);
    fflush(stdout);
    // Execute on binary
    if (execvp(arguments[0], arguments) < 0)
    {
      free(line);
      fclose(stream);
      perror("execvp");
    }
    exit(0);
  }
  else
  {
    // Do nothing if parent
  }
}
```

Figure <1>: callExec() from ‘part2.c’

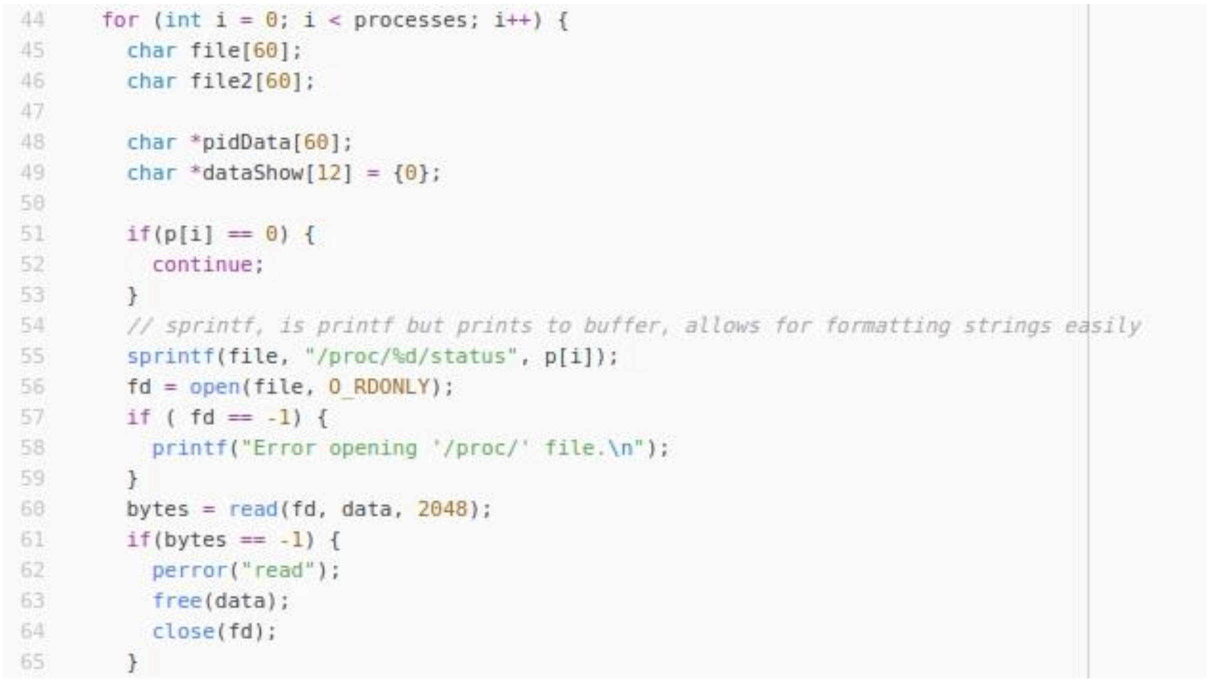
To schedule the processes that are created in callExec() it was important to account for children processes terminating and no longer queuing or continuing them. This required creating a signal handler which only processed ‘SIGCHLD’, the signal sent by a child process when terminated. To remove the process from further scheduling it was shifted from the list of arrays sent to the scheduler and the number of processes was reduced.



```c
void signalHandler2(int sig) {
  // variables for pid that exits, status it exits with, and index of elem that exited
  pid_t pid;
  int status;
  int index;
  // waitpid will return the pid of any (-1) process that terminates
  // WNOHANG makes it return if no terminate so asynchronous not blocking
  while((pid = waitpid(-1, &status, WNOHANG)) != -1) {
    // 0 indicates no change, break while
    if(pid == 0){ break ;}
    // loop through arrays and see which index matches pid that terminated
    for(int i = 0; i < processes; i++) {
      if(arraytmp[i] == pid) {
        index = i;
      }
    }
    printf("   === Process: %d -> Finished\n", arraytmp[index]);
    // lower num of processes as one terminated
    processes--;
    // if index is last elem it will be ignored
    if(index != processes) {
      // else then we must shift the array over the process that ended
      // dont alter array as it messes up running children processes so tmp must be used
      for(int j = index; j < processes; j++) {
        arraytmp[j] = arraytmp[j+1];
      }
    }
  }
}
```

Figure <2>: signalHandler2() from ‘part3.c’

The most problematic code snippet provided to be creating the file name for accessing ‘/proc/’ files. These files contain process information, stored under ‘/proc/<process id>/’, which must be accessed for part 4. To parse a string using characters and integers it was necessary to consult the manpages. A function such as ‘printf’ would have worked great, but that prints to ‘stdout’. Therefore, I settled upon sprintf which will print a specified number of bytes and the string into a buffer. This buffer was then used as the file name.



```c
  for (int i = 0; i < processes; i++) {
    char file[60];
    char file2[60];

    char *pidData[60];
    char *dataShow[12] = {0};

    if(p[i] == 0) {
      continue;
    }
    // sprintf, is printf but prints to buffer, allows for formatting strings easily
    sprintf(file, "/proc/%d/status", p[i]);
    fd = open(file, O_RDONLY);
    if ( fd == -1) {
      printf("Error opening '/proc/' file.\n");
    }
    bytes = read(fd, data, 2048);
    if(bytes == -1) {
      perror("read");
      free(data);
      close(fd);
    }
```

Figure <3>: code from part4() of ‘part4.c’ implementing creating file name using integers and characters.

Performance Results and Discussion

There were no large issues that arose during the implementation of this project, as stated earlier. The string formulation was fixed within minutes after combing through the Linux manual pages. It was difficult to choose which parameters to display for part 4 of the project, but it was decided by loading the status file of a running process and determining which were conducive to displaying processes. In the final version of part 4 the statistics being tracked are process name, state, id, data utilization, and the context switches the process undergoes. From the file ‘io’ under the process in ‘/proc/’ I used the data that corresponds to characters read and written which is interesting when observing ‘iobound.c’ execute.

Conclusion

Throughout this project I found most of the implementation to be straightforward and relatively concise. In this sense it seems to me as though processes are great for managing concurrent execution, are easily implemented, and may result in a higher efficiency than merely operating on one process.