Principles of Operating Systems

CMSC 421 — Spring 2022

Project 2: Simple Shell, /proc directory, CPU Scheduling Algorithms

100 points - see the rubric in blackboard for details

CHANGELOG

Nothing so far.

Introduction

In part 1 of this assignment, you will be writing a C program that works as a simple *nix shell program (*nix is shorthand for Unix or Linux machines). This part of the assignment only requires a few basic features of a shell and leaves out much of the functionality that more advanced shells such as Bash include. That is not to say that this assignment will be trivial — this is a 400-level course, after all. There are still several parts of this assignment that could trip you up, especially if you are not comfortable with lower-level C Programming.

In part 2 of this assignment, you will be adding functionality to explore information from the `/proc` filesystem on Linux and to organize it in a way to present it to the user.

This project is designed to help you get a warm-up with C, including topics that will be of importance going forward with your programming projects. This assignment is to be completed entirely in user-space — not within the Linux kernel source code and does not involve recompiling the kernel. You may complete this assignment outside of your VM for the class if you wish, however the TAs will be using the class Debian setup on a VM to grade the assignment (so you should at least compile and run your submitted assignment once in your VM to ensure it works before submitting it).

You are not allowed to use the GL system (or any other shared environment) at UMBC to complete this assignment! You can get into real trouble for this!

This assignment must be completed in the C programming language and must run on the Debian setup for this class.

What is a shell?

At its core, a shell is the interface between the linux kernel and users of the system. It is software that takes a string like `ls -la` from the user, formats it, then attempts to execute it. `ls` is an actual program located in `/bin` for instance. When you type `ls` in your shell, it finds that program and executes it. The shell itself does not execute any logic that says: `for each file in directory: print filename` [1]. It just invokes the program named `ls`. Modern shells have many extra bells and whistles, but you do not need to worry about them in this assignment. You do not have to create any GUIs, windows, or anything of that sort.

[1]: While some shells do have such functionality built-in, your shell will not. Your shell will have two very specific features that are described below.

Part 1

Requirements:

For this assignment, you will only need to support a few very basic features of a full-fledged *nix shell. Specifically, you will need to have support for all of the following:

1. When your shell program first executes, it should take no arguments. While it is running, the shell shall present the user with a prompt at which they can enter commands. Upon the completion of a command, it shall prompt the user for a new command.

2. If executed with any arguments, the shell shall print an error message to `stderr` explaining that it does not accept any command line arguments and exit with an error return code (1).

3. The shell shall accept a command input of arbitrary length (meaning you cannot set a hard limit on command length).

4. The shell shall parse command-line arguments from the user's input and pass them to the program that the user requests to be started. The command to be called will be either specified as an absolute path to a program binary (like `/bin/ls`), as the name of a program that resides in a directory in the user's `$PATH` environment variable (like `gcc`), or as a relative path (for instance, if we are in the `/usr` directory, we could type `bin/gcc` as a command to run `/usr/bin/gcc`. In addition, your argument parsing code must properly handle escape sequences and quoting. That is to say that the input `/bin/echo Hello\nWorld` should be parsed into two pieces — the program name `/bin/echo` and one argument to that program containing the string "Hello World" with an actual newline character in place of the space (and no quotes).

5. The shell shall support a built-in `exit` command. This command shall accept zero or one argument. If provided with zero arguments, the shell shall exit with a normal exit status (that is to say, it will exit with a status of 0). If provided with one argument, it shall attempt

to parse that argument as an integer. If this parsing fails, the command must be ignored, and the shell must prompt for another command. If the parsing succeeds, the shell shall exit with a status of whatever integer the argument parses. In either case of the shell exiting, it **MUST** clean up all memory it has allocated before exiting, along with ensuring that any child processes it has created have exited.

6. The shell shall not leak memory after it is done with it. The `valgrind` program can be your friend while debugging this program (unlike projects that are done in the kernel). We will also be using `valgrind` to test your implementation.

You can find examples of multiple commands that may be used to test your code at the bottom of this page.

NOTE: This is NOT an exhaustive list of commands that your shell must support. They are only given as examples. Any program installed on the VM should be runnable with your shell.

You are not expected to support any of the following features:

- Scripting control features (like if statements or loops).
- Reading input from a file (like a shell script).
- Support for pipes (including stdin/stdout redirection).
- Built-in functionality that is often part of a shell (such as implementations of common commands like cd), other than what is outlined above for exit and in part 2.
- Changing the current working directory in the shell.
- Running programs in the background or resuming backgrounded processes.

Part 2

`/proc` is a virtual file system in Linux. Most files in `/proc` appear to have zero length; however, you can view them using the cat command or an editor and see data in many of the files. These are not real files, but virtual files and symbolic links to files that exist elsewhere. Thus, they are not taking up real space on disk. Most files are read only, but others you can modify, and these will let you change the kernel's characteristics.

Read the article at the following URL to learn more about the /proc virtual filesystem: https://www.linux.com/news/discover-possibilities-proc-directory/ .

In part 2 of this assignment, you will explore some of the information available in the `/proc` filesystem. To create a process to use in this assignment, from another terminal run the `top` command in the background (type: "`top &`" without the quotes) and use the pid from this process for your exploration.

You are required to add functionality to your shell program that reads information from the `/proc` filesystem and displays it on the normal stdout of the shell. This command shall accept a single command line argument that will be the name of the file from the `/proc` filesystem that the user wishes to read information from. Only a small subset of all of the files in `/proc` shall be available for use with this command, as detailed below.

Your code will read information from the following files from the `/proc` filesystem:

- `cpuinfo` — displays various CPU characteristics such as the CPU speed in MHz, cache size, CPU core count, address sizes, etc.
- `loadavg` — displays the number of running processes and information about how much CPU time is being used currently (and over a couple different ranges).
- `filesystems` -- displays the type of file systems that the operating system recognizes.
- `mounts` – displays the mounted file systems.
- `pid/status` (with `pid` being a process ID running currently on the system) — displays various characteristics of the running process, including how much memory it is using, the owner of the process, its state, etc.
- `pid/environ` (with `pid` being a process ID running currently on the system) — displays information about the environment variables known by the specified process.
- `pid/sched` (with `pid` referencing a process ID running currently on the system) — displays information about scheduling characteristics of the process specified.

You may support more files in the /proc filesystem than those we have listed here, but you MUST at least support these.

The `proc` command shall be implemented by opening the file specified within the `/proc` filesystem, reading all input from the file, and printing all of that information to `stdout`, followed by a newline character. Your shell then shall prompt for a new command, as usual.

You are not required to parse the information you have read from the files in `/proc` at all prior to presenting it to the user. For instance, if you run the command proc cpuinfo, your output may look identical to running the command `cat /proc/cpuinfo`, however, you may not implement your code by actually running that command. **You must implement the `proc` command by opening the appropriate file in the `/proc` filesystem, reading from it, and printing out its output to `stdout`.**

Return an appropriate error if needed.

To sum up what you are expected to implement in this project:

• Present the user with some sort of prompt at which the user may enter a command to execute.
• Parse out the program the user is attempting to call from its arguments and build an appropriate argument array which can be used to execute the program.
• Determine if the program specified is a built-in (`proc`, or `exit`) and handle those functions without creating a new process or attempting to execute another program (except as might be required for `proc`).
• If the program specified is not built into the shell like "**exit**", your shell must create a new process to execute the new program and pass in the correct arguments to one of the `exec` family of functions to execute the program with the arguments provided. Your shell then must wait for the newly created process to finish executing. Your shell must also handle the case in which a program cannot be executed properly and print out an appropriate error message to `stderr`.
• Once the specified built-in or program has been executed (or failed executing), your shell should prompt the user for another command to run (unless the shell has exited from the `exit` built-in command).

Dos and Don'ts

Dos

Here is a list of functions that you might find it worthwhile to take a look at from the C library. You don't necessarily have to use all of them, but you may find several of them useful for this assignment:

- fgetc
- malloc
- realloc
- free
- strtok_r
- strchr
- isspace
- fork
- exec (a family of functions)
- fprintf
- getline

Additionally, you will want to make use of the utility functions that we have provided to you in the utils.c and utils.h files that are in your repository. Particularly useful in this code are the

functions unescape (which removes escape sequences and quotes from strings) and first_unquoted_space which will tell you the location of the next space in the string that is not quoted or part of an escape sequence. You are not required to use this code in your shell if you would rather implement this part yourself (but writing this code is difficult, it is probably easier to use the code we give you).

Don'ts

Your shell program is not allowed to use any external libraries other than the system's C library. Do not try to use libraries like Readline. You will lose points for using external libraries to implement shell functionality! In addition, you are not allowed to use any of the following functions in the C library to implement your shell:

- `system` (insecure and can lead to major problems)
- `scanf` (this is largely to save you trouble)
- `fscanf` (ditto)
- `popen` (no reason to use this with no pipe support)
- `readline` (in case this wasn't obvious from the above)

You are not allowed to implement any of your shell's functionality by calling on another shell to do the work. You must do the argument parsing and calling of programs in your own code!

A header file is **NOT** a library. In order to add an external library you have to link against it. So to link with the threading library for example, you would have to add -lpthreads to your build command in the Makefile. So as long as you are not adding an -lsomething in your build, or copying code from an external library into your code you should be ok.

`#include <stdio.h>` is not using an external library

Submitting the project

When submitting your shell program, please be sure to include the source code of the shell program (in one or more C source code files), as well as a Makefile that can be used to build the shell along with util.h and util.c. Your shell must be able to compile and run on a VM with Debian 11. Also, you must include a README.md file describing your approach to each of the requirements outlined above. Additionally, your program must be compiled to a binary called simple_shell with the Makefile you provide (again, the Makefile we have provided does this).

Create a zipped file with the following directory structure and files:

```
FirstName_LastName_project2.zip/
```

```
├─ extra_credit/
|   ├─ files for extra credit
├─ simple_shell/
|   ├─ files for simple_shell
|   ├─ helper files
|   ├─ Makefile
├─ scheduling_algorithms/
|   ├─ files for scheduling algorithms
|   ├─ helper files
|   ├─ Makefile
```

To submit the project, upload the zipped file on Blackboard under the project assignment. Be sure to include all of your uncompiled source files and your makefile. The GTAs will not add any files to your submission, you must include everything necessary to run your makefile.

You can use github to backup your work. Though github is not required for this project, it is convenient and you will need your github skills for the next project. You can choose to use a different application if you wish, but it is your responsibility to routinely backup your work so be sure to take care of this.

Hints

The code that is provided to you is very useful. It is highly suggested that you use it in your Shell.

The `first_unquoted_space` function provided can greatly ease the work of parsing a string into arguments. For instance, on the input `/bin/echo "Hello World"`, the function would return 9, which is the index of the first space. If run on the remainder of the string after that space, it would return -1, telling you that there are no further spaces in the string that are not quoted.

The `unescape` function allocates memory. If it returns non-NULL, you must free the value that it returns when you no longer need it. Additionally, the second argument to `unescape` should probably always be `stderr`.

You will lose points on this assignment for memory leaks. A shell is intended to be a long-running program and thus it is very important not to leak memory.

Examples

Here are a few commands that you can use and expand on for testing your shell. Please note that this is obviously not an exhaustive list and does not test every edge case.

```
ls
/bin/ls -la
ps -el
proc 1/status
echo \x48\151\x20\157\165\164\040\x74\x68\x65\x72\x65\041
echo Goodbye, \'World\'\a
exit 0
```

Here is an example of what the output might look like on your terminal from running those commands:

```
$ ls
CMakeLists.txt  Makefile  README.md  src
$ /bin/ls -la
$ /bin/ls -la
total 60
drwxr-xr-x 6 lj lj  4096 Feb 13 17:34 .
drwxr-xr-x 3 lj lj  4096 Feb 13 17:13 ..
-rw-r--r-- 1 lj lj 15314 Feb 13 17:13 .clang-format
-rw-r--r-- 1 lj lj   958 Feb 13 17:30 CMakeLists.txt
-rw-r--r-- 1 lj lj   293 Feb 13 17:13 .editorconfig
drwxr-xr-x 8 lj lj  4096 Feb 13 17:35 .git
-rw-r--r-- 1 lj lj    19 Feb 13 17:13 .gitignore
drwxr-xr-x 3 lj lj  4096 Feb 13 17:13 .idea
-rw-r--r-- 1 lj lj  1236 Feb 13 17:31 Makefile
-rw-r--r-- 1 lj lj  2829 Feb 13 17:13 README.md
drwxr-xr-x 2 lj lj  4096 Feb 13 17:34 src
drwxr-xr-x 2 lj lj  4096 Feb 13 17:13 .vscode
$ ps -el
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S     0     1     0  0  80   0 -   222 -      ?        00:00:00 init
5 S     0    99     1  0  80   0 -   222 -      ?        00:00:00 init
1 S     0   100    99  0  80   0 -   222 -      ?        00:00:00 init
4 S  1000   101   100  0  80   0 - 190697 futex_ pts/0    00:00:02 docker
1 Z     0   102    99  0  80   0 -     0 -      ?        00:00:00 init
<defunct>
1 S     0   113    99  0  80   0 -   222 -      ?        00:00:00 init
```

```
4 S     0   114   113  0  80   0 - 329161 -     pts/1    00:00:00 docker-
desktop-
5 S     0   140     1  0  80   0 -   222 -      ?        00:00:00 init
1 S     0   141   140  0  80   0 -   222 -      ?        00:00:01 init
4 S  1000   142   141  0  80   0 -  1780 do_wai pts/2    00:00:00 bash
0 S  1000   299   296  0  80   0 -  2753 -      pts/3    00:00:00 top
0 R  1000   350   142  0  80   0 -  2636 -      pts/2    00:00:00 ps
$ proc 299/environ
SHELL=/bin/bashWSL_DISTRO_NAME=DebianWT_SESSION=7db2b336-f609-4f16-b8d3-
97cf50e3b759
NAME=akagiPWD=/mnt/c/Users/Lawrence
SebaldLOGNAME=ljHOME=/home/ljLANG=en_US.UTF-8TER
TERM=xterm-
256colorUSER=ljPATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sb
in:/bin:/usr/games:/usr/local/games=/usr/bin/top
$ echo \x48\151\x20\157\165\164\040\x74\x68\x65\x72\x65\041
Hi out there!
$ echo Goodbye, \'World\'\a
Goodbye, 'World'
$ exit 0
```

Part 3 - Scheduling Algorithms

This part of the project involves implementing several different CPU process scheduling algorithms. The scheduler will be assigned a predefined set of tasks and will schedule the tasks based on the selected scheduling algorithm. Each task is assigned a priority and CPU burst. The following scheduling algorithms will be implemented:

1. **First-come, first-served (FCFS),** which schedules tasks in the order in which they request the CPU.
2. **Shortest-job-first (SJF)**, which schedules tasks in order of the length of the tasks' next CPU burst.
3. **Priority* scheduling**, which schedules tasks based on priority.
4. **Round-robin** (RR) scheduling**, where each task is run for a time quantum (or for the remainder of its CPU burst).
5. **Priority* with round-robin****, which schedules tasks in order of priority and uses round-robin scheduling for tasks with equal priority.

* Priorities range from 1 to 10, where a higher numeric value indicates a higher relative priority.
** For round-robin scheduling, the length of a time quantum is 10 milliseconds.

A. Implementation

The implementation of this project must be completed in C, and program files are provided with the project details. These supporting files read in the schedule of tasks, insert the tasks into a list, and invoke the scheduler.

The schedule of tasks has the form **[task name] [priority] [CPU burst],** with the following example format:

T1, 4, 20
T2, 2, 25
T3, 3, 25
T4, 3, 15
T5, 10, 10

Thus, task T1 has priority 4 and a CPU burst of 20 milliseconds, and so forth. It is assumed that all tasks arrive at the same time, so your scheduler algorithms do not have to support higher-priority processes preempting processes with lower priorities. In addition, tasks do not have to be placed into a queue or list in any particular order.

There are a few different strategies for organizing the list of tasks, as first presented in Section 5.1.2. of the textbook. One approach is to place all tasks in a single unordered list, where the strategy for task selection depends on the scheduling algorithm. For example, SJF scheduling would search the list to find the task with the shortest next CPU burst. Alternatively, a list could be ordered according to scheduling criteria (that is, by priority). One other strategy involves having a separate queue for each unique priority, as shown in Figure 5.7 of the textbook. These approaches are briefly discussed in Section 5.3.6 of the textbook. It is also worth highlighting that we are using the terms list and queue somewhat interchangeably. However, a queue has very specific FIFO functionality, whereas a list does not have such strict insertion and deletion requirements. You are likely to find the functionality of a general list to be more suitable when completing this project.

B. Execution Details

Along with this project overview you will be provided with the zip file (**project2_part3.zip**) which contains all the supportive code files to implement this part. The file driver.c reads in the schedule of tasks, inserts each task into a linked list, and invokes the process scheduler by calling the schedule() function. The schedule() function executes each task according to the specified scheduling algorithm. Tasks selected for execution on the CPU are determined by the pickNextTask() function and are executed by invoking the run() function defined in the CPU.c file. A Makefile is used to determine the

specific scheduling algorithm that will be invoked by driver. For example, to build the FCFS scheduler, we would enter

make fcfs

and would execute the scheduler (using the schedule of tasks in the input file schedule.txt) as follows:

./fcfs schedule.txt

Important Notes:

It is very highly recommended to refer to the README file in the source code download for further details. Before proceeding, be sure to familiarize yourself with the source code provided as well as the Makefile.

Make sure to name your files as "schedule_<algorithm_name>.c", where algorithm names are "fcfs", "sjf", "rr", "priority", "priority_rr".

Your implementation MUST include printf statements for each algorithm as follows:

1. A brief, well-formatted explanation of the algorithm demonstrating your understanding of the algorithm before it starts executing.
2. Informative statements that briefly explain each step of the algorithm as it is taking place.

In effect, Output should look like a step by step display of the algorithm execution.

You are allowed to take inspiration from solutions available online, but it is solely your responsibility to make sure that they are correct and working as expected with the provided code architecture. You must document any online sources in the README file.

Extra Credit:

You probably have already worked on this activity in the class. You just have to resubmit the files for this activity along with the other project 2 files to receive (extra) credit.

Practice with fork()

The objective of this activity is to help students understand how fork() works and to prepare for Project 2. There are 3 parts.

Note: Do not use these commands anywhere except on your debian machine inside a vm. These commands can crash an operating system.

Part 1

At the end of this document is a sample program that uses fork(). Create a c program on your debian virtual machine and run this program.

Part 2

Copy the fork demo program to another file and name it ***LastnameFirstname_3forks.c***. Then edit the program to call fork 3 times and display the parent and child pids for each process that is created during the program. Using the pids, draw a process tree hierarchy where the node values are the pids and the edges represent the relationships between parent and children nodes. Look up these functions in the man pages if you have questions about how they work.

Part 3

Conduct a small experiment with this program. Copy the program from part 2 and rename it ***LastnameFirstname_forkExperiment.c***. Use your imagination to create your own experiment. Examples could be changing the sleep times to alter when processes terminate, or creating orphans by also removing the wait() function, or creating 4 forks. You can also look up these commands in the man pages and experiment with different features and/or parameters.

Write 7 - 10 sentences describing your experiment, what you expected the results to be, the actual results, and whether or how the results were different from what you expected.

What to submit

Create a google doc while you are working with your answers, then save it as a pdf named ***LastnameFirstname_forkActivity.pdf*** and submit it along with project 2. The pdf should contain your written answers and examples of output from the activities above.

Submit your c programs

Submit a photo or pdf of your process tree. The process tree can be neatly hand drawn.

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>

int main()
{
   pid_t retVal;

   retVal = fork();
   if(retVal < 0){
      printf("fork() failed\n");
      return 1;
   }
   else if(retVal == 0){
      printf("fork1 retVal == 0 ");
      printf("in child process pid = %d \n", getpid() );
      printf("child sleeping ... \n" );
      sleep(5);
      printf(" finished sleeping\n" );
   }
   else {
      printf("parent pid = %d \n", getpid() );
      printf("fork1 in parent process waiting for child ...\n");
      wait(NULL);
      printf("wait() finished in parent process \n");
   }
   return 0;
}
```

Here is a sample of the output:

```
parent pid = 9657
fork1 in parent process waiting for child ...
fork1 retVal == 0 in child process pid = 9658
child sleeping ...
 finished sleeping
wait() finished in parent process
```