CMPUT 379 Fall 2023

Assignment 1

Mini Shell

Description

> In computing, a **shell** is a user interface for access to an operating system's services. In general, operating system shells use either a command-line interface (CLI) or graphical user interface (GUI), depending on a computer's role and particular operation. It is named a shell because it is the outermost layer around the operating system kernel. [Wikipedia]

This assignment is for you to build a CLI shell interface. It will accept a set of simple commands that your program will execute. The assignment has several important goals, including exposing you to systems programming, running multiple processes, resource management, and process communication.

The shell being built in this assignment has three important characteristics:

- Minimalist functionality: only a small set of features is being required for the assignment. The intent is to maximize the pedagogical value for the effort expended. Hopefully in doing this assignment you will appreciate how easy it is to add useful functionality to the shell that you build.
- Simple interface: command lines for the shell are simple and structured. Clearly no widely-used shell would accept such constraints. The intent here is to reduce the amount of work needed to program the parsing of command lines. You won't learn much if all your effort goes to writing code to implement the command line interface.
- As defined in the assignment, some of the implementation details may differ from what would be seen if one were trying to build a high-performing, widely-used product. Here the emphasis is to have the student explore using a number of system interfaces, even if the solution might not be the best in the given context. Again, the intent is to maximize the pedagogical value for the effort expended.

Specifications

You will write a program called `shell379` that accepts and executes the following commands. Some of the commands accept an integer parameter (`<pid>`).

`exit`	End the execution of `shell379`. Wait until all processes initiated by the shell are complete. Print out the total user and system time for all processes run by the shell.
`jobs`	Display the status of all running processes spawned by `shell379`. See the print format below in the example.
`kill <pid>`	Kill process `<pid>`.
`resume <pid>`	Resume the execution of process `<pid>`. This undoes a `suspend`.
`sleep <int>`	Sleep for `<int>` seconds.

`suspend <pid>` Suspend execution of process `<pid>`. A `resume` will reawaken it.
`wait <pid>` Wait until process `<pid>` has completed execution.

If none of the above commands is input, then the resulting input string is to be executed by `shell379`.

`<cmd> <arg>*` Spawn a process to execute command `<cmd>` with 0 or more arguments `<arg>`.[1] `<cmd>` and `<arg>` are each one or more sequences of non-blank characters.

Any command can be supplemented by three special arguments:

& If used, this must be the last argument and indicates that the command is to be executed in the background.

<fname This argument is the "<" character followed by a string of characters, a file name to be used for program input.

>fname This argument is the ">" character followed by a string of characters, a file name to be used for program output.

The above syntax is overly restrictive, again to limit the amount of programming.

Sample Output

`shell379` input lines are shown in bold. All output is in regular font. `Runner` and `sleeper` are programs Jonathan Schaeffer wrote to test out the shell; they are not part of the assignment. Processes can have the status of "R" (running) or "S" (stopped/suspended).

```
SHELL379: jobs
Running processes:
Processes =      0 active
Completed processes:
User time =      0 seconds
Sys  time =      0 seconds
SHELL379: cat input
15
200000000000
SHELL379: time runner <input
       15.77 real        11.30 user         3.39 sys
SHELL379: jobs
Running processes:
Processes =      0 active
Completed processes:
User time =     11 seconds
Sys  time =      3 seconds
SHELL379: runner <input >output &
SHELL379: jobs
Running processes:
 #    PID S SEC COMMAND
 0: 56188 R   0 runner <input >output &
```

[1] The "*" is often used in shell programming to indicate "0 or more" of something. The "+" can mean "1 or more", depending on the context.

```
Processes =      1 active
Completed processes:
User time =     11 seconds
Sys  time =      3 seconds
SHELL379: sleeper 5 &
SHELL379: sleeper 6 &
SHELL379: jobs
Running processes:
 #    PID S SEC COMMAND
 0: 56188 R  10 runner <input >output &
 1: 56190 R   0 sleeper 5 &
 2: 56192 R   0 sleeper 6 &
Processes =      3 active
Completed processes:
User time =     11 seconds
Sys  time =      3 seconds
SHELL379: wait 56188
SHELL379: jobs
Running processes:
 #    PID S SEC COMMAND
 0: 56192 R   0 sleeper 6 &
Processes =      1 active
Completed processes:
User time =     22 seconds
Sys  time =      6 seconds
SHELL379: runner <input &
SHELL379: jobs
Running processes:
 #    PID S SEC COMMAND
 0: 56205 R   0 runner <input &
Processes =      1 active
Completed processes:
User time =     22 seconds
Sys  time =      6 seconds
SHELL379: kill 56205
SHELL379: jobs
Running processes:
Processes =      0 active
Completed processes:
User time =     28 seconds
Sys  time =      8 seconds
SHELL379: exit
Resources used
User time =     28 seconds
Sys  time =      8 seconds
```

Implementation

Your program will maintain a Process Table (Process Control Block) that contains information on all currently running processes created by `shell379`. The `jobs` command prints out the contents of the Process Table. Running a `<cmd>` adds an entry to the table. `Kill` ends a

process and removes it from the table. As jobs finish, they are removed from the table. Resume/suspend change the execution of a process, and this state change has to be updated in the table.

The `jobs` command displays two sets of times. Under the heading "`Completed processes`", the times given are the total execution times of all completed processes. Under the "`Running processes`" heading, the time given for each process is the current amount of execution time used. For this assignment, you are to get the information from the `ps` command and use a pipe to access the data.

Some of the important system calls you should use in your implementation include `exec` (note that there are multiple variations of `exec`), `fork`, `getrusage`, `kill`, `popen`, `signal` (again, there are multiple `sig`-related calls), `times`, `wait` and `perror`. Note that there may be different implementation solutions for each of the `shell379` commands. You are not allowed to use the system call `system`.

Your program must be implemented in C or C++. Create a `makefile` to compile your program and produce an executable called `shell379`. Your program must consist of at least three source code files and at least one header file. Your code should be logically organized between these files. The makefile should be constructed to do the minimum amount of work needed to recompile your program.

To make things simple, we will use some constants in your program. Again, this is to minimize the programming effort:

```
LINE_LENGTH    100    // Max # of characters in an input line
MAX_ARGS         7    // Max number of arguments to a command
MAX_LENGTH      20    // Max # of characters in an argument
MAX_PT_ENTRIES  32    // Max entries in the Process Table
```

A useful resource for programming this assignment is Chapter 5 of *Three Easy Pieces*.

Warning

During your development of a solution to this assignment, an incorrect implementation may leave processes running in the background after you log out. It is your responsibility to make sure that you clean up all processes.

Consider using the command `processfence` to monitor your creation of processes and kill them if needed. It is only available on the CS undergraduate machines.

Submission

You may submit your assignment for feedback from the TAs. This is optional – the assignment is not being graded.

All assignment assessments will be done on the undergraduate lab machines. Before you submit your assignment, please verify that your program compiles and runs on the lab machines. A submitted assignment will only be looked at by the TAs if:

- It compiles cleanly on the CS undergraduate machines.
- It runs without dying on the CS undergraduate machines.

Feedback will consist of looking at key parts of the functionality required in this assignment and offering feedback as to whether you have the right approach. Other details (e.g., output format; completeness of implementation) will not be considered.

Submit the following as a single ZIP or TAR file:

- Source code files (C, C++, headers),
- `Makefile`, and
- (optionally) a file called README containing anything special that may help the TA understand the approach taken in your assignment.

The zip/tar file contains only the above files with no directory structure, and no additional files.

The assignment is due no later than 10:00 PM on Sunday, October 1. We are on a tight schedule to get you feedback before the Assignment 1 Test. Thus, there is no guarantee that a late submission will receive feedback.

Collaboration

It is strongly recommended that you do the assignment individually. However, discussion/ collaboration with other students is permitted. Since the assignment is not being graded, you do not have to implement all the specifications given in the assignment to the exactness you would need if your work was being marked.

Warning! Reading code gives you a simplistic idea of how things work. Implementing code gives you a much deeper understanding. Remember that the first CMPUT 379 test will test whether you understand the implementation ideas needed to do this assignment.