CMPUT 379 Winter 2020

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 (<int>).

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 <int>	Kill process <int>.
resume <int>	Resume the execution of process <int>. This undoes a suspend.
sleep <int>	Sleep for <int> seconds.

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

There are three special arguments that a command may have:

&	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.

```
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:
```

[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.

```
 #    PID S SEC COMMAND
 0: 56188 R   0 runner <input >output &
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 (or Process Control Block) that contains information on all currently running processes. 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 system calls you might consider for your implementation include `exec()`, `fork()`, `getrusage()`, `kill()`, `popen()`, `signal()`, `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.

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 *Five Easy Pieces*.

Grading

Here are some important things to watch out for:

- You will be penalized if your program leaves processes running after it exits.
- Your program output should match that given in this document.
- Your `makefile` should do the minimum amount of work required to produce an executable.
- Make sure your program compiles and runs on the lab machines!

Submission

Submit the following:

- All source code files (C, C++, headers), and
- `Makefile`.

The assignment is due no later than 10:00 PM on Sunday, February 2. Late assignments received before 10:00 PM on Monday, February 3 will have their assignment grade lowered by 20%.