

CSCE 3600: Systems Programming

Major Assignment 1 – The Shell and System Calls

Due: 11:59 PM on Wednesday, March 30, 2016

COLLABORATION:

You should complete this assignment as a group assignment with the other members of your group. Each group should have 3 or 4 members, no more, no less. Submit only ONE program per group. Also, make sure that you list the names of all group members who participated in this assignment in order for each to get credit.

GOALS:

There are four objectives to this assignment:

- To familiarize yourself with the Linux programming environment;
- To develop programming skills in C;
- To gain exposure to the necessary functionality in shells; and
- To learn how processes are handled (i.e., starting and waiting for their termination).

PROGRAM DESCRIPTION:

In this assignment, you will implement a command line interpreter or shell. The shell should operate in this basic way: when you type in a command (in response to its prompt), the shell creates a child process that executes the command you entered and then prompts for more input when it has finished.

The shell that you implement will be similar to, but much simpler than, the one you run in everyday UNIX/Linux. You can find out which shell you are running by typing `echo $SHELL` at a prompt. You may then wish to look at the man pages for `bash` to learn about all of the functionality that can be present. For this project, you do not need to implement much functionality, but you will need to be able to handle running multiple commands simultaneously.

Your shell can be run in two ways: (1) **interactive** and (2) **batch**. In interactive mode, you will display a prompt (any string of your choosing) and the user of the shell will type in a command at the prompt. In batch mode, your shell is started by specifying a batch file on its command line. The batch file contains the list of commands that should be executed. In batch mode, you should **not** display a prompt. In batch mode, you should echo each line you read from the batch file back to the user before executing it. This will help you when you debug your shells. In both interactive and batch mode, your shell stops accepting new commands when it sees the `quit` command on a line or reaches the end of the input stream (i.e., the end of the batch file or the user types 'Ctrl-D'). The shell should then exit **after** all running processes have terminated.

Each line (of the batch file or typed at the prompt) may contain multiple commands separated with the semi-colon (;) character. Each of the command separated by a ; should be run simultaneously, or concurrently. Note that this is different behavior than standard UNIX/Linux shells that run these commands one at a time, in order. The shell should not print the next prompt or take more input until **all** of these commands have finished executing (the `wait()` and/or `waitpid()` system calls may be useful here). For example, the following lines are all valid and have reasonable commands specified:

```
prompt>
prompt> ls
prompt> /bin/ls
prompt> ls -l
prompt> ls -l; cat file1
prompt> ls -l; cat file1; grep Dallas file2
```

For example, on the last line, the commands `ls -l`, `cat file`, and `grep Dallas file2` should all be running at the same time. As a result, you may see that their output is intermixed.

To exit the shell, the user can type `quit`. This should just exit the shell and be done with it (the `exit()` system call will be useful here). Note that `quit` is a built-in shell command. It is not to be executed like other programs the user types in. If the `quit` command is on the same line with other commands, you should ensure that the other commands execute (and finish) before you exit your shell.

These are all valid examples for quitting the shell:

```
prompt> quit
prompt> quit; cat file1
prompt> cat file1; quit
```

Although most of the commands that users type at the prompt are the name of other UNIX/Linux programs, such as `ls` or `more`, shells recognize some special commands called internal commands that are not program names. For example, the `exit` command terminates the shell, and the `cd` command changes the current working directory. Shells directly make system calls to execute these commands, instead of forking a child process to handle them. In addition to creating a shell that knows how to launch new programs, your shell will also recognize two internal commands (`exit` and `cd`) that work by calling existing system calls, `exit` and `chdir`.

Writing your shell in a simple manner is a matter of finding the relevant library routines and calling them properly.

REQUIREMENTS:

Your C program must be invoked exactly as follows:

```
shell [batchFile]
```

The command line arguments to your shell are to be interpreted as follows:

- `batchFile`: an optional argument (indicated by square brackets as above). If present, your shell will read each line of the `batchFile` for commands to be executed. If not present, your shell will run in interactive mode by printing a prompt to the user at `stdout` and reading the command `stdin`.

For example, if you run your program as:

```
shell /home/mat0299/csce3600/batchfile
```

then it will read commands from `/home/mat0299/csce3600/batchfile` until it sees the `quit` command.

Defensive programming is an important concept in operating systems: an OS cannot simply fail when it encounters an error. It must check all parameters before it trusts them. In general, there should be no circumstances in which your C program will core dump, hang indefinitely, or prematurely terminate. Therefore, your program must respond to all input in a reasonable manner. By “reasonable”, this means that you should print a meaningful and understandable error message and either continue processing or exit, depending upon the situation.

You should consider the following situations as errors – in each case, your shell should print a message to `stderr` and exit gracefully:

- An incorrect number of command line arguments to your shell program; and
- The batch file does not exist or cannot be opened.

For the following situation, you should print a message to the user (`stderr`) and continue processing:

- A command does not exist or cannot be executed.

Optionally, to make coding your shell easier, you may print an error message and continue processing in the following situation:

- A very long command line (for this project, over 512 characters including the “\n”).

Your shell should also be able to handle the following scenarios, which are not errors (i.e., your shell should not print an error message):

- An empty command line;
- Extra white spaces within a command line; and
- Batch file ends without `quit` command or user types ‘Ctrl-D’ as command in interactive mode.

In no case should any input or any command line format cause your shell program to crash or exit prematurely. You should think carefully about how you want to handle oddly formatted command lines (e.g., lines with no commands between a semi-colon). In these cases, you may choose to print a warning message and/or execute some subset of the commands. However, in all cases, your shell should continue to execute.

```
prompt> ; cat file1 ; grep Dallas file2
prompt> cat file1 ; ; grep Dallas file2
prompt> cat file1 ; ls -l ;
prompt> cat file1 ;;;; ls -l
prompt> ;; ls -l
prompt> ;
```

In addition to the above, your shell should recognize two internal commands: `exit` and `cd`. `exit` terminates the shell, i.e., the shell calls the `exit()` system call or returns from `main`. `cd` uses the `chdir()` system call to change to the new directory.

OPTIONAL SHELL FUNCTIONALITY:

The shell that you are building is fairly simplistic. For teams who have completed all requirements for this program and are looking for an additional challenge, teams may add the following optional functionality to gain bonus points added to your team's overall score:

- Add a `PATH` variable.
- Add a shell history of previous commands run on the shell.
- Allow the user to customize the prompt.

However, all required functionality must be implemented prior to attempting this extra credit work as no points will be given for attempting this functionality if all requirements have not been completed. In other words, make sure your program is complete before attempting this extra credit.

GRADING:

This assignment must be submitted via Blackboard with the following elements:

- Your code should be well documented in terms of comments. For example, good comments in general consist of a header (with your name, course section, date, and brief description), comments for each variable, and commented blocks of code.
- A **README** file with some basic documentation about your code. This file should contain the following four components:
 - Your name(s).

- Design Overview: A few paragraphs describing the overall structure of your code and any important structures.
- Complete Specification: Describe how you handled any ambiguities in the specification. For example, for this project, explain how your shell will handle lines that have no commands between semi-colons.
- Known Bugs or Problems: A list of any features that you did not implement or that you know are not working correctly.
- A `Makefile` for compiling your source code, including a clean directive.
- Your program will be graded based largely on whether it works correctly on the CSE machines (e.g., `cse01`, `cse02`, ..., `cse06`), so you should make sure that your program compiles and runs on a CSE machine.

To ensure that your C code is compiled correctly, you will need to create a simple `Makefile`. This allows our scripts to just run `make` to compile your code with the right libraries and flags. If you do not know how to write a `Makefile`, you may reference the man pages for `make` or read the accompanying tutorial for this assignment. Here is a sample `Makefile` that may help as well:

```
#####
#
# shell.c is the name of your source code; you may change this.
# However, you must keep the name of the executable as "shell".
#
# Type "make" or "make shell" to compile your code
#
# Type "make clean" to remove the executable (and object files)
#
#####
CC=gcc
CFLAGS=-Wall -g
shell: shell.c
    $(CC) -o shell $(CFLAGS) shell.c
clean:
    $(RM) shell
```

Your program will be tested using a suite of about 20 test cases on the CSE machines, some of which will exercise your program's ability to correctly execute commands and some of which will test your program's ability to catch error conditions. Be sure that you thoroughly exercise your program's capabilities on a wide range of test suites.

SUBMISSION:

- You will electronically submit your bash program to the **Major Assignment 1** dropbox in Blackboard by the due date.