Systems and Networks	Project 1: UNIX Shell
CS3600 Spring 2013	January 16, 2013

This project is due at 11:59:59pm on February 4, 2013 and is worth 10% of your grade.. You must complete it with a partner. You may only complete it alone or in a group of three if you have the instructor's explicit permission to do so for this project.

Note that there is a milestone deadline at 11:59:59pm on January 28, 2013. More details are in the Milestone section below.

1 Description

There are several goals for this assignment. First, you will get a chance to familiarize yourself with the C programming language. Second, you will learn how to use some of the I/O facilities and library functions provided by UNIX and C. Third, you get some experience with the process creation and program starting facilities in UNIX. Fourth, you'll get some experience making your program robust (i.e., resistant to crashing).

Your assignment is to write a shell: a program that starts up other programs. Your shell, called `3600sh` will read input lines from standard input, parse them into a command name and arguments, and then start a new process running that command. When you start your shell, it will provide a prompt and then wait for a command line of the form:

```
foobar -l this and that
```

This starts a program stored in a file called `foobar` with four arguments: `-l`, `this`, `and`, and `that`.

2 Requirements

Your shell will read commands, line by line, from standard input, execute the commands, and write the output of commands to standard output

2.1 Starter code

Very basic starter code for the assignment is available in `/course/cs3600sp13/code/project1`. You must use this code as a basis for your project. Provided is `3600sh` start code and header file, and a `Makefile`. The `Makefile` is configured to correctly compile your code and to test your code on sample input.

To get started, you should copy down this directory into your own local directory (i.e., `cp -r /course/cs3600sp13/code/project1 ~/cs3600`). You can compile your code by running `make`. You can also delete any compiled code and object files by running `make clean`.

2.2 Input format

Each line has exactly one command and zero or more arguments, and is of the form:

```
cmd arg_1 ... arg_k
```

Each of these items is separated by one more more blank characters (spaces or tabs). There is one command per input line. Your shell will fork a child process and then overwrite itself (via `exec`) with the command in the file named by `cmd`. The shell parent process will default to waiting for the child process to complete before prompting for the next command.

2.3 Features

Your client must support the following features:

Command prompt When requesting input, your shell should print a prompt with the format

```
[user]@[host]:[dir]>[space]
```

For example, this might look like

```
amislove@joshua:/usr>
```

If, for any reason, you are unable to launch the program that the user requested, you should print out an error

```
Error: [meaningful error]
```

and then issue another prompt. In particular, if you are unable to find the requested executable, you should print out

```
Error: Command not found.
```

Or, if the requested executable is not executable by you due to a permissions error, you should print

```
Error: Permission denied.
```

Arguments You must support an arbitrary number of command-line arguments for each command, and each argument may be of arbitrary length. Note that there may be any number of spaces or tabs between arguments, at the beginning of the command, and at the end of the command. Your shell should "eat" these extra spaces or tabs, and they should not be passed to the program. For example, if you see

```
␣␣␣ls␣␣␣␣-a␣␣-l␣␣␣␣␣␣
```

(where ␣ indicates a space or tab) should have the same exact effect as the command

```
ls␣-a␣-l
```

Background processes Optionally, at the end of any command input, can be a `&` character, which means that the parent process does not wait for the child process to complete before prompting for the next command. Note that the `&` character may be followed by spaces or tabs (which you should "eat"), and may *not* be separated by spaces or tabs from the final argument of the command. For example, the command

```
␣␣␣ls␣␣␣␣-a&␣␣␣␣␣␣
```

should have the same effect as the command

```
ls␣-a␣&
```

If any arguments follow a `&`, your shell should print out

```
Error: Invalid syntax.
```

Escape characters Your shell should allow users to pass arguments with spaces in them through the use of backslash-space (`\␣`) and with tabs through the use of backslash-t (`\t`). For example, if the user runs

```
echo foo\ bar\tbaz
```

You should launch the `echo` program with a single argument of `foo bar  baz`, rather than three separate arguments. Note that the backslash characters are "eaten" by your shell. Your shell should allow users to send the backslash character by entering a double backslash (`\\`). For example, if the user runs

```
echo foo \\bar blah\ baz
```

You should launch the `echo` program with three arguments of `foo`, `\bar`, and `blah baz`. Similarly, you should allow users to send the ampersand character through the use of backslash-ampersand (`\&`). For example,

```
echo ls\&\  &
```

should launch `echo` with one argument `ls&␣` (note the space), and should do so in the background.

If the user enters a backslash followed by any character other than a `\`, space, `t`, or `&`, your shell should print out

```
Error: Unrecognized escape sequence.
```

Input/output redirection Your shell should allow the user to specify that a command be run with `STDOUT`, `STDIN`, and/or `STDERR` being read from/written to a file. The syntax for redirecting standard input is

```
cmd arg1 arg2 ... argN < stdin_file
```

where `stdin_file` is the name of the file that should be fed to standard input to `cmd`. Similarly, redirecting STDOUT is done by

```
cmd arg1 arg2 ... argN > stdout_file
```

and STDERR is done by

```
cmd arg1 arg2 ... argN 2> stderr_file
```

Both > and 2> should create the file if it does not exist, and should truncate the file to zero-length before writing.

Of course, these redirections can be combined in any order, so you should support commands like

```
cmd arg1 arg2 ... argN < stdin_file 2> stderr_file > stdout_file
```

Each redirection must contain a filename, and each redirection can only be used once. If the file does not exist, or if you do not have the appropriate permissions to read/write it, you should print out

```
Error: Unable to open redirection file.
```

If redirections exist, they **must** come *after* the command and all of its arguments, but *before* any ampersand (the backgrounding character). Thus, if you receive a command like

```
cmd arg1 arg2 ... argN & < stdin_file
```

or

```
cmd arg1 arg2 ... argN 2> <
```

or

```
cmd arg1 arg2 ... argN > &
```

or

```
cmd arg1 arg2 ... argN > file > file2
```

you should print out

```
Error: Invalid syntax.
```

Shell commands Your shell should support the special `exit` command, which causes the shell to exit. You should print out

```
So long, and thanks for all the fish!
```

3 Running and debugging your program

Read this section *in its entirety* before your begin implementing and testing your shell. If you have any questions, please contact the course staff.

In past semesters, students have managed to take down various CCIS resources by incorrectly using functions like `fork` and by leaving old processes burning the CPU. A few notes:

Do not run your code on `login.ccs.neu.edu` This machine is the central login server, so IT asks that you not develop your code there. Instead, use any other CCIS machine (fun fact: you can `ssh` into any of the lab machines, and those serve as great places to develop and test your code).

Do not leave old processes hanging around In the past, students have caused the load average (the average number of processes waiting to run) on CCIS machines to spike in the hundreds, which makes the machines rather unusable. Make sure to clean up any old processes you have left running (see notes on `ps`/`kill` below).

Be careful with `fork()` Recall that I mentioned fork bombs in class – this is when a process recursively `fork()`s, causing more and more processes to be created. Make sure that you code doesn't do this. Your code should *never* recursively call `fork` (a child forking another child). Double- and triple-check your `fork()` calls to make sure this doesn't happen.

Learn how to use ps and kill To list the processes that you have running, use the `ps` command:

```
bash$  ps
  PID TTY          TIME CMD
36284 ttys000    0:00.28 -bash
39471 ttys001    0:00.34 -bash
```

This means that I have two `bash` process running. If you see, like, 100 `3600sh` processes, congratulations! — your processes is likely running a fork bomb. To fix this, you can run

```
bash$  killall -KILL 3600sh
```

Sometimes you'll have to run this multiple times in succession in order to make sure they're all dead. Don't worry; this will only kill your processes (and no one else's). Note that if the `3600sh` process is running under your partner's login, she/he will have to be the one to run that command.

Every so often (and before you leave the machine) check to make sure that you haven't left any processes hanging around using up CPU.

4 Implementation hints

The command line is processed (by the shell) into a list of character strings, one for each argument (including the command name). These arguments are passed as parameters to the exec command (you'll probably want to use `execvp`). Make sure that your program can handle input lines that

have very long command names and arguments. You must also be able to deal gracefully with command lines that are arbitrarily long; you must be able to gracefully reject them.

You should develop your client program on the CCIS Linux machines, as these have the necessary compiler and library support. You are welcome to use your own Linux/OS X machines, but you are responsible for getting your code working, and your code *must* work when graded on the CCIS Linux machines. If you do not have a CCIS account, you should get one ASAP in order to complete the project. Your code must be `-Wall` clean on `gcc`. Do not ask the TA for help on (or post to the forum) code that is not `-Wall` clean unless getting rid of the warning is what the problem is in the first place.

You will want to make sure that you break your program up into modules, such that each module represents a sensible type abstraction. If you have questions on how to do this, please come see me or the TA.

5 Testing

First test that your program can start up simple programs and pass parameters to them. Then test it by running standard UNIX/Linux utilities like `ls`. Your program should not crash, no matter how weird the input; in all cases, your program should print an error and ask for another command. For example, suppose that the input has zero-value characters (bytes) or lines that have 1,000,000 characters? As a result, you may not be able to use the input routines that first occur to you.

You can create test scripts to test your shell with by simply putting commands in a test file, one on each line. For example, you can create the file `script.txt` with the contents

```
ls -al
whoami
foobar
```

and then run your shell on this input with

```
bash$ ./3600sh < script.txt
```

Your shell should read the input file as if it were coming from the keyboard, and print the results (and any error messages) out to your terminal.

Additionally, we have included a basic test script to check the output of your code against our reference solution and check your code's compatibility with the grading script. If your code fails in the test script we provide, you can be assured that it will fare poorly when run under the grading script. To run the test script, simply type

```
bash$ make test
```

This will compile your code and then test your shell on a number of inputs, comparing the results against the reference solution. If any errors are detected, the test will print out the expected and actual output. For example, you might see something like

```
bash$ make test
./test
  Trying with script 'ls'                                    [FAIL]
```

```
    Diff in expected output:
< amislove@bubbles:/home/amislove/project1> 3600sh
< 3600sh.c
< 3600sh.h
< Makefile
< test
> blah
bash$
```

This indicates that the test with input `./cs3600sh` was expected to print out the lines preceded by <, but instead returned `blah` (the output is simply a `diff` between the output of your shell and the reference shell). We include a few sample tests, but these are by no means exhaustive. We expect that you will create additional test to ensure that your programs behave as expected.

6 Submitting your project

6.1 Registering your team

You and your partner should first register as a team by running the `/course/cs3600sp13/bin/register` script. You should pick out a team name (no spaces or non-alphanumeric characters, please) and run

```
/course/cs3600sp13/bin/register project1 <teamname>
```

This will either report back success or will give you an error message. If you have trouble registering, please contact the course staff.

You must register your team by 11:59:59pm on January 21, 2013.

6.2 Milestone

In order to ensure that you are making sufficient progress, you will have an interim milestone deadline. For the milestone, you do not need to submit any README or documentation; you only need to submit your code. You code must successfully launch programs and support the `exit` shell command (for example, all of the tests marked "Milestone Tests" should pass). It need not support background processes or backslash-quote. For the milestone, we will not be testing it against malicious (incorrectly formatted) input.

You should submit your milestone by running the `/course/cs3600sp13/bin/turnin` script. Specifically, you should create a `project1` directory, and place all of your code in it. Then, run

```
/course/cs3600sp13/bin/turnin project1-milestone <dir>
```

Where `<dir>` is the name of the directory with your submission. The script will print out every file that you are submitting, make sure that it prints out all of the files you wish to submit!

You must submit your milestone by 11:59:59pm on January 28, 2013. No slip days can be used on the milestone.

6.3 Final submission

For the final submission, you should submit you (thoroughly documented) code along with a plain-text (no Word or PDF) README file. In this file, you should describe your high-level approach, the challenges you faced, a list of properties/features of your design that you think is good, and an overview of how you tested your code. In your README, you should also point out any extra features of your project that you have implemented.

You should submit your project by running the `/course/cs3600sp13/bin/turnin` script. Specifically, you should create a `project1` directory, and place all of your code and README files in it. Then, run

```
/course/cs3600sp13/bin/turnin project1 <dir>
```

Where `<dir>` is the name of the directory with your submission. Again, the script will print out every file that you are submitting, make sure that it prints out all of the files you wish to submit!

You must submit your project by 11:59:59pm on February 4, 2013.

7 Grading

The grading in this project will consist of

50% Program functionality

25% Correct error handling

15% Style and documentation

10% Milestone functionality

8 Advice

A few pointers that you may find useful while working on this project:

- You should check the output of your program versus other shells. On Linux/OS X/UNIX, you can use the `sh` shell, which provides basic shell functionality.
- Check the Piazza forum for question and clarifications. You should post project-specific questions there first, before emailing the course staff.
- Finally, get started early and come to the TA lab hours—these are held in the lab at 102 West Village H. You are welcome to come to the lab and work, and ask the TA any questions you may have.