Semester Project – A UNIX Shell
Villanova University
Fall 2013 – CSC1600: Operating Systems

LEARNING OUTCOMES

Once you have completed this project you will have coded a Linux shell.

PROJECT OVERVIEW

You have decided none of the many available shells satisfies your needs. As such, you have decided to code your own shell. You have decided that at a minimum your shell must support the following functionality: process execution, I/O redirection, background process execution and signal handling.

Complete this program on felix. (Do not try to complete this project on any of our Solaris boxes. There are subtle differences between the system functions on Linux and Solaris.)

This is a pair programming assignment. Work on this program with your semester programming partner following the guidelines in the pair programming introductory document. With the exception of phase 06, you and your partner must submit one working program for each phase of the assignment. To submit a program, rename your .c file to myShellPhaseXX.c and submit a word-processed document listing the complete path name of your saved code. If you have created any header files, list them as well. Phase 06 is an individual assignment. For phase 06, each of you must submit your own word-processed document.

PROJECT DESCRIPTION

The input to your shell will be a sequence of commands entered interactively from the keyboard with each command entered on a separate line of input. You must support the following basic commands.

`programname [args]`	This command must execute the program with the name `programname` and the optional argument list. Example(s):

```
my-shell$: ./primes 4
2
3
4
7
```

```
my-shell$:ls -al
total 4
drwx------ 2 nbercich users  . . .
drwx------ 8 nbercich users  . . .
-rw------- 1 nbercich users . . .
my-shell$:
```

`exit`	This command must exit the shell
`resume`	This command must resume a suspended program.

In addition, your shell must support the execution of background processes and I/O redirection.

`programname [args]&`	This command must execute the program with the name `programname` and the optional argument list in the background.
	Example:
	`my-shell$: ./primes 4`
	`my-shell$`

`programname [args]> temp`	This command must execute the program with the name `programname` and the optional argument list redirecting the output to the file, temp.
	Example:
	`my-shell$: ./primes 4 > temp`
	`my-shell$`

Finally, your shell must handle the `SIGINT` and `SIGTSTP` signals in the following manner. If a program is executing in the foreground, `SIGINT` must stop execution of the process. If a program is executing in the foreground, `SIGTSTP` must suspend execution of the process

PROJECT PHASES AND DESCRIPTIONS

Phase #	Task	Due Date	Points
01	Command Line Parsing	Oct 29	32
02	Process Execution	Nov 7	16
03	I/O Redirection	Nov 14	16
04	Background Processing	Nov 26	16
05	Signal Handling and the Resume Function	Dec 5	16

06	Project Evaluation (Individual)	Dec 12	4
	Total		100

PHASE 01 – COMMAND LINE PARSING

Make certain you understand the assignment before you begin coding. As you can see by the point value, and dues date, this phase will probably take the longest to complete.

Your main routine should consist of an infinite loop that does nothing but print the shell prompt: (`my-shell$:`), read and parse a line of input and display the parsed data one token per line. (You will remove this display in a later phase.) You may store your parsed data in any manner you choose. You must code one or two subroutines to handle reading and parsing the data, rather than coding this functionality directly in main. Of course, you will need to code the functionality to exit your shell when your user inputs `exit`.

A token is a string of characters separated by white space.

You can assume that the `<` and `>` operators, when present, are used for I/O redirection and that, the token immediately following either of these symbols is a file name. (Keep in mind that other shells do not require a space between the redirection operator and the file name and yours should not either.)

To simplify parsing, your shell does NOT need to support quoted strings such as the following.

```
Echo "Here I am"
```

PHASE 02 – PROCESS EXECUTION

In this phase, you must modify your main loop to execute user programs, including programs such as ls, found in /bin. You will need to use the `fork()` and `exec()` functions to do this. (Review the `execvp()` and `execlp()` functions as these functions will search your $PATH for an executable.) While your user process is executing, your shell process will need to wait for the process to complete by calling the `wait()` function. In addition, your shell should check the exit code returned by the process and print an error message when the return value is not zero. At the end of this phase, you should be able to execute your code, such as `primes` and shell commands such as `ls`, both with and without arguments. (Note, it is here that you should remove your parsed data display from phase 01.)

PHASE 03 – I/O REDIRECTION

In this phase, you will add I/O redirection to your shell. To do this, you will need to open the input and/or output file(s) specified on your user command line. You should do this by closing the appropriate input and/or output file(s) and then using the `open()` system call to replace stdin and/or stdout, for the child process, as requested before executing `exec()`.

PHASE 04 – BACKGROUND PROCESSING

In phase 04, you will add background processing functionality to your shell. With other shells, you add an ampersand at the end of the command line to execute a process in the background. Your shell should replicate this functionality. Executing a job in the background is tricky. You will want your background job to keep executing even if you exit the shell. To do this you will need to daemonize your background process. For an explanation of the steps involved in daemonizing a process, see the following link: http://www.netzmafia.de/skripten/unix/linux-daemon-howto.html.

PHASE 05 – SIGNAL HANDLING AND THE RESUME FUNCTION

In this phase, you will add support for signals by coding signal handlers for the `SIGINT` and `SIGTSTP` signals. The system generates the `SIGINT` signal whenever you press control-c from the keyboard and the `SIGTSTP` signal whenever you press control-z from the keyboard. We normally use control-c to halt execution of a program and control-z to suspend execution of a program. You will need to catch both of these signals in your shell and pass the command on to the process executing in the foreground. You cannot suspend or halt execution of a background process in this manner. (In short, control-c should stop execution of the process your user executed from the shell, not the shell itself and control-z should suspend execution of the process your user executed from the shell, not the shell itself.)

Once your user has suspended a process, he or she may wish to resume that process. To resume a suspended process your user must enter `resume`. Once your user has entered `resume`, you must send the `SIGCONT` signal to your stopped process.

PHASE 06 – PROJECT EVALUATION (INDIVIDUAL)

Please write two or three grammatically correct paragraphs describing your experience with this project. Consider the following questions when writing your evaluation. What did you like about the project? What did you dislike? What problems did you encounter while coding your shell? What did you learn? What would you do differently if you were to code this project again?

GRADING – PHASES 01 THROUGH 05

Topic	Poor (1 point)	Fair (2 points)	Good (3 points)	Excellent (4 points)
Design	Team design has numerous flaws and needs significant reworking.	Team design has some flaws. Design will need some reworking to move forward.	Team design has few flaws. Design should work with minimal modification.	Team design is solid. Design should work, if implemented.
Coding	Implementation was less than 25% complete.	Implementation is between 25% and 50% complete.	Implementation is between 50% and 75% complete.	Implementation is between 75% and 100% complete.
Testing	Greater than 50% of tests resulted in erroneous output.	Between 25% and 50% of tests resulted in erroneous output.	Fewer than 25% of tests resulted in erroneous output.	No obvious errors found when testing.
Readability	Code is poorly organized and difficult to read. Team did not follow many coding standards.	Code is readable only by someone who knows what it is supposed to do. Team ignored some coding standards.	Code is fairly easy to read. Team attempted to follow coding standards.	Code is exceptionally well organized and easy to follow. Team attempted to follow all coding standards.

I will double points awarded for phase 01.