COP 4600 – Fall 2011

Project 1 – Creating a Shell

I. Project Objectives

In this project, you will design a simple command line shell that satisfies the following criteria and implement it on the specified UNIX platform.

Due Date and Submission Instructions: The due date for Project 1 is October 20th, 2011 at 11:59pm on Blackboard. You must submit all of your *.c and *.h files, as well as a makefile and a readme.txt file, which includes basic instructions on compiling your code and the basic functionality of your code. No executable program should be included. The project ***must*** compile on the c4labpc machines. If your submitted code does not compile on c4labpc11.cse.usf.edu-c4labpc29.cse.usf.edu, you will not receive any credit for any portion of the grade that requires compiled code. There are no exceptions to this rule.

Extra Credit: For each part 1-5, you may submit code that satisfies the specified functionality by the listed due date to receive 5% extra credit. We will check the code for functionality, and if it works, you will receive 5% extra credit. If it does not work, you will receive 0% extra credit. Part 6 is due at the same time as the overall project, so no extra credit is possible for Part 6. The due dates for each portion are listed below.

Optional due dates for project components:

Part	Due Date
Part 1 – The Operating System Shell (5% EC)	September 15th, 2011
Part 2 – Roll Your Own Shell (5% EC)	September 21nd, 2011
Part 3 – Enhancing Your Shell (5% EC)	September 28th, 2011
Part 4 – Adding fork and exec to your shell (5% EC)	October 5th, 2011
Part 5 – I/O Redirection (5% EC)	October 12th, 2011
Part 6 – Finishing Off Your Shell/Final Submission	October 19th, 2011

II. Project Requirements

Design a simple command line shell that satisfies the following criteria and implement it on the c4lab platform.

1. The shell must support the following internal commands:
   a. cd <directory> - change the current default directory to <directory>. If the <directory> argument is not present, report the current directory. If the directory does not exist an appropriate error should be reported. This command should also change the PWD environment variable.
   b. clr - clear the screen.
   c. dir <directory> - list the contents of directory <directory>
   d. environ - list all the environment strings

e. echo <comment> - display <comment> on the display followed by a new line (multiple spaces/tabs may be reduced to a single space)
f. help - display the user manual using the more filter
g. pause - pause operation of the shell until 'Enter' is pressed
h. quit - quit the shell
i. The shell environment should contain shell=<pathname>/myshell where <pathname>/myshell is the full path for the shell executable (not a hardwired path back to your directory, but the one from which it was executed)

2. All other command line input is interpreted as program invocation which should be done by the shell forking and execing the programs as its own child processes. The programs should be executed with an environment that contains the entry: parent=<pathname>/myshell where <pathname>/myshell is as described in 1.ix. above.

3. The shell must be able to take its command line input from a file. i.e. if the shell is invoked with a command line argument:

```
myshell batchfile
```

then batchfile is assumed to contain a set of command lines for the shell to process. When the end-of -file is reached, the shell should exit. Obviously, if the shell is invoked without a command line argument it solicits input from the user via a prompt on the display.

4. The shell must support I/O-redirection on either or both stdin and/or stdout. i.e. the command line:

```
programname arg1 arg2 < inputfile > outputfile
```

will execute the program programname with arguments arg1 and arg2, the stdin FILE stream replaced by inputfile and the stdout FILE stream replaced by outputfile.

* stdout redirection should also be possible for the internal commands: dir, environ, echo, & help.

* With output redirection, if the redirection character is > then the outputfile is created if it does not exist and truncated if it does. If the redirection token is >> then outputfile is created if it does not exist and appended to if it does.

5. The shell must support background execution of programs. An ampersand (&) at the end of the command line indicates that the shell should return to the command line prompt immediately after launching that program.

6. The command line prompt must contain the pathname of the current directory.

Note: you can assume that all command line arguments (including the redirection symbols, <, > & >> and the background execution symbol, &) will be delimited from other command line arguments by white space - one or more spaces and/or tabs (see the command line in 4. above).

III. Suggested Components and Optional Submissions for Extra Credit

Note: You are not required to follow these suggestions as long as you deliver the functionalities listed above. However, you might find it helpful for structuring your work.

Part 1 – The Operating System Shell

Due Date for 5% Extra Credit – September 15th, 2011 at 11:59PM

The first project will be for you to write your own shell to replace the standard Operating System shell.

The Shell or Command Line Interpreter (CLI) that is your environment when you logon to a UNIX system is usually the TENEX/TOPS/Thomas C Shell (tcsh), a completely compatible version of the Berkeley UNIX C shell, csh. It is a command language interpreter usable both as an interactive login shell and a shell script command processor. It includes a command-line editor, programmable word completion, spelling correction, a history mechanism, job control, i/o redirection, and a C-like syntax (see http://www.tcsh.org/Home for the full description of the ongoing development of tcsh).

Write a small program, sleepy, that gets a loop count from the command line:

```
sleepy n
```

where n is the number of seconds for which the program should run. Implement this timing by putting a loop n times of sleep(1) - this will put the program to sleep for one second n times before exiting.

```
#include <unistd.h>
...
unsigned int sleep(unsigned int seconds);
```

This function causes the calling process to be suspended until either

1. the amount of wall clock time specified by seconds has elapsed, or
2. a signal is caught by the process and the signal handler returns.

The return value from sleep is 0 or the number of unslept seconds if the sleep has been interrupted by a signal interrupt.

In each loop print out the process ID and the loop count so that that particular process can be identified.

The process ID can be obtained from the getpid function:

```
#include <sys/types.h>
#include <unistd.h>
...
pid_t getpid(void);
```

This function returns the process ID of the calling process.

The process ID is returned as a type pid_t which is actually an integer so it can be treated as such in a printf format statement (use an int cast to avoid a compile warning from gcc).

Use this program to investigate the process control of the shell provided on your UNIX system.

In particular learn how to:

1. list all current active processes
2. suspend an active process
3. put an active process into background mode
4. put an active process into foreground mode
5. also remind yourself about i/o redirection and piping

Part 2 – Roll Your Own Shell

Due Date for 5% Extra Credit – September 22th, 2011 at 11:59PM

Write a small program that loops reading a line from standard input and checks the first word of the input line. If the first word is one of the following internal commands (or aliases) perform the designated task. Otherwise use the standard ANSI C system function to execute the line through the default system shell.

Internal Commands/Aliases:

clr
clear the screen using the system function clear: system("clear").

dir <directory>
list the current directory contents (ls -al <directory>) - you will need to provide some command line parsing capability to extract the target directory for listing . Once you have built the replacement command line, use the system function to execute it.

environ
list all the environment strings - the environment strings can be accessed from within a program by specifying the POSIX compliant environment list:

```
extern char **environ;
```

as a global variable. environ is an array of pointers to the environment strings terminated with a NULL pointer.

quit
quit from the program with a zero return value. Use the standard exit function.

You might want to design your program to make this list of 'aliases' extendable by storing the alias strings and the alias functions in arrays...

External Commands:

For all other command line inputs, relay the command line to the parent shell for execution using the system function.

When parsing the command line you may have to explicitly or implicitly malloc (strdup) storage for a copy of the command line. Ensure that you free any malloced memory after it is no longer needed. You may find strtok useful for parsing.

The C Standard Library has a number of other string related functions that you may find useful (string.h)contains links to descriptions of the other main "string" functions).

Part 3 – Enhancing Your Shell

Due Date for 5% Extra Credit – September 29th, 2011 at 11:59PM

Environment & Command Line Arguments

Basic information can be passed to a C process via:

* Command Line Parameters
    - argv - an array of character pointers (strings) 2nd argument to main the command line parameters include the entire parsed command line (including the program name).
    - argc - number of command line parameters
* Environment
    - environ - an array of character pointers (strings) in the format "name=value"
      the list is terminated with a NULL pointer.

Command Line Parameters

```
int main(int argc, char * argv[])
{
   int i;


   printf("argc = %d\n");                  // print arg count
   for (i = 0; i < argc; i++)              // print args
      printf("argv[%d]: %s\n",i, argv[i]);
   exit(0);
}
```

invoked through the command line:

```
echoarg arg1 TEST fooo
```

will result in the following output:

```
argc = 4
argv[0]: echoarg
argv[1]: arg1
argv[2]: TEST
```

```
    argv[3]: fooo
```

Accessing The Environment

While some implementations of C provide a 3rd argument to main (char * envp[]) with which to pass the environment, ANSI C only defines two arguments to main so the preferred way of passing the environment is by a built-in pointer (POSIX):

```
    extern char **environ;  // NULL terminated array of char *

    main(int argc, char *argv[])
    {
       int i;
       for (i = 0; environ[i] != NULL; i++)
          printf("%s\n",environ[i]);
       exit(0);
    }
```

example output:

```
    GROUP=staff
    HOME=/usr/user
    LOGNAME=user
    PATH=/bin:/usr/bin:usr/user/bin
    PWD=/usr/user/work
    SHELL=/bin/tcsh
    USER=user
```

Updating The Environment

You can use the functions getenv, putenv, setenv to access and/or change individual environment variables.

Internal storage is the exclusive property of the process and may be modified freely, but there is not necessarily room for new variables or larger values.

If the pointer environ is made to point to a new environment space it will be passed on to any programs subsequently invoked.

If copying the environment, you should find out how many entries are in the environ table and malloc enough space to hold that table and any extra string pointers you may want to put in it.

Remember, the environment is an array of pointers. The strings pointed to by those pointers need to exist in their own malloced memory space.

Extending Your Shell

Try adding the extra functionality listed below.

Internal Commands/Aliases:

Add the capability to change the current directory and set and change environment strings:

```
    cd <directory>
```

Change the current default directory to <directory>. If the <directory> argument is not present, report the current directory. This command should also change the PWD environment string. For this you will need to study the chdir (Glass p 431), getcwd and putenv functions.

While you are at it you might as well put the name of the current working directory in the shell prompt!

Be careful using putenv - the string you provide becomes part of the environment and should not be changed (or freed!). i.e. it should be made static - global or malloced (or strduped).

Part 4 – Adding fork and exec to your shell

Due Date for 5% Extra Credit – October 6th, 2011 at 11:59PM

So far, the shell has used the system call to pass on command lines to the default system shell for execution. Since we need to control what open files and file descriptors are passed to these processes (i/o redirection), we need more control over there execution.

To do this we need to use the fork and exec system calls. fork creates a new process that is a clone of the existing one by just copying the existing one. The only thing that is different is that the new process has a new process ID and the return from the fork call is different in the two processes.

The exec system call reinitializes that process from a designated program; the program changes while the process remains!

Make sure you read the notes on fork and exec and understand how and why fork and exec work before continuing.

1. In your Shell program, replace the use of system with fork and exec.

2. You will now need to more fully parse the incoming command line so that you can set up the argument array (char *argv[] in the above examples). N.B. remember to malloc/strdup and to free memory you no longer need!

3. You will find that while a system function call only returns after the program has finished, the use of fork means that two processes are now running in foreground. In most cases you will not want your shell to ask for the next command until the child process has finished. This can be accomplished using the wait or waitpid functions. e.g.

```
switch (pid = fork ()) {
   case -1:
      syserr("fork");
   case 0:                  // child
      execvp (args[0], args);
      syserr("exec");
   default:                 // parent
      if (!dont_wait)
         waitpid(pid, &status, WUNTRACED);
 }
```

Obviously, in the above example, if you wanted to run the child process 'in background', the flag dont_wait would be set and the shell would not wait for the child process to terminate.

4. The commenting in the above examples is minimal. In the projects you will be expected to provide more descriptive commentry!

Part 5 – I/O Redirection

Due Date for 5% Extra Credit – October 13th, 2011 at 11:59PM

Your project shell must support i/o-redirection on both stdin and stdout. i.e. the command line:

```
programname arg1 arg2 < inputfile > outputfile
```

will execute the program programname with arguments arg1 and arg2, the stdin FILE stream replaced by inputfile and the stdout FILE stream replaced by outputfile.

With output redirection, if the redirection character is > then the outputfile is created if it does not exist and truncated if it does. If the redirection token is >> then outputfile is created if it does not exist and appended to if it does.

Note: you can assume that the redirection symbols, < , > & >> will be delimited from other command line arguments by white space - one or more spaces and/or tabs. This condition and the meanings for the redirection symbols outlined above and in the project differ from that for the standard shell.

I/O redirection is accomplished in the child process immediately after the fork and before the exec command. At this point, the child has inherited all the filehandles of its parent and still has access to a copy of the parent memory. Thus, it will know if redirection is to be performed, and, if it does change the stdin and/or stdout file streams, this will only effect the child not the parent.

You can use open to create file descriptors for inputfile and/or outputfile and then use dup or dup2 to replace either the stdin descriptor (STDIN_FILENO from unistd.h) or the stdout descriptor (STDOUT_FILENO from unistd.h).

However, the easiest way to do this is to use freopen. This function is one of the three functions you can use to open a standard I/O stream.

```
#include <stdio.h>

FILE *fopen(const char *pathname, const char * type);

FILE *freopen(const char * pathname, const char * type, FILE *fp);

FILE *fdopen(int filedes, const char * type);

All three return: file pointer if OK, NULL on error
```

The differences in these three functions are as follows:

1. fopen opens a specified file.
2. freopen opens a specified file on a specified stream, closing the stream first if it is already open. This function is typically used to open a specified file as one of the predefined streams, stdin, stdout, or stderr.
3. fdopen takes an existing file descriptor (obtained from open, etc) and associates a standard I/O stream with that descriptor - useful for associating pipes etc with an I/O stream.

The type string is the standard open argument:

type	Description
r or rb	open for reading
w or wb	truncate to 0 length or create for writing
a or ab	append; open for writing at end of file, or create for writing
r+ or r+b or rb+	open for reading and writing
w+ or w+b or wb+	truncate to 0 length or create for reading and writing
a+ or a+b or ab+	open or create for reading and writing at end of file

where b as part of type allows the standard I/O system to differentiate between a text file and a binary file.

Thus:

```
freopen("inputfile", "r", stdin);
```

would open the file inputfile and use it to replace the standard input stream, stdin.

You may want to use the access function to check on existence or not of the files:

```
#include <unistd.h>

int access(const char *pathname, int mode);
```

Returns: 0 if OK, -1 on error

The mode is the bitwise OR of any of the constants below:

mode	Description
R_OK	test for read permission
W_OK	test for write permission
X_OK	test for execute permission
F_OK	test for existence of file

Part 6 – Finishing Off Your Shell/Final Submission

Due Date for Final Project – October 20th, 2011 at 11:59PM

This week you will be putting the finishing touches to your shell:
Script file support

Make the shell capable of taking its command line input from a file whose name is provided as a command line argument. i.e. if the shell is invoked with a command line argument:

```
myshell batchfile
```

then batchfile is assumed to contain a set of command lines for the shell to process. When the end-of-file is reached, the shell should exit. When a batch file is provided, the shell should provide no prompt.

Obviously, if the shell is invoked without a command line argument it should solicit input from the user via a prompt on the display as before.

You will probably need to have a look at the difference between the gets and fgets functions and insert a test for end-of-file for when reading from a file (int feof(FILE*)). You may find it easier to use fgets for both types of input using the default FILE stream pointer stdin when no batch file is specified.

Internal Commands/Aliases:

To provide support for batch file operation, add the following internal commands:

echo <comment>
display this command line on the display (without the word echo). Remember that <comment> can be multiple words.
pause
display "Press Enter to continue..." and pause operation of the shell until 'Enter' is pressed (ignore any intervening non-'Enter' input). Turning off echo of keyboard input is possible and there is also a system function to read a buffer from the current tty input without echo - this is left as a research exercise for the student!

To comply with the project specifications you also need to provide a help command that will print out your readme file. Note that you may not actually still be in the 'startup' directory! Although you can assume that the readme file will be in the current working directory when the shell is started.

Background Execution of External Commands:

Having put in the wait function to ensure that the child process finishes before the shell gets its next command line, you will now have to provide the option of 'background' operation. i.e. the child is started and then control by-passes the wait function to be immediately passed back to the shell to solicit the next command line.

An ampersand (&) at the end of the command line indicates that the shell should return to the command line prompt immediately after launching that program. You can assume that & will be surrounded by white space (' ', '\t', or '\n') and will be the last non-white space character in the line.

makefile Support

Marking the project will entail the marker recompiling your source code from scratch. To automate this process, a requirement of the project is that your shell must be able to be built using a make file. If you are using two source files myshell.c and utility.c, both of which reference the local 'include' file myshell.h, then your makefile should look like this:

```
# Joe Citizen, s1234567 - Operating Systems Project 1
# CompLab1/01 tutor: Fred Bloggs
myshell: myshell.c utility.c myshell.h
	gcc -Wall myshell.c utility.c -o myshell
```

Then your shell program is re-built by just typing make at the command prompt if any of the .c or .h files have been changed since the last time the shell was built.

Note that this makefile names the resulting executable file as myshell, another requirement of the project.

Note also that the third line of the above makefile - gcc -Wall myshell... - the action line - must start with a tab.

Polishing:

1. Check that arguments exist before accessing them.
2. What happens when Ctrl-C is pressed during
    1. command execution?
    2. user entry of commands?
    3. would SIGINT trapping control this better?
3. Check existence of input redirection files
4. Check makefile works
5. Ensure that all files are named as requested

IV. Late Assignments

Per the syllabus, for each day a programming project is late, the grade you receive is reduced by 15%. For example, if you earned a 95% and submitted your assignment one day late, your grade will become an 80%.

V. Academic Honesty

You must work on this assignment by yourself. In Proj0, you got to experience working with MOSS. We will be checking your submitted assignment against all of your classmates to protect the integrity of your grade. Therefore, cheating will be treated according to University policies. If you are caught cheating on this assignment, you will receive an FF grade for the class. Material copied from the Web and submitted as your work *is cheating*.

VI. Advice

* Many of you found out the hard way that working on the code on your favorite C compiler, then transferring it to the C4 Lab machines will cause you considerable headache and wasted time.

* Do try every extra credit. This is done to reward people for good time management! Also, even if you get 0% on an extra credit submission, it is beneficial since you will become aware of errors in your code, which you'll be able to fix, and will improve your project code (and project grade!).

* Ask for help early and often! Our office hours and e-mails are listed on the syllabus.

* E-mail the professor or TAs immediately if you find yourself in a situation in which you are unsure of whether or not something you are doing constitutes academic dishonesty.

XII. References

* Refer to Dr. William Stalling's write up on this project (attached with this project as Project-Shell.pdf), which provides an excellent overview of how to construct and test your work.