CSC-565 Operating Systems
Spring, 2008
UNIX Shell Project

Due Tuesday, February 12

Objective

The shell is the part of the operating system with which a computer user actually interacts. It provides an interface for the user to run programs and to manage files. Early shells were nearly all command line-based; at the prompt, the user types in commands or names of programs to be executed. In this project, you will write a simple command-line shell that will run on a UNIX / Linux system. Your shell should allow the user to start programs and perform several simple file manipulation tasks.

In this project you will learn to program in C and become comfortable with the UNIX command line.

Implementation

You must write your shell in the C language. To complete this project you will need to use both standard C functions and POSIX system calls. You should not make use of any existing shell; in other words, you should not do any form of scripting. This project requires that you have access to a UNIX or Linux account - if you do not have a CAP Linux account already you should contact your professor.

This project should be an individual effort. You are permitted to discuss your project with others at the conceptual level, but all your code should be your own.

Project components

Your basic shell should prompt the user to enter a command, read in a text string from the user, parse the text string to determine a command, execute the command, and repeat these steps forever. Your shell should be able to recognize incorrect commands and give the appropriate error message.

Your shell should be able to execute a variety of commands. Your grade will depend on which commands you successfully implement.

<u>The following is required to earn a minimum of a **C** (75/100):</u>

- Your shell should prompt the user, read in a command string, execute the command, and repeat the process.
- Your shell should include the command “exit”, which should terminate your shell.

- Your shell should include the command "type <filename>", which should print out the contents of <filename> to the screen. For example, if the user types **type test.c** at the command line, the contents of test.c should be printed out.
- Your shell should include the command "copy <filename1> <filename2>" which copies <filename1> to <filename2>. The shell should create a file <filename2>, open <filename1>, and copy the contents of <filename1> byte-by-byte to <filename2>.
- If the user types a command other than exit, type, or copy, your shell should give an error message and prompt again. If the user tries to "type" a file that doesn't exist, or "copy" a file that doesn't exist, your shell should give an error message and prompt again. In other words, it should be reasonably difficult to crash your shell.

<u>The following is required to earn a minimum of a **B** (85/100):</u>

- You should complete all the above components, plus the following:
- Your shell should be able to execute programs. If the user types a command that is not "type","exit", or "copy", the shell should assume that the command is the name of a program located in the same directory, and should attempt to run that program. You should not worry about passing command line arguments to the program. If the program does not exist or is not an executable, your shell should give an error message and prompt again.
- Your shell should include the "delete <filename>" command, which should delete the file <filename>. For this you should use the **remove** system call.

Some comments on executing programs:

The **execvp** system call is used to load a program into memory and execute it. , It has the syntax: "execvp(filename, args)". filename is a character array (string) consisting of the name of the program to be executed. args is an array of strings consisting of the command line arguments to be passed to the program. If you are not passing any command line arguments, you should define an empty array of strings and pass that as args. This may be done easily:

```
char args[1][1];
args[0][0]='\0';
```

One challenge you will face is that if you use **execvp** to run the program, when the program finishes it will not go back to your shell; instead your shell will terminate. To get around this, you should create a new process, use that process to execute the program, and keep your shell in the background to take over when the program finishes. This may be done

with the **fork** system call, which has the syntax "int pid=fork()" Once you call **fork**, two processes will execute the subsequent code. One process will have the number 0 in pid, the other will have a different number in pid. The process with the number 0 in pid should call **execvp**, the other should return to the prompt.

The following is required to earn a minimum of an **A** (95/100):

- Once you finish all this, you will notice that the above method of executing a program has a problem: your shell will prompt again before the program you called finishes running. You should improve your shell so that the process that does not call **execvp** waits until the program has finished before it prompts again. This can be done with the **wait** system call, which suspends a process until another process terminates.

The following is required to earn an **A+** (130/100):

- Execute ELF executable files without using execvp. You should decode the ELF file format produced by gcc.

Executing a file is a matter of loading the program file into memory and calling it. Executable files have a file header that tells how to interpret the file.

All executable files have two components that must be initialized: the machine code instructions (called perversely .text), and the global data variables (called .data). The header tells where to put the .text and .data into memory.

To do this part, you will need to perform four steps:

1. Read and decipher the ELF header at the beginning of the file. It tells where to find the section headers.
2. Read and decipher the section headers. There is a header for each .text and .data section. The header tells where the section is located in the file, and where to put it in memory.
3. Allocate the memory that the headers are asking for to a character array. Use the **mmap** instruction. It will return 0 if it refuses to allocate.
4. Copy the sections to the appropriate places in the character array.
5. The entry point to the program is given in the ELF header. Jump to it. This is a matter of stating:

```
void (*func)();
func=(void*)entry_point;
(*func)();
```

To complete this part, you will need to learn how structs work in C. The structs for decoding the ELF header and section headers are provided in "elf.h". You should look online for guides on decoding the header.

Important note:

This shell project is a fairly common operating system project, and I am aware of C code available on the internet that duplicates much of what I am asking you to do. You will almost certainly need to make use of books and websites in order to get the POSIX calls working. However, please do not copy any online code into your project. If you make any extensive use of a book or website, please cite it in your submission.

Submission:

You should submit the following deliverables to me by Blackboard Digital Dropbox at or before 11:59 pm on the date due:

- Your source code, well commented
- Your executable file
- A README file consisting of the following:
    - a step-by-step description of how to compile your source code and run your shell
    - an explanation of how to use your shell
    - a list of what commands your shell contains and whether they work correctly
    - citations of any resources you used in completing your project
- This should all be compressed into a single .zip or .tar file (no .rar please) with your name as the filename.

Your code must compile without errors and run for any credit to be given for your project.