

Project 1: Implementing a Shell

Eric Cronin & Gaurav Shah & Micah Sherr*
University of Pennsylvania
CIS381 Fall 2007

September 12, 2007

1 Overview

In this *individual*¹ project you will have to design and implement a simple shell command interpreter, called *mysh*. The basic function of a shell is to accept lines of text as input and execute programs in response. While a shell can have built-in commands, it must be able to execute programs existing as independent files in the system. These programs must be executed in a process different from the one executing *mysh*.

Tcsh, *bash* and *ksh* are examples of widely used real shells. You can check out the respective manpages to get an idea of what sort of features they support. These shells are large and have support for features like scripting, commandline completion, wildcard expansion, command history etc... While these features are neat, implementing them would not necessarily increase your understanding of Operating System concepts, thus we do not ask you to do so. We are much more interested in the job control features of the shell. While working on this project you will get familiar with the semantics of the system calls that control processes, in particular their creation, destruction, hierarchy, and file I/O.

2 Requirements

When first started, your shell should initialize any necessary datastructures and then enter a loop of printing a prompt, e.g. “mysh# ”, reading any commands entered by the user and executing them.

2.1 Simple execution of commands

The simplest case is when *mysh* has to execute a program in the foreground, wait for it to terminate, and then resume control, e.g.

```
mysh# vi
```

would launch the *vi* editor, the user can then perform text processing and after the user quits *vi*, control is returned back to the shell. The user shouldn't have to type in the full path to executables as long as they are within the search path specified in the environment by the *PATH* variable. In your initial implementation *mysh* should spawn a child process and block wait for the child to terminate. The relevant system calls here are *fork()*, the *wait()* and *exec()* families of calls, and *exit()*.

*Based on previous projects authored by: Stefan Miltchev

¹You are allowed to discuss the assignment with your friends, but you should do all coding on your own.

2.2 Execution of commands in the background

The next step of complexity in your implementation will be handling execution of commands in the background, e.g.

```
mysh# xemacs project1.tex &
```

would run the *xemacs* editor in the background. When a process is executing in the background, the shell should not block and wait but rather continue accepting commands. A shell with forked child processes executing in the background must be able to tell when the child processes exit. It can do this either synchronously or asynchronously.

In the synchronous approach, the shell periodically polls for child termination. You can do this by using some of the *wait()* system calls with the *WNOHANG* flag. This approach is easier to implement, and you might want to start with it, but for the final version of *mysh* you will have to use asynchronous notification of job termination. The problem with polling is figuring out at what intervals to poll. If the intervals are too long, it might take too long to detect that an event has taken place. Conversely, if the polling frequency is increased, the overhead and program complexity are going to increase.

In the asynchronous approach, the shell would receive a signal from the OS as soon as the child exits. You should consult the *signal()* manpages in section 3 to familiarize yourself with how signals are used². The signal that the OS sends to notify a parent process that a child has changed states (exited, killed, stopped) is the *SIGCHLD* signal. To receive a signal, the process must tell the OS that it is interested in receiving it by using the *sigaction()* system call. This call registers a signal handler, that will be triggered when the specified asynchronous signal arrives.

In both the synchronous and asynchronous cases, your shell must notify the user as soon as it detects any change in status for a job (running→done, running→stopped, etc). Do not worry about making this notification unobtrusive (e.g. redrawing the command prompt afterwards).

Using signals you have to be careful to deal with any concurrency issues that might potentially arise, e.g. bad things could happen if your shell is updating a list of jobs in response to user input and at the same time the signal handler for child process termination also needs to update the list. You will have to identify critical sections of code where concurrent access could be disastrous. You could delay the signal handler until the program is past a critical section by masking signals. Consult the *sigprocmask()* manpage. Other synchronization mechanisms, such as *pthread()* are not, in general, safe with respect to signals, and should not be used.

Suspending a process group by typing Control-Z should also be implemented using signals. The signal sent to the foreground process is *SIGTSTP*. A process stopped by *SIGTSTP* will generate a *SIGCHLD* signal to your shell; your shell should then send a *SIGSTOP* signal to all the process(es) executing the current foreground process group (in case any are ignoring *SIGTSTP*, and return with a new prompt. To continue a stopped job, the shell must send a *SIGCONT* signal.

You will have to make sure that processes launched by your shell are in appropriate process groups. Note that more than one process can be started from the same commandline. Each group of processes launched on the same commandline and attached with pipes should be in a distinct process group. If you don't do this and the spawned processes are in the same process group as the shell, they will receive signals along with the shell, e.g. the terminal driver sends a *SIGTSTP* to all processes in the foreground process group. To set the process group of a process use the *setpgid()* system call. To ensure that the currently running foreground process's process group is the foreground process group use the *tcsetpgrp()* system call.

²When referring to manpage sections we are using the Linux layout. Other UNIX dialects might have slightly different layouts, e.g. Solaris has separate *3HEAD* and *3C* sections

Project 1 — CIS 381, Fall 2007

You will also have to use signals to implement the *kill* command described later in this handout.

The signal handling part of your shell implementation is the trickiest so do not start on it before having completed the easier parts.

2.3 Redirection of input and output

Mysh should also support input and output redirection, e.g.

```
mysh# w > users.txt
```

would run the *w* command that displays information about currently logged-in users and save its output in a file called *users.txt*. Similarly, if we wanted to know the number of lines in the file we just created:

```
mysh# wc -l < users.txt
```

would make the file *users.txt* appear as standard input to the *wc* program. To implement input/output redirection of child processes you will have to manipulate file descriptors using the *dup2()* system call. For example, to connect a file to standard input:

```
if (( fd = open (fname, O_RDONLY ) ) == -1 )
{
    fprintf(stderr, "Error: couldn't open %s\n.", fname);
    exit(1);
}

dup2(fd, 0);
close(fd);
```

Be careful as to the distinction between parent and child when you use this.

2.4 Pipes

In addition to sending standard output to a file and/or reading standard input from a file, *mysh* should also support "pipes", i.e. the ability of sending standard output of one program as standard input of another, e.g.:

```
mysh# w | sort
```

would send the output of the *w* program as input to the *sort* program and the result would be a list of currently logged-in users in alphabetical order. Note that pipes can be chained (e.g. *w|sort|more*) and/or combined with input/output redirection (e.g. *w|sort > sorted_users.txt*). The next section includes a grammar that should cover all possibilities.

To implement pipes, you could use the *pipe()* system call. You will have to study the manpage to get all the gory details, e.g. how to generate EOF. Together with signal handling this is one of the trickier parts of your implementation and you should not attempt it before you have finished the easier tasks.

2.5 Parsing the commandline

When parsing the commandline, *mysh* should ignore all whitespace. You might find the *isspace* and *strtok* man pages useful, as well as the provided string tokenizer discussed in Section 3 (which already implements most of the parsing requirements). Please keep in mind that your commandline parser should work, but we do not expect it to be particularly fancy (hopefully you will take or have taken other courses that address this topic). For example, " and ' should not be treated specially by the parser. We'd much rather see you spend a lot of effort on the job control features of your shell. If you are familiar with *lex* and/or *yacc* you can use them, but you certainly don't have to. Below is the *mysh* commandline grammar:

```
CommandLine      :=  NULL
                   FgCommandLine
                   FgCommandLine &

FgCommandLine    :=  SimpleCommand
                   FirstCommand MidCommand LastCommand

SimpleCommand    :=  ProgInvocation InputRedirect OutputRedirect

FirstCommand     :=  ProgInvocation InputRedirect

MidCommand       :=  NULL
                   | ProgInvocation MidCommand

LastCommand      :=  | ProgInvocation OutputRedirect

ProgInvocation   :=  ExecFile Args

InputRedirect    :=  NULL
                   < STRING

OutputRedirect   :=  NULL
                   > STRING
                   >> STRING

ExecFile         :=  STRING

Args             :=  NULL
                   STRING Args
```

2.6 Built-in commands

Mysh should support the following built-in commands. Exec'ing binary files such as */bin/kill* is not permitted. You may assume that these built-in commands will always be run in the foreground as a single process job with no redirection, and generate an error message if this is not the case.

- **exit**: Exits *mysh* and returns to whatever shell you started *mysh* from.

Project 1 — CIS 381, Fall 2007

- **jobs**: Print a list of commands that are currently running or suspended. This list should include an integer job ID, the commandline that started the respective program, and the status (Running or Suspended) of the job, e.g.

```
mysh# jobs
[1]      Running          xemacs project1.tex
[2]      Running          xdvi project1
[3]      Suspended        vi
[4]      Suspended        firefox
```

- **kill %job**: Sends a *SIGTERM* signal to the specified job ID, e.g.

```
mysh# kill %2
```

would terminate the *xdvi* program from the previous example.

- **fg %job**: Continue executing the specified command in the foreground, e.g.

```
mysh# fg %3
```

would bring the *vi* program to the foreground. The shell would then wait for the user to either terminate the *vi* program or to suspend it using Control-Z.

- **bg %job**: Put the specified job in the background, continuing it if it was suspended, e.g. if a user had started the *firefox* web browser, and suspended it using Control-Z, then

```
mysh# bg %4
```

should let the browser continue running in the background.

- **pwd**: Print the current working directory.
- **cd path**: Change the current working directory to specified path.

3 Provided Files

To give you a running start, we have provided an example Makefile and string parser. The code is available at

<http://www.seas.upenn.edu/~cse381/cool-shell.tar.gz>

You can use the *wget* utility (see the manpage for help) to download this file from the *eniacy-l* prompt. To extract the contents of this tarball, execute “*tar -xzf cool-shell.tar.gz*”. More information concerning the string tokenizer can be found in the accompanied README file.

You are obviously under no obligation to use the provided code (you are of course also welcome to adapt it to your needs). Regardless, you may want to briefly look it over, as it contains the same level of commenting that we expect in your submissions.

4 Hints and Tips

- The man pages are your friends! Manpages of particular interest would be:
 - fork(2)
 - exit(2)
 - exec(2), execl(2), execv(2), execl(2), execve(2), execlp(2), execvp(2)
 - wait(2), waitpid(2), pause(2), sigpause(3C)
 - dup2(3C), pipe(2)
 - signal(3), signal(3), sigaction(2), sigprocmask(2)
 - setpgid(2), tcsetpgrp(3), tcgetpgrp(3)
 - kill(2)
 - isspace(3), strtok(3)
 - getcwd(3), chdir(2)

Depending on your implementation, you might have to use fewer or more calls. Note that using *system(3)* instead of the system calls is not acceptable. However, it is OK to use the standard C library functions (e.g. *scanf(3)*) for your parser instead of using *read(2)* or *write(2)* directly.

- You will have a much easier time if you divide and conquer, i.e. develop your implementation in incremental stages, starting with the easier features and only adding the more advanced ones as you progress. Here is a suggested order of development stages:
 1. Write a shell that can fork single program invocations and wait for their completion.
 2. Develop the commandline parser.
 3. Add support for running programs in the background ("&"). At this stage you can use synchronous notification of job termination via polling.
 4. Add support for the *jobs* command.
 5. Add support for I/O redirection.
 6. Add support for pipes.
 7. Add asynchronous notification of background termination through the use of signals.
 8. Add support for job control features: Control-Z, *fg* and *bg*.
- Utilize unit testing. Part of the documentation requirements are to explain how you tested each feature for correctness. These tests should check both that your implementation does the right things with good input, and that it does not do bad things with bad input.
- This project should be developed on a UNIX OS. You are free to use any flavor of UNIX (e.g. Solaris, Linux, FreeBSD, OpenBSD) you want for development. However, you should make sure that your code compiles and executes on the *eniach-l* cluster machines before turning in your project.
- Good coding practices and documentation are essential.
- This is not a trivial program. Start *early*! If you wait until the last minute you will not be able to finish.

5 Attribution

This is a large and complex project, using arcane and compactly documented APIs. We do not expect you to be able to complete it without relying on some outside references. That said, we *are* still sadistic monsters, and want to watch you struggle some and really learn something about systems programming from this assignment.

The primary rule to keep you safe from plagiarism/cheating in this project is to attribute in your documentation any outside sources you use. This includes both resources used to help you understand the concepts, and resources containing sample code you incorporated in your shell. The course text, APUE, need only be cited in the later case. You should also use external code sparingly. Using most of the example from the *pipe(2)* man page is ok; using the *ParseInputAndCreateJobAndHandleSignals()* function from *Foo's Handy Write Your Own Shell Tutorial* is not (both verbatim, and as a closely followed template on structuring your shell).

6 Approximate Grading Guidelines

The approximate grading guidelines are as follows:

- Documentation - 25%
- Implementation - 75%, further broken down as:
 - Single program execution in the foreground and builtins (exit, pwd and cd) - 10%
 - Asynchronous Job control (including job control commands) - 40% (synchronous job notification only will get you 10%)
 - I/O redirection - 15%
 - Pipes - 10%

7 Deadline and Turnin

This project is due at **6:00pm, October 10, 2007**. Since this is a longterm project no late submissions will be accepted.

Place all your files in a directory called *project1-username*. The directory should contain a *Makefile* so that we can run *make* which should produce an executable called *mysh*. You must also include a **text README** file explaining any particular design decisions or features of your implementation. It should contain at least three parts: an introduction to the basic structure of your shell; 2) a list of which features are fully implemented, partially implemented, and not implemented; and 3) a discussion of how you tested your shell for correctness. Documenting code and having an overview document is an important component of building software and will make grading easier in cases where you are unable to get something to work and we need to take a closer look at the code.

To submit your project you should use the *turnin* program on the *eniac* cluster machines. For example, if your project is contained in a subdirectory *project1-username* (as it should be), you would use:

```
turnin -c cse381 project1-username
```