CMSC 23000	Operating Systems	Project 1
Winter 2006		January 6

Unix shell

Due: January 20

1 Introduction

Despite recent bad press, an unnamed agency of the government is eager to continue the fight against enemies of the state. To this aim, they have contracted with you to write a new Unix shell called the "*Freedom shell*" (or **fsh**). This shell will allow responsible unnamed agencies to monitor the activities of suspected terrorists, environmental activists, and other "bad actors."

Your task is to write an implementation of the **fsh** using POSIX threads (pthreads). The **fsh** is a simple Unix shell that allows a third party to remotely monitor shell commands. Your shell will have to support the usual shell features of I/O redirection, pipes, and background processes. But it also has to provide this clandestine monitoring feature.

2 Description

The input to the **fsh** is a sequence of commands, each provided on a separate line of input text typed interactively at the keyboard: The **fsh** supports the following command syntax:

Command
::= *Program* (`|` *Command*)^{opt}
| *Program* `&`^{opt}
| `exit`

Program
::= *Path Args*^{opt} (`<` *Path*)^{opt} (`>` *Path*)^{opt}

where a *Path* has the following syntax:

Path
::= `/`^{opt} *Filename* (`/` *Filename*)*

Filenames are non-empty sequences of letters, digits, or one of "`.`", "`-`", "`+`", "`=`", "`@`", or "_."

2.1 Signals

Finally, your shell needs to ignore a single signal, `SIGINT`. This signal is generated when a user presses ctrl-`C` on the keyboard. When received, it should be passed to the currently running program, but it should not cause your shell to terminate. It should also have no effect on background jobs.

2.2 Background jobs

When a program runs, it normally blocks you from performing any other operations until it has completed. However, you can put a program into the background using the "**&**" operator. For example:

```
progname args &
```

Detaches the program `progname` and runs it in the background. Control is immediately returned to the command shell where additional commands can be executed. Background jobs should continue to run even if you quit the shell before they have finished.

2.3 I/O redirection

In addition to the above commands, your shell must support I/O redirection. I/O redirection is specified using the "**<**" and "**>**" operators at the end of a command line. For example, the command

```
progname args >file.out
```

directs the standard output of `progname` to the file `file.out` and

```
progname args <file.in
```

uses the contents of the file `file.in` as the standard input to program `progname`. Both input and output redirection may be specified for a single command so your shell will have to check for both.

2.4 Pipes

Your shell also needs to support pipes. A pipe is nothing more than a way of hooking up the standard output of one program to the standard input to another. A pipe is indicated using the "**|**" operator as follows:

```
progname1 args | progname2 args
```

Pipes the output of program `progname1` to the input of program `progname2`. For example:

```
fsh % ls | wc
      5       5     28
fsh % foo < infile | bar > outfile
```

Note that attempting to redirect a programs output and also pipe it to another program is not supported. For example, the following produces an error message:

```
fsh % ls > outfile | wc
fsh: illegal command
```

2.5 The "Freedom" interface

To allow secret monitoring, the shell should listen for connections on a network port that is equal to its process id (pid) + 10000. Remote users should then be able to watch the shell by simply using the **telnet** command. For example, suppose that a user launches **fsh** and it has a process ID of 11538 (you can use **ps** to find the process ID). The the following command will connect to the **fsh**:

```
% telnet localhost 21538
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
[ Welcome to the fsh shell ]
```

Once a user connects to the **fsh**, the user should see all commands and all output from those commands displayed as long as the telnet connection stays open. Commands should be prefixed with the string "`[fsh]`" and the output from a command `progname` should be prefixed with "`[progname]`." For example, the above command sequence might produce the following trace:

```
[fsh] ls | wc
[ls]bar.h\nbaz.c\nfoo\nfoo.c\nfoo.out
[wc]\t7\t5\t28\n
[fsh]foo < infile | bar > outfile
```

Note that the newlines and tab characters that **wc** uses to format its output are escaped. Note that the **fsh** should support multiple connections to the freedom interface.

3 The project

We recommend that you break the project into steps. You use your subversion repository to manage your source; tag major milestones so that you can isolate the source of problems later on.

Chapter 3 of the text has a shell project; you may find its description helpful.

3.1 Step 0: Gforge account

The first step is to create a gforge account on `cs230.cs.uchicago.edu`. Please email your account name to the TA (`jriehl@cs.uchicago.edu`). We will then create a project for you and an initial subversion repository.

3.2 Step 1: Design

The first step is to design the architecture of your implementation. For this purpose, I recommend thinking how you want to map the major components of the system onto Posix threads.

3.3 Step 2: Command interpreter

Your initial subversion repository includes a parser for the command language. This parser builds a parse-tree representation of the input. You should implement an interpreter for this representation. Start with basic command execution, then add I/O redirection, then background jobs, and finally

pipes. Once you have these features working, add support for ctl-C. Commit your changes at each step of the way.

3.4 Step 3: Freedom interface

Once you have the basic shell working, add the monitoring mechanism.

3.5 Step 4: Commit final version

Make sure that you have committed your final version. Using Doxygen, generate the documentation for your code. Make sure that it includes your name!. The documentation is due in class on Friday, January 20th.

4 Grading

Your project will be graded on both correctness (70%) and programming style (30%). The documentation is evaluated as part of the style component of your grade.