Lab 04 – The UNIX Shell (Part II)

Objectives

1. Learn about running a process in the background.
2. Learn about file redirection.
3. Learn about pipes for inter-process communication (IPC).
4. Extend the functionality of your shell program.

Pre-Reading

The following material contains commands and/or other information that could be essential to completing this and future labs. Students are strongly encouraged to take a look at the material below *before* coming to the lab, since knowing this material before you start will likely make task of completing the lab much faster and easier.

Running a Process in the Background

When you type a command and hit **`<Enter>`**, the shell executes the command and returns by displaying the shell prompt. While your command executes, you do not have access to you shell and therefore cannot execute any command until the current command finishes and the shell returns. When commands execute in this manner, we say that they execute in the foreground. More technically, when a command executes in the foreground, it keeps control of the keyboard and terminal display screen.

Say a command will take a long time to finish, and while it executes you wish to execute other commands. You cannot do this if that command runs in the foreground, because that program has control of the keyboard and hence you cannot type in commands to the shell program. UNIX allows you to run a program such that, while it executes, you get the shell prompt back and can do other work. This capability is called running the command in the background. You can run a command in the background by ending the command with the ampersand (**&**) metacharacter. For example:

```
$ command &
```

Note that a command running in the background can still write its output to the terminal screen by default (via **stdout**). A background process and the foreground process writing to the terminal screen at the same time can result in garbled text. Hence, it is recommended that you redirect the output of **stdout** for the background process to a file. If you don't want to save the output or send it to the screen, then redirect it to /dev/null. This file is often called the black hole, because any bytes written to it just… disappear. That is, no matter how many bytes you write to it, it remains empty.

File Redirection

As you may recall…

- A process reads its input from the **stdin** file.
- A process writes its output to the **stdout** file.
- A process writes its error messages to the **stderr** file.

The standard files (**stdin**, **stdout**, and **stderr**) are actually virtual files in the /proc directory, and hence the operating system controls the sequence of bytes that a command reads from **stdin**, as well as where to send the sequence of bytes that a command writes to **stdout** and **stderr**.

By default…

- The sequence of bytes read from **stdin** originates from the device file for the terminal keyboard.
- The sequence of bytes written to **stdout** and **stderr** are written to the device file for the terminal screen.

Input file redirection allows you specify the file from which the sequence of bytes read from **stdin** originate. That is, instead of a command reading its standard input stream from the terminal keyboard device file, it reads it from another file of your choosing.

Output file redirection allows you to specify the file to which the sequence of bytes written to **stdout** and **stderr** end up. That is, instead of a command writing its standard output and standard error streams to the terminal screen device file, it writes them to another file of your choosing.

The **stdin** stream is redirected on the command line using the **<** metacharacter, and the **stdout** and **stderr** streams are redirected together on the command line using the **>** metacharacter. For example:

```
$ command < inputfile
$ command > outputfile
$ command < inputfile > outputfile
```

UNIX Pipes

The UNIX system allows **stdout** of a command to be connected to **stdin** of another command. You can make it do so by using the pipe connector (**|**) metacharacter. For example:

```
$ command1 | command2
```

You can form long chains of commands in a single command line using the pipe connector. For example:

```
$ command1 | command2 | command 3 | command 4 ...
```

Procedure

Extend the shell program you wrote in the previous lab to support all of the following metacharacters (the last four are new):

Metacharacter	Purpose	Example
`/`	To be used as the root directory and as a component separator in a pathname	`/usr/bin`
`New Line`	To end a command line	
`Space`	To separate elements on a command line	`ls /etc`
`Tab`	To separate elements on a command line	`ls /etc`
`.`	To represent the current working directory	`ls .`
`..`	To represent the parent directory of the current working directory	`ls ..`
`&`	To execute a command in the background	`command &`
`|`	To create a pipe between commands	`command1 | command2`
`<`	To redirect input for a command	`command < file`
`>`	To redirect output for a command	`command > file`

Hints:

- Take a look at the **`man`** page for the following C functions provided by the standard C library and system call library:
  - `pipe   (use for creating a pipe)`
  - `read   (use for reading from a pipe)`
  - `write  (use for writing to a pipe)`
  - `open   (use for file redirection - see next hint)`
  - `close  (use for file redirection - see next hint)`
  - `dup    (use for file redirection - see next hint)`

- Each process has a file descriptor table that maps integer values to file pointers. By default, *fileDescriptor[0]* maps to **stdin**, *fileDescriptor[1]* maps to **stdout**, and *fileDescriptor[2]* maps to **stderr**. You can, for example, redirect the output stream to a file by first opening the file to be used as the output using the **`open()`** system call, then closing *fileDescriptor[1]* using the **`close()`** system call, then using the **`dup()`** system call on the file you just opened to create a duplicate entry in the file descriptor table. The duplicate entry is placed in the first available slot in the table, which in this case is *fileDescriptor[1]*. This is the key to redirecting **stdin**, **stdout**, and **stderr**.

```
fid = open(foo, O_WRONLY | O_CREAT);
close(1);
dup(fid);
close(fid);
```

- Test your shell with command lines like:

```
man socket > file.txt
man socket | grep open
man socket | grep open > file.txt
gcc minishell.cc &
man socket | grep open > file.txt &
```

Deliverables

1. All of the source and header files for your extended shell program.
2. A make file for building your program. The output executable file should be called **`minishell`**.