Assignment 3: The Lazy Shell

CS-232 Operating Systems
Spring 2022

Due: 11:59pm Tuesday, 1st November 2022

1 Overview

In this assignment, you will implement a basic command shell known as Lazy, exclusively for amateur hackers. Your shell will have to support I/O redirection, pipes, signals, and background processes. The primary goal of this assignment is to familiarize yourself with the basic functionality of the Linux operating system including processes, files, and interprocess communication.

2 Instructions

At the heart of a Linux shell are two system calls: `fork()` which creates a new child process as a duplicate of its parent and `exec()` which replaces the image of the callee process pertaining to an executable. So, the structure of our shell program looks like the following:

```
main() {
    while (1) {
        <Display prompt>
        <Input command>
        <Parse the command args>
        int pid = fork();
        /* From this point on, there are two instances of the program */
        /* pid takes different values in the parent and child processes */
        if (pid == 0) {
            /* Execute the command in child process */
            exec(...args...);
        }
        else {
            /* Wait for child process to terminate */
            waitpid(pid);
        }
    }
}
```

Command Format:

We will assume the following format for commandline:

```
#path/to/dir> cmd [ | cmd2 | cmd3...] [<inFile] [>outFile] [&]
```

Each `cmd` itself is of the form `cmdname [arg1 arg2 ...]` and as implied above, more than one commands may be optionally piped together with | as separator. Commandline may optionally have input and/or output direction indicated by an angular bracket followed by a filename. The ampersand & in the end indicates a background command.

Command parsing has already been done for you in the accompanied `parse.c`. You can optionally install and use `readline` library for displaying the prompt and reading commands. You will read man pages for the required system calls and your main task is to build a working shell that supports:

2.1 I/O Direction

To achieve I/O direction, modify the child process to redirect its I/O stream to a file just before calling `exec()`. Open the appropriate file using `open()` and then use `dup2()` to replace the std input/output with the new file descriptor. Try command `sort <words.txt >sorted.txt` in your terminal as a testing example with an existing words.txt. The sorted.txt file does not need to exist before the command.

2.2 Background Jobs

A background command is contrasted from a normal command in that our shell does not wait for it to complete and simply continues with the prompt. Our shell maintains a record of all running background processes with their `cmdname`s and `ID`s. The `ID` as returned by the `fork()` call will be used only by the shell and the shell will assign each job a local `pID` for user's reference. Assume that at most 10 such background jobs can run at a time and each is identified with a local `pID` in the range 1-10 by the user. You will find `sleep` command useful as a testing example.

Hint:

A background running child process may finish anytime and our shell needs to keep track of them! Make a signal handler with the `signal()` call. Set the `signum` argument to `SIGCHLD` so signal handler will be called whenever a child process terminates. To get the `ID` of the terminated child process use `waitpid()` with `WNOHANG` as third argument to return immediately.

2.3 Built-in Commands

Apart from running executables, a shell should support a few essential built-in commands. For our shell, the following commands will trigger the indicated actions:

1. `jobs` - provides a list of all background processes and their local pIDs.
2. `cd PATHNAME` - sets the `PATHNAME` as working directory.

3. `history` - prints a list of previously executed commands. Assume 10 as the maximum history size.
4. `kill PID` - terminates the background process identified locally with `PID` in the jobs list.
5. `!CMD` - runs the command numbered `CMD` in the command history.
6. `exit` - terminates the shell only if there are no background jobs.
7. `help` - prints the list of builtin commands along with their description.

2.4 Pipes

Pipes allow one-way communication between two related processes. In this context, piping means taking the output of one command and feeding it as input to another command and, this is done using the same IPC mechanism. As an example, you can try running `ls | head -3` in your terminal which should print the top 3 items in the current directory.

Hint:

To create a pipe, use `pipe()` call followed by a `fork()` as shown in the following snippet. Then duplicate its reading and writing ends to child's output and parent's input FD, respectively.

```
int pd[2];
pipe(pd);
if (!fork()) {  /* Child prints on the writing end */
    dup2(pd[1], 1);     /* Redirect output to the write end */
    close(pd[0]);    /* Close the unused read end */
    exec(...args...);   /* Call exec() here */
}
else{   /* Parent reads on input */
    dup2(pd[0], 0);     /* Redirect input to the read end */
    close(pd[1]);    /* Close the unused write end */
    <Read the output of child process from input>
}
```

3 Edge Cases

Make sure to print an error message in the case of:

1. Input redirecting to a non-existing file.
2. I/O redirecting to an inaccessible file. Use `access()` call.
3. Exiting shell with background jobs running.
4. Changing directory to a non-existing path.
5. Entering an ambiguous command in which case `exec()` will return.

4 Submission

Your submission will consist of all '.c' and '.h' files along with a Makefile with rules for 'build', 'rebuild', 'clean' and 'run'. Please submit your code as a single zipped file on the canvas.

5 Rubric

1. Shell prompt working [10 marks]
2. Input/Output redirection working [20 marks]
3. Background jobs working [20 marks]
4. Built-in commands working [20 marks]
5. Pipes working [20 marks]
6. Memory deallocation and edge cases [10 marks]