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>
        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 piped together using | as separator. Commandline may have input and/or output direction indicated by an angular bracket followed by filenames. The ampersand & in the end indicates a background job.

Command parsing has already been done for you in the `parse.c`. Your task is to read man pages for the required system calls and 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` as a testing example.

2.2 Background Jobs

Our shell is not to wait for a background command to complete but simply continue with the prompt. Our shell maintains a record of all running background processes with their `cmdname`s and `pID`s. Assume that at most 10 background jobs can run at a time and each is identified with a local pID in the range 1-10.

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 `pID` 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 enumerated from 1.
2. `cd PATHNAME` - sets the `PATHNAME` as working directory.
3. `history` - prints a list of previously executed commands. Assume 10 as 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

Pipe is an IPC mechanism to allow one-way communication between processes. In this context, piping means taking the output of one command as input for another command. As an example, you can try running `ls | grep -i '.*.c$' | wc | awk 'print $1'` which prints the number of '.c' files 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 writes on output */
    dup2(pd[1], 1);     /* Output redirect to write end of pipe */
    close(pd[0]);    /* Close unused read end */
    printf("talk to parent");   /* or call exec() */
}
else{   /* Parent reads on input */
    dup2(pd[0], 0);     /* Input redirect to read end of pipe */
    close(pd[1]);    /* Close unused write end */
    scanf("%s", buf);
}
```

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]