Project 1: 352> Shell

COM S 352 Spring 2021

1. Introduction

This project consists of designing a C program to serve as a shell interface that accepts user commands and then executes each command in a separate process. Your implementation will support input and output redirection, pipes, and background processes. Completing this project will involve using the Linux `fork()`, `exec()`, `wait()`, `waitpid()`, `pipe()` and `dup2()` system calls. It must be tested on pyrite.

2. Main Loop

In its normal mode of operation, the shell prompts the user, after which the next command is entered. In the example below, the prompt is `352>` and the user's next command is `cat -n prog.c`.

```
352> cat -n prog.c
```

The command is executed while the shell waits for it to complete, after completion the shell prompts the user for the next command. Following the UNIX philosophy, most commands are separate utility programs distributed along with the OS, there are only a few built-in commands: `exit`, `jobs` and `bg`. The purpose of exit should be clear, `jobs` and `bg` are described in Section 7.

Most of the main loop is provided for you in the starter code in Section 9.

3. Basic Commands

A basic command consists of the name of an executable file, followed by zero or more space separated arguments. The input and parsing of the command line is performed for you in the starter code. Parsing populates an instance of the `Cmd` struct. For example, calling the parser with the argument `cmd` having the field `cmd->line` of `"cat -n prog.c"` results in the following assignments (note that the character `\0` in a string literal means `NULL`).

```
cmd->line      = "cat -n prog.c";
cmd->tokenLine = "cat\0-n\0prog.c";
cmd->args      = {pointer_to_c, pointer_to_minus_sign, pointer_to_p, NULL…};
cmd->symbols   = {NULL,         NULL,                  NULL,         NULL…};
```

Commands should be run in a separate child process. Basic commands must be executed in the foreground, this means the shell is blocked (waiting) until the child process terminates. The child process is created using the `fork()` system call, the user's command is executed using

one of the system calls in the `exec()` family and the shell waits for the command to complete using one of the system calls in the `wait()` family. Get started by writing a function to execute a command and call it from `main()` at the location indicated by the line:

```
/* TODO: Run command in foreground. */
```

4. File Redirection

Your shell should be modified to support the `'>'` and `'<'` redirection operators, where `'>'` redirects the output of a command to a file and `'<'` redirects the input to a command from a file. For example, if a user enters

```
352> ls > out.txt
```

the output from the `ls` command will be redirected to the file `out.txt`. Similarly, input can be redirected as well. For example, if the user enters

```
352> sort < in.txt
```

the file `in.txt` will serve as input to the `sort` command.

The parser recognizes `'<'` and `'>'` as special symbols, for example, calling the parser with the argument `cmd` having the field `cmd->line` of `"sort < in.txt"` results in the following.

```
cmd->line      = "sort < in.txt";
cmd->tokenLine = "sort\0<\0in.txt";
cmd->args      = {pointer_to_s, NULL,          pointer_to_i, NULL…};
cmd->symbols   = {NULL,         pointer_to_lt, NULL,         NULL…};
```

Simplifying Assumption: You can assume that commands will contain either one input or one output redirection and will not contain both. In other words, you do not have to be concerned with command sequences such as `sort < in.txt > out.txt`.

5. Pipes

Your shell should be modified to support the `'|'` pipe operator, which connects the `stdout` of one process to the `stdin` of another.

```
352> ls | grep txt
```

The output from the `ls` command will be piped to the utility `grep`.

Managing the redirection of both input and output will involve using the `dup2()` function, which duplicates an existing file descriptor to another file descriptor. For example, if `fd` is a file descriptor to the file `out.txt`, the call

```
dup2(fd, STDOUT_FILENO);
```

duplicates `fd` to standard output (the terminal). This means that any writes to standard output will in fact be sent to the `out.txt` file.

Simplifying Assumption: You can assume that commands will contain no more than one pipe. In other words, you do not have to be concerned with command sequences such as `ls | grep txt | wc -l`.

6. Background Commands

The use of an ampersand (`&`) at the end of a line indicates that the command should be executed in the background. The shell does not wait for background commands to complete, it immediately prompts the user for the next command. However, the shell is expected to monitor the status of background commands, in order to inform the user when a command completes. Helper functions should be created and called from main() to start a command and check if a command is completed at the following TODO lines.

```
/* TODO: Run command in background. */
/* TODO: Check on status of background processes. */
```

When a background command is started the shell assigns it a number (starting at 1) and displays the `pid` of the process. When the process exits normally the shell displays the message `Done commandArgs` as shown in the following example.

```
352> ls -la &
[1] 84653
-rw-r--r--   1 user  group     100 Aug 28 03:22 file1
-rw-r--r--   1 user  group     100 Aug 28 03:22 file2
-rw-r--r--   1 user  group     100 Aug 28 03:22 file3
352>
[1] Done ls -la
```

It is common for the `Done` message to not be displayed immediately, for example it may only be displayed after the user's next line of input (which can be a blank line as shown above). This is due to the status check only being called once per iteration of the main loop. Because the status check cannot block the shell, like it does with foreground processes, `waitpid()` should be called in a non-blocking mode as discussed in lecture.

Sometimes a process exits with a non-zero exit code, in that case the shell should display (replacing `exitCode` and `commandArgs`): `Exit exitCode commandArgs`

```
352> grep &
[1] 84867
usage: grep [-abcDEFGHhIiJLlmnOoqRSsUVvwxZ] [-A num]
     [-B num] [-C[num]] [-e pattern] [-f file]
     [--binary-files=value] [--color=when]
     [--context[=num]] [--directories=action] [--label]
     [--line-buffered] [--null] [pattern] [file ...]
352>
[1] Exit 2 grep
```

A process might not exit on its own and instead it is terminated using the kill command, when that happens the shell should display (replacing commandArgs): `Terminated commandArgs`.

```
352> sleep 100 &
[1] 84665
352> kill 84665
[1] Terminated sleep 100
```

The kill command works by sending an unhandled signal such as `SIGTERM` to a process. You can test if a child process was terminated in this way, as opposed to the process self-exiting with a call to `exit()`, by using `WIFSIGNALED`.

Finally, add the capacity to run multiple background processes concurrently.

```
352> sleep 15 &
[1] 85119
352> sleep 8 &
[2] 85120
352> sleep 20 &
[3] 85122
352>
[2] Done sleep 8
352>
[1] Done sleep 15
352>
[3] Done sleep 20
```

Your shell can now perform sleep sort!

Simplifying Assumption: You can assume that background commands will not contain a pipe. In other words, you do not have to be concerned with command sequences such as `ls | grep txt &`.

7. Job Control Commands

Jobs include background commands and stopped foreground commands. Implement the following built-in commands to control jobs.

jobs

Implement the built-in `jobs` command which prints the list of background commands and stopped commands. The status of a job can be running or stopped. Background commands created with `&` are running by default. The following is an example.

```
352> jobs
[1]   Stopped                 sleep 100
[2]   Running                 sleep 100 &
[3]   Running                 sleep 100 &
```

The next two requirements provide situations where a command may switch between running and stopped.

control+z

Implement `control+z` to stop the currently running foreground command. When a user presses `control+z`, the terminal send the signal `SIGTSTP (SIGnal - Terminal SToP)` to the shell. Forwarding this signal, from the shell to the process running in the foreground, has already be implemented for you with the function `sigtstpHandler()`. What still needs to be done, is the stopped command needs to be add to the list of jobs. Its status is stopped.

bg job_id

Implement the `bg job_id` command which resumes a stopped command. The status of the command should be set to running. For example, consider the following scenario.

```
352> sleep 100
[User presses control+z.]
352> jobs
[1]   Stopped                 sleep 100
352> bg 1
[1]   Running                 sleep 100
```

To make a process continue from a stopped state, send it a `SIGCONT` signal.

Simplifying Assumption: You can assume that stoped commands will not contain a pipe. In other words, you do not have to be concerned with the user pressing control+z on a command sequence such as `ls | grep txt`.

8. Additional Requirements

makefile (5 points)

You get 5 points for simply using a makefile. Name your source files whatever you like. Please name your executable **shell352**. Be sure that "`make`" will build your executable **on pyrite**.

Documentation (10 points)

If you have more than one source file, then you must submit a **Readme** file containing a brief explanation of the functionality of each source file. Each source file must also be well-documented. There should be a paragraph or so at the beginning of each source file describing the code; functions and data structures should also be explained. Basically, another programmer should be able to understand your code from the comments.

Project Submission

Put all your source files (including the `makefile` and the `README` file) in a folder. Then use command `zip -r <your ISU Net-ID> <src_folder>` to create a `.zip` file. For example, if your Net-ID is `ksmith` and `project1` is the name of the folder that contains all your source files, then you will type `zip -r ksmith project1` to create a file named `ksmith.zip` and then submit this file.

9. Starter Code

You are not required to use this code, however, it is highly recommended. Use the TODOs as a guide to start working on the project.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <signal.h>

#define MAX_LINE 80
#define MAX_ARGS (MAX_LINE/2 + 1)
#define REDIRECT_OUT_OP '>'
#define REDIRECT_IN_OP '<'
#define PIPE_OP '|'
#define BG_OP '&'

/* Holds a single command. */
typedef struct Cmd {
    /* The command as input by the user. */
    char line[MAX_LINE + 1];
    /* The command as null terminated tokens. */
    char tokenLine[MAX_LINE + 1];
    /* Pointers to each argument in tokenLine, non-arguments are NULL. */
    char* args[MAX_ARGS];
    /* Pointers to each symbol in tokenLine, non-symbols are NULL. */
    char* symbols[MAX_ARGS];
    /* The process id of the executing command. */
    pid_t pid;

    /* TODO: Additional fields may be helpful. */

} Cmd;

/* The process of the currently executing foreground command, or 0. */
pid_t foregroundPid = 0;
```

```
/* Parses the command string contained in cmd->line.
 * Assumes all fields in cmd (except cmd->line) are initailized to zero.
 * On return, all fields of cmd are appropriatly populated. */
void parseCmd(Cmd* cmd) {
    char* token;
    int i=0;
    strcpy(cmd->tokenLine, cmd->line);
    strtok(cmd->tokenLine, "\n");
    token = strtok(cmd->tokenLine, " ");
    while (token != NULL) {
        if (*token == '\n') {
            cmd->args[i] = NULL;
        } else if (*token == REDIRECT_OUT_OP || *token == REDIRECT_IN_OP
                || *token == PIPE_OP || *token == BG_OP) {
            cmd->symbols[i] = token;
            cmd->args[i] = NULL;
        } else {
            cmd->args[i] = token;
        }
        token = strtok(NULL, " ");
        i++;
    }
    cmd->args[i] = NULL;
}

/* Finds the index of the first occurance of symbol in cmd->symbols.
 * Returns -1 if not found. */
int findSymbol(Cmd* cmd, char symbol) {
    for (int i = 0; i < MAX_ARGS; i++) {
        if (cmd->symbols[i] && *cmd->symbols[i] == symbol) {
            return i;
        }
    }
    return -1;
}

/* Signal handler for SIGTSTP (SIGnal - Terminal SToP),
 * which is caused by the user pressing control+z. */
void sigtstpHandler(int sig_num) {
    /* Reset handler to catch next SIGTSTP. */
    signal(SIGTSTP, sigtstpHandler);
    if (foregroundPid > 0) {
        /* Foward SIGTSTP to the currently running foreground process. */
        kill(foregroundPid, SIGTSTP);

        /* TODO: Add foreground command to the list of jobs. */

    }
}
```

```
int main(void) {
    /* Listen for control+z (suspend process). */
    signal(SIGTSTP, sigtstpHandler);
    while (1) {
        printf("352> ");
        fflush(stdout);
        Cmd *cmd = (Cmd*) calloc(1, sizeof(Cmd));
        fgets(cmd->line, MAX_LINE, stdin);

        parseCmd(cmd);
        if (!cmd->args[0]) {
            free(cmd);
        } else if (strcmp(cmd->args[0], "exit") == 0) {
            free(cmd);
            exit(0);

        /* TODO: Add built-in commands: jobs and bg. */

        } else {
            if (findSymbol(cmd, BG_OP) != -1) {

                /* TODO: Run command in background. */

            } else {

                /* TODO: Run command in foreground. */

            }
        }

        /* TODO: Check on status of background processes. */

    }
    return 0;
}
```