UNIX Shell

Adapted from Project 1 in the textbook.

This project consists of designing a C/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, as well as background processes. Completing this project will involve using the UNIX fork(), exec(), wait() and dup2() system calls, it must be tested on pyrite.

The examples in this project description are color coded as shown.

```
Shell output is blue.
User input is green.
Application output is black.
```

1. Overview

A shell interface gives the user a prompt, after which the next command is entered. The example below illustrates the prompt `352>` and the user's next command: `cat prog.c`. (This command displays the file prog.c on the terminal using the UNIX cat command.)

```
352> cat prog.c
```

One technique for implementing a shell interface is to have the parent process first read what the user enters on the command line (in this case, `cat prog.c`) and then create a separate child process that performs the command. Unless otherwise specified, the parent process waits for the child to exit before continuing. This is similar in functionality to the new process creation illustrated in Figure 3.9. However, UNIX shells typically also allow the child process to run in the background, or concurrently. To accomplish this, we add an ampersand (&) at the end of the command. Thus, if we rewrite the above command as

```
352> cat prog.c &
```

the parent and child processes will run concurrently.
The separate child process is created using the fork() system call, and the user's command is executed using one of the system calls in the exec() family (as described in Section 3.3.1).

A C program that provides the general operations of a command-line shell is supplied below. The main() function presents the prompt `352>` and outlines the steps to be taken after input from the user has been read. The main() function continually loops as long as should_run equals 1; when the user enters exit at the prompt, your program will set should_run to 0 and terminate.

```
#include <stdio.h>
#include <unistd.h>

#define MAX_LINE 80  /* The maximum length command */

int main(void) {
char *args[MAX_LINE/2 + 1]; /* command line arguments */
int should_run = 1; /* flag to determine when to exit
program */

  while (should_run) {
    printf("352>");
    fflush(stdout);

    /**
     * After reading user input, the steps are:
     * (1) fork a child process using fork()
     * (2) the child process will invoke execvp()
     * (3) parent will invoke wait() unless command
     * included &
     */
  }
  return 0;
}
```

2. Redirecting Input and Output

Your shell should then 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.

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.

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.

3. Background Processes

The use of an ampersand (&) at the end of a line indicates that the command should be executed as a background process. When a background process is created 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 (replacing commandArgs): `Done commandArgs`

```
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). Maintaining this convention will make our implementation simpler because we do not have to implement non-blocking input in the main loop. In other words, we can just wait for the users next input and then check for any exited background processes.

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> cat &
[1] 84665
352> kill 84665
[1] Terminated cat
```

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> sleep 10 &
[4] 85123
[2] Done sleep 8
[4] Done sleep 10
[1] Done sleep 15
[3] Done sleep 20
```

Your shell is now capable of sleep sort, that's great, because why implement sorting algorithms when your OS can do it for you?

4. Requirements and Grading Criteria

4.1. 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.

4.2. 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.

4.4. Main Loop (10 points)

The general structure of the main loop is given in the project description. You program should terminate when user type the command `exit`.

4.5 Executing Command in a Child Process (15 points)

A "foreground job" is a command of the form ARG0 ARG1 ... ARGn .

Examples:

```
cp file1 file2
cp -R dir1 dir2
rm -f -R dir1 dir2 file1 file2
```

A foreground job should be executed as follows:

1. Create a new process using fork()
2. Replace the image of the created process with the desired command using execvp().
3. Wait for the process to terminate using waitpid().

4.6 Redirecting Input and Output (30 points)
See project desciption for details.

4.7 Background Processes (30 points)
See project desciption for details.

4.8 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.