Shell Project

You may discuss this Schedule Project description with others to understand the problem. You may use the Internet for point-specific questions. For example, you may search the Internet for how to call `dup` and `dup2`. You **may not** use the Internet to discover shell code that is copied or mimicked into your solution. You are provided a starting code base along with descriptions in this document. Use this material to design and implement your own code.

Use the file `ShellProjectAnswers.docx` to answer questions in this project.

Introduction

A Linux shell is a command line interpreter. A shell is a program that allows a user to enter commands such as `cd`, `ls`, `cat`, `more`, and `gcc`. Some of the commands are executed by the shell and some of the commands are programs that the shell runs via `fork` and `exec` system calls.

Shell - stdin, stdout, stderr and Redirection

A shell supports input from standard input, output to standard output, and error messages to standard error. By default standard input is the user's typed commands, standard output consists of prints to the terminal window, and standard error consists of errors to the terminal window. A shell allows I/O redirection. You can redirect standard input to be from a file. You can also redirect standard output and standard error to a file. When a shell starts running the integer file descriptors associated streams stdin, stdout, and stderr are 0, 1, and 2, respectively.

A shell achieves redirection by closing a file descriptor of `stdin`, `stdout`, or `stderr`; and opening a file. Since file descriptors are allocated from the lowest to the highest, the newly opened file is now connected to the file descriptor of (for example) stdout.

dup and dup2 System Calls

The dup and dup2 system calls are helpful.

```
int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

Let's first understand the `dup` function. The `dup` system call creates a copy of the file descriptor oldfd, using the lowest-numbered unused file descriptor for the new descriptor. After a successful return, the old and new file descriptors may be used interchangeably. They refer to the same open file description (see open) and thus share file offset and file status flags; for

example, if the file offset is modified by using lseek on one of the file descriptors, the offset is also changed for the other.[1]

The `dup2` system call performs the same task as `dup`, but instead of using the lowest-numbered unused file descriptor, it uses the file descriptor number specified in `newfd`. If the file descriptor `newfd` was previously open, it is silently closed before being reused. The steps of closing and reusing the file descriptor newfd are performed atomically. This is important, because trying to implement equivalent functionality using `close` and `dup` would be subject to **race** conditions, whereby `newfd` might be reused between the two steps. Such reuse could happen because the `main` program is interrupted by a signal handler that allocates a file descriptor, or because a parallel thread[2] allocates a file descriptor.[3]

The following code shows how to use `dup2` to redirect `stdin` and `stdout` from/to files. The code is provided in `redirect_stdout.c`.

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

/*
 * $ gcc -o redirect_stdinout redirect_stdout.c
 * $ ./redirect_stdinout infile outfile
 */

int main(int argc, char **argv) {
    int pid, status;
    int infd, outfd;

    if (argc != 3) {
        fprintf(stderr, "usage: %s input_file output_file\n", argv[0]);
        exit(1);
    }
    if ((infd = open(argv[1], O_RDONLY, 0644)) < 0) {
        perror(argv[1]);
        exit(1);
    }
    if ((outfd = open(argv[2], O_CREAT|O_TRUNC|O_WRONLY, 0644)) < 0) {
        perror(argv[2]);
        exit(1);
    }
```

[1] Words from Linux man
[2] We study threads during our Concurrency study
[3] Words from Linux man

```
    printf("After this message, stdout/stdin redirected to/from \"%s\"
\"%s\".\n", argv[0], argv[1]);

    /*
     * stdin is fd 0, stdout is fd 1
     * dup2 connects our file to fd 1 (stdout)
     */
    dup2(infd,  0);
    dup2(outfd, 1);

    char line[100];
    fgets(line, 100, stdin);
    printf("outfile: %s: data from infile: %s\n", argv[1], line);
    exit(0);
}
```

To build and run the above code, you do the following.

```
$ gcc -o redirect_stdout redirect_stdout.c
$ ./redirect_stdout infile outfile
```

Note this sample does not use the redirect arrows like a shell. The following is an example of redirection of output.

```
$ cat test.txt > cat.txt
```

Shell Pipes

A shell supports piping the output of one program to the input of another. You can pipe the output of `ls` to `grep` as follows.

```
$ ls | grep redirect
redirect_input.txt
redirect_output.txt
redirect_stdout
redirect_stdout.c
```

A shell achieves piping by using the system call `pipe` along with `fork` and `exec`.

A pipe system call returns two file descriptors in an array.

```
int p[2];
int status = pipe(p);
```

The input file descriptor is in p[0] and the output file descriptor is in p[1]. The following code[4] shows creating a pipe, writing to the output file descriptor, and reading from the input file descriptor. The code is provided in file `simple_pipe.c`.

```
// Code from https://www.geeksforgeeks.org/pipe-system-call/
```

[4] Code from https://www.geeksforgeeks.org/pipe-system-call/, tweaked a bit.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#define MSGSIZE 16
char* msg1 = "hello, world #1";
char* msg2 = "hello, world #2";
char* msg3 = "hello, world #3";

int main() {
    char inbuf[MSGSIZE];
    int p[2], i;
    if (pipe(p) < 0)
        exit(1);
    write(p[1], msg1, MSGSIZE);
    write(p[1], msg2, MSGSIZE);
    write(p[1], msg3, MSGSIZE);
    for (i = 0; i < 3; i++) {
        read(p[0], inbuf, MSGSIZE);
        printf("%s\n", inbuf);
    }
    return 0;
}
```

The following code[5] connects two processes (parent and child) with two pipes. The code is provided in file `parent_child_pipe.c`. The parent writes to the first pipe and the child reads from the first pipe. The child writes to the second pipe and the parent reads from the second pipe. Notice how a process closes the ends of pipes that are not used.

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/types.h>
#include<string.h>
#include<sys/wait.h>

int main() {
    // First pipe to send input string from parent to child
    // Second pipe to send concatenated string from child to parent
    int fd1[2]; // Used to store two ends of first pipe
    int fd2[2]; // Used to store two ends of second pipe
    char fixed_str[] = " CPSC 405";
    char input_str[100];
    pid_t p;
    if (pipe(fd1)==-1) {
        fprintf(stderr, "Pipe Failed" );
```

[5] Code from https://www.geeksforgeeks.org/c-program-demonstrate-fork-and-pipe/, tweaked a bit

```
        return 1;
    }
    if (pipe(fd2)==-1) {
        fprintf(stderr, "Pipe Failed" );
        return 1;
    }
    scanf("%s", input_str);
    p = fork();
    if (p < 0) {
        fprintf(stderr, "fork Failed" );
        return 1;
    } else if (p > 0) { // Parent process
        char concat_str[100];
        close(fd1[0]); // Close read end of first pipe
        write(fd1[1], input_str, strlen(input_str)+1); // write input string
        close(fd1[1]); // close write end of first pipe
        wait(NULL);    // Wait for child to send a string
        close(fd2[1]); // Close write end of second pipe

        read(fd2[0], concat_str, 100); // Read from child
        printf("Concatenated string: %s \n", concat_str);
        close(fd2[0]);  // close reading end
    } else { // child process
        close(fd1[1]); // Close write end of first pipe
        char concat_str[100];
        read(fd1[0], concat_str, 100); // Read string using first pipe

        int k = strlen(concat_str); // Concatenate a fixed string with it
        for (int i=0; i<strlen(fixed_str); i++)
            concat_str[k++] = fixed_str[i];
        concat_str[k] = '\0'; // string ends with '\0'

        close(fd1[0]); // Close both reading ends
        close(fd2[0]);
        write(fd2[1], concat_str, strlen(concat_str)+1); // write concatenated
string
        close(fd2[1]); // Close write end of second pipe
        exit(0);
    }
}
```

The following shows running `parent_child_pipe`. I typed `Hello`, which is read by the parent, which writes it to a pipe, which is read by the child, which concatenates it to CPSC 405, which writes it to a pipe, which is read by the parent, which prints the final string to stdout.

```
gustysh $ ./parent_child_pipe
Hello
Concatenated string: Hello CPSC 405
```

Commands - Native and Local

Execute the `ls` program that is native to Linux. The following shows a Linux `ls` in `gustysh`, which is a shell I created with help from code on the Internet.

```
gustysh $ ls -slag
ls -slag
total 160
 0 drwxr-xr-x   9 staff    288 Dec 17 21:11 .
 0 drwxr-xr-x  18 staff    576 Dec 20 17:11 ..
16 -rw-r--r--@  1 staff   6148 Dec 17 20:42 .DS_Store
32 -rwxr-xr-x   1 staff  13948 Dec 17 20:41 cat
16 -rw-r--r--   1 staff   4754 Dec 10 20:30 cat.c
40 -rwxr-xr-x   1 staff  18860 Dec 17 20:41 ls
24 -rw-r--r--@  1 staff  10767 Dec  9 21:46 ls.c
32 -rwxr-xr-x   1 staff  14572 Dec 17 21:10 mainshell
 0 drwxr-xr-x@  9 staff    288 Dec 17 21:11 simple-shell-master
```

Execute your own version of the `ls` command. The following shows running a local `ls` in `gustysh`.

```
gustysh $ ./ls -l
-rwxr-xr-x 1      gusty      staff     13948 Dec 17 20:41 cat
-rw-r--r-- 1      gusty      staff      4754 Dec 10 20:30 cat.c
-rwxr-xr-x 1      gusty      staff     18860 Dec 17 20:41 ls
-rw-r--r-- 1      gusty      staff     10767 Dec  9 21:46 ls.c
-rwxr-xr-x 1      gusty      staff     14572 Dec 17 21:10 mainshell
drwxr-xr-x 9      gusty      staff       288 Dec 17 21:11 simple-shell-master
```

Pipes and Redirection

The following shows piping a local ls program into the Linux grep command.

```
gustysh $ ./ls | grep sim
simple-shell-master
simple_pipe
simple_pipe.c
```

The following shows piping a local `ls` program into the Linux `grep` command and redirecting the output to a file.

```
gustysh $ ./ls | grep sim > localls_grep.txt
 grep sim > localls_grep.txt
gustysh $ cat localls_grep.txt
cat localls_grep.txt
simple-shell-master
simple_pipe
simple_pipe.c
```

Background Processes

Shells allow users to run a program as a background process. The shell starts the program as a process in the background and returns a prompt for more user input. The following shows running `parent_child_pipe` as a background process.

```
$ ./parent_child_pipe &
[1] 15725
Gustys-iMac:ShellProject gusty$ ps
  PID TTY           TIME CMD
15717 ttys000    0:00.03 -bash
15725 ttys000    0:00.00 ./parent_child_pipe
  438 ttys001    0:00.01 -bash
  459 ttys002    0:00.02 -bash
 5271 ttys002    0:00.12 python
 3451 ttys003    0:00.21 -bash
```

Reading Commands

Your shell program will be in an endless loop that terminates when the user enters Ctl-D or exit. On each iteration of the loop, your program reads a line that has several strings separated by whitespace. For example, the user may enter `$ ls  > out.txt`. You have to extract the strings `ls`, `>`, and `out.txt` from the input. There are many ways to accomplish this. You can use the C function `strtok`, you can use `split.c` and `split.h` (we used these in FirstLab), or you can create your own. The following is an example of creating your own. The code is provided in `skipwhitespace.c`.

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

char buf[100]; // buffer for line
char whitespace[] = " \t\r\n\v";
char *words_on_line[10]; // 10 words on a line

int main(int argc, char **argv) {
    int stop = 0;
    while (1) {
        fgets(buf, 100, stdin);
        char *s = buf;
        char *end_buf = buf + strlen(buf);
        int eol = 0, i = 0;
        while (1) {
            while (s < end_buf && strchr(whitespace, *s))
                s++;
            if (*s == 0) // eol - done
```

```
                break;
            words_on_line[i++] = s;
            while (s < end_buf && !strchr(whitespace, *s))
                s++;
            *s = 0;
        }
        for (int j = 0; j < i; j++)
            printf("words_on_line[%d]: %s\n", j, words_on_line[j]);
        if (strcmp(words_on_line[0],"stop") == 0)
            break;
    }
}
```

Your Shell

Design and implement a shell that satisfies the following.

- Design and Implement `cd` and `exit` as internal commands. Internal commands are not executing via `fork` and `exec`.
- Design and Implement other commands by running programs via `fork` and `exec`. You have to run programs that are native to Linux. You have to implement two of the native Linux programs. You have to implement an `ls` program. You choose your second program. For example, implement `cat` or `more`. `More` is tricky. Maybe you want to invent a brand new command - something like a `dog` command.
- Design and Implement I/O redirection. For our shell, you only have to support either output redirection or input redirection on a line - not both on the same line.
- Design and Implement the ability to pipe one command to another. A typical Linux shell allows stringing pipes together - for example, `ls | grep toy | grep cup`. Processing an arbitrary number of pipes complicates the design. Our shell will support one pipe.
- Design and Implement running a command in the background - for example `$ ls &`
- Design and Implement signal handlers for Ctl-D and Ctl-C.
    - Ctl-D is equivalent to an exit command. Your shell exits.
    - Ctl-C terminates the foreground process. The following shows a Ctl-C terminating `parent_child_pipe`

      ```
      gustysh $ ./parent_child_pipe
      ./parent_child_pipe
      ^C
      gustysh $
      ```

You are provided code to study. You cannot search the Internet for additional shell or user program code. You can search for specific help - for example, how to write a signal handler.

- Xv6 user programs, which are on Canvas in Files > Xv6 > uprogs. The Xv6 user programs include a shell (in file sh.c) and many of the Linux utility programs (e.g., ls, cat, etc.)

Guidance

1. Use a regular Linux shell to fully understand all of the proposed features.
2. Study the provided code to see design and implementation techniques.
3. Select the design techniques that you like.
4. Create the design artifact of your shell.
5. Implement the design in an incremental approach. For example,
    a. Implement the basic command loop. First create a program that reads a line and separates the input based on whitespace. For example, your program may place the input line `$ ls -l loop.c` into two variables `cmd` and `args`. `char *cmd` points to `"ls"`, and `char **args` is an array of pointers to `"ls"`, `"-l"`, and `"loop.c"`. Your shell then executes the command with code something like the following.

```
if (fork() == 0) {
    execvp(cmd, args);
    fprintf(stderr, "sh: cmd not found: %s\n", cmd);
    exit(1);
}
wait(NULL);
```

    b. Implement the internal commands. Internal commands are not executing via `fork` and `exec`. Suppose you have read the input into a `char *line`. The following shows one way of implementing the internal `cd` command.

```
if(line[0] == 'c' && line[1] == 'd' && line[2] == ' '){
    if(chdir(line+3) < 0) // chdir has no effect on parent
                          // if run in the child.
        fprintf(stderr, "cannot cd %s\n", line+3);
    continue;
}
```

    c. Implement the ability to `fork` and `exec` a program. Use this capability to run a native Linux command.
    d. Implement your local `ls` program. Use your shell's `fork` and `exec` to run your local `ls`.
    e. Implement I/O redirection. Use native and local commands.
    f. Implement the ability to run programs in the background.
    g. Implement Ctl-D and Ctl-C handlers.

Submission

Be sure your submission uses the `ShellProjectAnswers.docx` to answer the questions so I can easily decipher your answers in your submission.

Design

- Place the design of your implementation in this item. You must create a design artifact that shows the design of your shell. This should be two levels. A higher-level design shows the overall design of your shell - something that people can understand your design without getting into the details. The high level shows how users interact with your shell. A lower level design shows algorithmic flow and the underlying data structures. You can Search the Internet for design artifacts. In CPSC 240, you used UML to create items such as class diagrams and structure charts.

Features Completed

For each of the following features, indicate which you have designed and implemented in your shell. If you indicate yes for completing a feature, place a screen shot after the bullet showing your tests that demonstrate the feature.

- ❏ Basic reading of a command line and breaking into its parts.
- ❏ `cd` and `pwd` commands
- ❏ `fork()` and `exec()` of command that are programs such as `% cat file.txt`
- ❏ Connect two commands with a pipe such as `% cat file.txt | grep string`
- ❏ Redirect output such as `% cat file.txt > outputfile.txt`
- ❏ Redirect input such as `% grep string < inputfile.txt`
- ❏ Processes Control-D to exit the shell.
- ❏ Ability to run a program in the background such as `% ls &`
- ❏ Processes Control-C such that the shell does not terminate.
- ❏ Implemented your own version of the `ls` program.
- ❏ Implemented a second Linux program such as `cat`.
- ❏ Gusty, the code for my shell is so cool that I want you to read it.

Test Cases

- Place your test cases here. Your collection of test cases should demonstrate you have tested the features you completed. Your test cases should be a copy/paste of you applying your test cases to your program. If the test case is not obvious, you should annotate the copy/paste with descriptions. For example, the following commands are obvious test cases.

```
Gusty % ./teenysh
teenysh0 $ pwd
/Users/gusty/Google Drive/00UMW/16CPSC405/Labs/ShellProject/Gusty
teenysh1 $ ls
a.out       gusty.txt   loop.c      ls.c        sh.c        teenysh.c
class.txt   gustysh.c   ls          out.txt     shellhelp.c test.txt
dir         loop        ls copy.c   out1.txt    skipwhitespace.c
```

```
teenysh2 $ ls | grep loop
loop
loop.c
teenysh3 $ exit
Gusty %
```

Effort Paper

- Place your paper here. Write a paper that (1) describes your effort on the project (include a log of days/hours), (2) describes your difficulties on the project, and (3) describes the features completed in your shell.

Code

- Submit your code and a makefile.