CS3502 Operating Systems

Homework#2

(Implementation of a simple custom UNIX/Linux shell using fork() and exec())

Max Points: 100

You should save/rename this document using the naming convention hw2__yourNETID.docx (example: hw2__eahmed1.docx).
Objective: The objective of this Project exercise is as follows:

- Design a C++ program to serve as a shell interface
- Shell interface will accept user commands and executes each command in a separate process

Note: ssh YourNetID@cs3.kennesaw.edu
Fill in answers to all of the questions.

Name:

Part #1: Implementing a simple Shell

For interprocess communication, we need a few more concepts:

- Pipes: A UNIX *pipe* is a kernel buffer with two file descriptors, one for writing (to put data into the pipe) and one for reading (to pull data out of the pipe), as illustrated in the following Figure 1.

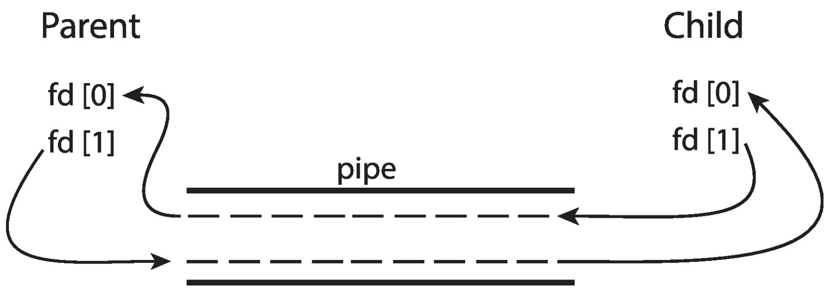

Figure 1: File descriptors for an ordinary pipe

Data is read in exactly the same sequence it is written, but since data is buffered, the execution of the *producer/writer* and *consumer/reader* (another concept, we will elaborate when discussing *Threads and Process Synchronization*) can be decoupled, reducing waiting in the common case. The pipe terminates when either endpoint closes the pipe or exits.

The Internet has a similar facility to UNIX pipes called TCP (Transmission Control Protocol), we will talk about TCP more later on Client-Sever Socket programming. When UNIX pipes connect processes on the same machine, TCP provides a bi-directional pipe between two processes running on different machines. In TCP, data is written as a sequence of bytes on one machine and read out as the same sequence on the other machine.

On UNIX systems, ordinary pipes are constructed using the function

pipe(int fd[])

This function creates a pipe that is accessed through the int fd[] file descriptors: fd[0] is the read-end of the pipe, and fd[1] is the write-end. UNIX treats a pipe as a special type of file (*BTW, UNIX treats everything as a file*). Thus, pipes can be accessed using ordinary read() and write() system calls.

- **Replace file descriptor**. By manipulating the file descriptors of the child process, the shell can cause the child to read its input from, or send its output to, a file or to a pipe instead of from a keyboard or to the screen. This way, the child process does not need to be aware of who is providing or consuming its I/O. The shell does this *redirection* using a special system call named dup2(from, to) that replaces the *to file* descriptor with a copy of the *from file* descriptor.

The dozen UNIX system calls listed in the following Table 1 will be help in designing a simple command line shell, one that will run entirely at user-level with no special permissions.

System call	**Description**
Creating and managing processes	
fork()	Create a child process as a clone of the current process. The fork call returns to both the parent and child
exec(prog, args)	Run the application prog in the current process
exit()	Tell the kernel the current process is complete, and its data structures should be garbage collected
wait(processID)	Pause until the child process has exited
signal(processID, type)	Send an interrupt of a specified type to a process
I/O operations	
fileDesc open(name)	Open a file, channel, or hardware device, specified by name; returns a file descriptor that can be used by other calls

System call	Description
pipe(fileDesc[2])	Create a one-directional pipe for communication between two processes. Pipe returns two file descriptors, one for reading and one for writing.
dup2(fromFileDesc, toFileDesc)	Replace the toFileDesc file descriptor with a copy of fromFileDesc. Used for replacing stdin or stdout or both in a child process before calling exec
int read(fileDesc, buffer, size)	Read up to size bytes into buffer, from the file, channel, or device. Read returns the number of bytes actually read. For streaming devices this will often be less than size. For example, a read from the keyboard device will (normally) return all of its queued bytes.
int write(fileDesc, buffer, size)	Analogous to read, write up to size bytes into kernel output buffer for a file, channel, or device. Write normally returns immediately but may stall if there is no space in the kernel buffer.
close(fileDescriptor)	Tell the kernel that the process is done with this file, channel, or device

Table 1: List of UNIX system calls required to implement a simple shell

Compilation using: g++ -g –Wall –std=c++14 –o NameOfExecutable source.cpp

Note here that source.cpp is a any source file name. You have been given fig3-33.c as the starter code. Save the same file as .cpp extension and use C++ syntax.

Part #1.1: Show/Run the Shell code Fig3-33.cpp as it is described above and attach a screen shot

Part #1.2: Modify the code as follows

Instead of execlp, make use of execvp as described below:

The first task is to modify the main() function in the above code so that a child process is forked and executes the command specified by the user. This will require parsing what the user has entered into separate tokens and storing the tokens in an array of character strings(agrs). For example, if the user enters the command ps –ael at the *mysh>* prompt, the values stored in the args array are:

args[0] = “ps”

args[1] = “-ael”

args[2] = NULL

This args array will be passed to the execvp() function, which has the following prototype

execvp(char *command, char *params[]);

Here, command represents the command to be performed and params stores the parameters for this command. For this part of the Lab #6, the execvp() function should be invoked as execvp(args[0], args).

Another example to run command from shell: **ls –l | wc –l**

char *ls_args[] = {"ls", "-l", NULL}; //for parent process

char *wc_args[] = {"wc", "-l", NULL}; //for child process

Part # 1.3 Outline of a simple shell

```
#include <iostream>
#include <unistd.h>
#define MAXLINE 80 /* The maximum length command */
int main(void)
{
char *args[MAXLINE/2 + 1]; /* command line arguments */
int should_run = 1; /* flag to determine when to exit program */
char *prog = NULL;
char **args = NULL;
while (should_run) {
printf(“mysh”);
fflush(stdout);
/**
*After reading user input, the steps are:
* (1) fork a child process using fork()
* (2) the child process will invoke execvp()
*/
int child_pid = fork();
if (child_pid == 0) { /* I'm the child process, run program with parent's I/O
exec(prog, args);
}
else { /* I'm the parent: wait for the child to complete
wait(child_pid);
return 0;
}
}
return 0;
}
```

The above pseudocode illustrates the basic operation of a shell. The shell reads a command line from the input, and it forks a process to execute that command. UNIX fork automatically duplicates all open file descriptors in the parent, incrementing the kernel's reference counts for those descriptors, so the input and output of the child is the same as the parent. The parent waits for the child to finish before it reads the next command to execute.

Because the commands to read and write to an open file descriptor are the same whether the file descriptor represents a keyboard, screen, file, device, or pipe, UNIX programs do not need to be aware of where their input is coming from, or where their output is going. This is helpful in a number of ways:

- **A program can be a file of commands**.
- **A program can send its output to a file.**
- **A program can read its input from a file.**
- **The output of one program can be the input to another program.**

Show/Run the Shell code CustomShell.cpp as it is described above and attach a screen shot

Part #2: Implement an advanced Shell

Your job is to implement an advanced UNIX/Linux shell in C++ capable for executing a sequence of programs that communicate through a pipe. More specifically, for example, if the user types ls | wc, your program should fork off the two programs, which together will calculate the number of files in the directory.

Our goal for this part of the Lab is two-fold:

- **To exercise your C++/Linux skills**
- **And to get you to think about a clean design that handles the concurrency aspect of the shell**

Because of this, you shouldn't worry about any of the more complex aspects of shells, like supporting command line arguments, environment variables, or other shell features (&, >, etc.).

All you need to do is handle the execution of programs and being able to pipe stdout from one program to the stdin of another. For this, you will need to use several of the Linux system calls described above in Part # 1: fork, exec, open, close, pipe, dup2, and wait. Note: You will have to replace stdin and stdout in the child process with the pipe file descriptors; that is the role of dup2.

Part 2: Specification

1. Read commands to execute from stdin. A command is terminated by a newline character ('\n') and consists of one or more sequence of programs separated by the string " | ".
2. Continue reading and executing commands until stdin returns EOF.
3. Wait for the current command to terminate before starting the next command.
4. The child programs of a command must execute in parallel (*effective use of pipe*, |)
5. Programs can be named by either an absolute path or just by the program name (exec*p should handle this for you).
6. For input “exit”, break the infinite while loop
7. Print an easy-to-parse prompt to stdout

Part 2: Hints

To solve this programming exercise, you'll need to use some (but perhaps not all) of the following Linux system calls: fork, exec*p, close, pipe, dup2, and wait. As a hint, you should use pipe to create a pipe, fork to execute the programs, and dup2 to replace stdin or stdout, and wait to wait for the processes to terminate, probably in that order.

- In case stdin not recognizable in dup2, just can use 0
- In case stdout not recognizable in dup2, just can use 1

You might also want to look at the strtok()/ split function, this will simplify the job of parsing user input in your shell. Following is intended just as a sketch of a solution:

```
#include <unistd.h>

#include <iostream>

int main(int argc, char *argv[]) {

char *prog1 = NULL, *prog2 = NULL;

char **args1 = NULL, **args2 = NULL;

int pipefd[2]; // file descriptors for the pipe
int childpid1 , childpid2 ;
// we omit the details of parsing a pipe command
while (readAndParsePipeCmd(&prog1 , &args1 , &prog2 , &args2 ))
{
// create a pipe with two open file descriptors
// can write to pipefd[1] and read from pipefd[0]
// open file descriptors are inherited across fork
pipe(pipefd);
// Create the first child process
childpid1 = fork ();
if ( childpid1 == 0) {
        // I 'm the first child process.
        // Run prog1 with stdout as the write end of the pipe
        dup2(pipefd [1] , stdout );
        exec(prog1 , args1 );
// NOT REACHED
}
else {
        // I 'm the parent
        // Create the second child process
        childpid2 = fork ();
        if ( childpid2 == 0) {
        // I 'm the second child process.
        // Run prog2 with stdin as the read end of the pipe
        dup2(pipefd [0] , stdin ); exec(prog2 , args2 );
        // NOT REACHED
        } else {
        // I 'm the parent
        wait(childpid2);
        wait(childpid1);
return 0;
```

```
        }
}
} // end while
} // end main
```

Show/Run the Shell code myShell2021.cpp as it is described above and attach a screen shot

Part #3: Submission

- No late assignments will be accepted!
- This work is to be done individually
- The submission file will be saved with the name ***hw2_yourNETID.pdf***
- Assignment is due before Wednesday, September 29, 2021 11:59pm Midnight
- On or before the due time, drop the *electronic copy* of your work in the *canvas*
- Don't forget to Turn in the file! HW2_yourNETID.pdf