CSS 430 Operating Systems
Program 1A: Process Management

0. Attention

The series of C++ programming assignments 1A, 2A, 3A, and 4A intends to familiarize you with Linux system programming through invoking Linux OS functions directly from your C++ programs. **While you may develop your programs with any IDE or on any OS, you must compile and run the code on the in-person or remote CSS Linux lab machines.**

1. Purpose

This assignment is designed to introduce how to manage processes: creating a child process, executing a new program in the child process, communicating with it, remapping this communication channel to a different file descriptor, and waiting for the child to finish.
Each operation is performed with ***fork, execlp, pipe, dup2,*** **and** ***wait,*** respectively.

2. Linux System Calls

The following table summarizes Linux system functions to manage process execution.

Linux System Functions	**Actions**
pid_t fork( )	Creates a child process. The parent process receives the child process ID, while the child process receives 0 as a return value. The child process starts right after fork( ).
int execlp(const char *path, const char *arg, ...)	Overwrites the calling process image with a new program located at path and starts its execution from main( ) as passing *arg* to it. No return on success.
pid_t wait(int *status);	Waits for a child process to complete its execution and returns the child process ID. The parent receives in status the return value from the child's main( ) or exit( ) is.
int pipe(int pipefd[2])	Creates a pipe, a unidirectional data channel. pipefd[0] refers to the read end of the pipe, whereas pipefd[1] referr to write end of the pipe. Zero is returned on success.
int dup2(int oldfd, int newfd)	Makes newfd be the copy of oldfd. This is frequently used to map a pipe's read/write ends to standard input/output. For example, dup(pipefd[0], 0) and dup(pipfd[1], 1).
int close(int fd)	Close a file descriptor. Use it for closing unnecessary pipe ends.

The shell utilizes these system functions internally for executing commands and programs given by a user.

3. Shell – Read this section carefully to understand how Shell works!

The *shell* is a **command interpreter** that provides Unix/Linux users with an interface to the underlying operating system. It interprets user commands and executes them as independent processes. The *shell* also allows users to code an interpretive script using simple programming constructs such as if, while, and for. With *shell* scripting, users can coordinate a scenario of their jobs.

3.1. Command Interpretation

The behavior of shell simply repeats:

1) Displaying a prompt to indicate that it is ready to accept a next command from a user,
2) Reading a line of keyboard input as a command, and
3) Spawning and having a new child process execute the user command.

How does the *shell* execute a user command? The mechanism follows the steps give below:

1) The *shell* locates an executable file whose name is specified in the firs string give from a keyboard input,
2) It creates a child process by duplicating itself (through *fork*),
3) **The duplicated child shell overloads its process image with the executable file** (through *execpl*),
4) The overloaded process receives all the remaining strings given from a keyboard input and starts a command execution.

For instance, assume that your current working directory includes the *a.out* executable file and you have typed the following line input from your keyboard.

```
csslabXXX$ a.out a b c
```

This means that the *shell* duplicates itself and has this duplicated process execute a.out that receives a, b, and c as its arguments.

The *shell* has some built-in commands that rather change its status than execute a user program. (Note that user programs are distinguished as external commands from the *shell* built-in commands.) For instance,

```
csslabXXX$ cd public_html
```

changes the *shell*'s current working directory to public_html. Thus, *cd* is one of the *shell* built-in commands.

The *shell* can receive two or more commands at a time from keyboard input. The symbols ';' and '&' are used as a delimiter specifying the end of each single command. If a command is delimited by ';', the *shell* spawns a new process to execute this command (by *fork* and *execlp*), waits for the process to be terminated (by *wait*), and thereafter continues to interpret the next command. If a command is delimited by '&', the *shell* goes forward and interprets the next command without waiting for the completion of the current command.

```
csslabXXX$ who & ls & date
```

executes *who*, *ls*, and *date* **concurrently**. Note that, if the last command does not have a delimiter, the *shell* assumes that it is implicitly delimited by ';'. In the above example, the *shell* waits for the completion of *date*. Taking everything in consideration, the more specific behavior of the *shell* is therefore:

1) Displaying a prompt to show that it is ready to accept a new line input from the keyboard,
2) Reading a keyboard input,
3) Repeating the following interpretation till reaching the end of input line:
    - Changing its current working status if a command is built-in, otherwise
    - Spawning a new process and having it execute this external command.
    - Waiting for the process to be terminated if the command is delimited by ';'.

3.2. I/O Redirection and Pipeline

One of the *shell*'s interesting features is I/O redirection.

```
csslabXXX$ a.out < file1 > file2
```

redirects *a.out*'s standard input and output to *file1* and *file2* respectively, so that *a.out* reads from *file1* and writes to *file2* as if it were reading from a keyboard and printing out to a monitor.

Another convenient feature is pipeline.

```
csslabXXX$ command1 | command2 | command3
```

connects *command1*'s standard output to *command2*'s standard input and connects *command2*'s standard output to *command3*'s standard input.

For instance,

```
csslabXXX$ who | wc -l
```

firstly executes the *who* command that prints out a list of current users. This output is not displayed but is rather passed to *wc*'s standard input. Thereafter, the *wc* is executed with its *-l* option. It reads the list of current users and prints out #lines of this list to the standard output. As a result, you will get the number of current users.

3.3. Shell Implementation Techniques

Whenever the *shell* executes a new command, it spawns a child *shell* and lets the child execute the command. This behavior is implemented with the **fork** and **execlp** system calls.

- If the *shell* receives ";" as a command delimiter or receives no delimiter, it must wait for the termination of the spawned child, which is implemented with the **wait** system call.
- If it receives "&" as a command delimiter, it does not have to wait for the child to be terminated.
- If the *shell* receives a sequence of commands, each concatenated with "|", it must **recursively create children** whose number is the same as that of commands. **Which child executes which command is tricky. The farthest offspring (e.g., the great grandchild) will execute the first command, the grand child will execute the 2nd last, and the child will execute the last command.** Their standard input and output must be redirected accordingly using the **pipe** and **dup2** system calls. The following pictures describe how to execute "who | wc -l", and how to use the **pipe** system call.

The following pictures show how the Shell command “who | wc -l” is executed and how the pipe is used between parent and child processes.

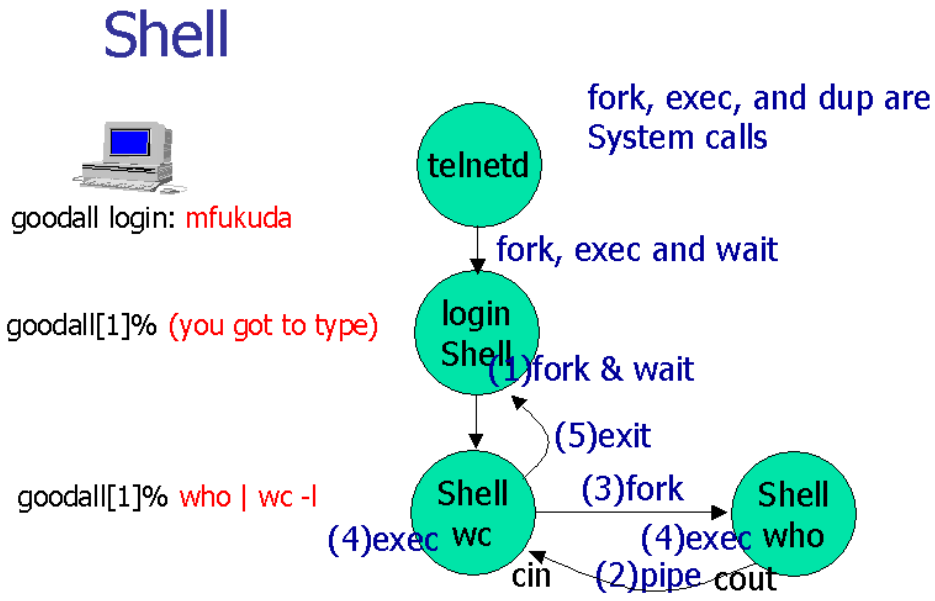

Figure 1: Execution flow of "who | wc -l" in Shell

```
int main( void ) {
  int n, fd[2];
  int pid;
  char line[MAXLINE];

  if (pipe(fd) < 0 )      // 1: pipe created
    perror( "pipe error" );
  else if ( (pid = fork( ) ) < 0 )     // 2: child forked
    perror( "fork error" );
  else if ( pid > 0 ) {   // parent
    close( fd[0] );       // 3: parent's fd[0] closed
    write( fd[1], "hello world\n", 12 );
  } else {                // child
    close( fd[1] );       // 4: child's fd[1] closed
    n = read( fd[0], line, MAXLINE );
    write( 1, line, n );
  }
  exit( 0 );
}
```

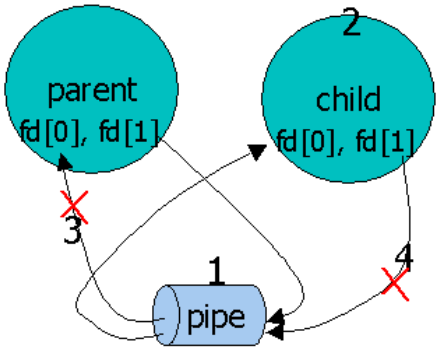

Figure 2: Open and close pipe descriptors for correct parent-child communication

4. Statement of Work

Task 1: Complete *processes.cpp*.
You can find this file from Canvas Files: Files/code/prog1.

This program receives one argument, (i.e., argv[1]) upon its invocation and searches how many processes whose name is given in argv[1] are running on the system where your program has been invoked. To be specific, your program should demonstrate the same behavior as:

```
ps -A | grep argv[1] | wc -l
```

Implement *processes.cpp* using the system functions listed above in Section 2: Linux System Calls.

For more details, type the **man** command from the *shell* prompt line. Use only the system calls listed in Section 2. You can also use "exit () " Imitate how the *shell* performs "ps -A | grep argv[1] | wc -l". In other words, your parent process spawns a child that spawns a grand-child that spawns a great-grand-child. Each process should execute a different command as follows:

Process	Command	Standard in (fd 0)	Standard out (fd 1)
Parent: starts main( )	wait for a child to complete	no change	no change
Child	wc -l	dup2(the read end of the pipe shared with grandchild, 0)	no change
Grandchild	grep argv[1]	dup2(the read end of the pipe shared with great grandchild, 0)	dup2(the write end of the pipe shared with child, 1)
Great grandchild	ps -A	no change	dup2(the write end of the pipe shared with grandchild, 1)

To verify the correctness of your processes.cpp, compare the following two executions:

csslabXXX$ ps -A | grep ssh | wc -l
10
csslabXXX$ g++ processes.cpp
csslabXXX$ ./a.out ssh
10
commands completed
csslabXXX$

In "**./a.out ssh**", "ssh" is the parameter which is the same one used in "**ps -A | grep ssh | wc -l**".

The lines of code (LoC) for this implementation would be less than 40 lines (excluding comments).

Note: When executing your program "**./a.out ssh**", you may get a number different from 10, which is very common! Your program should obtain the same number as of "**ps -A | grep ssh | wc -l**" on the same machine!

Task 2: Capture the execution outputs.
Task 3: Illustrate all these processes and their pipe connections involved in *processes.cpp*.

5. What to Turn in

	Materials	Points	Note
1	processes.cpp a. Code organization: **+2pts** • Well organized: 2pts, • Poor comments or bad organization: 1pt, or • No comments and horrible code: 0pts b. Correctness: **+10pts** • A parent waits for a child: **+1pt,** • A child executes "wc -l": **+3pts,** • A grandchild executes "grep argv[1]": **+3pts, and** • A great-grand-child executes "ps -A": **+3pts**	12	Submit this cpp file individually
2	An execution snapshot. **The snapshot must also clearly show your username and the Linux lab machine name.** • The same output as ps -A | grep argv[1] | wc -l: 4pts, • The different output from ps -A | grep argv[1] | wc -l: 2pts, or • No outputs: 0pts	4	All snapshots MUST clearly show the lab machine name and the username! DO NOT submit a snapshot with the result only! Include these two items in a pdf file named: FirstNameLastName_prog1A.pdf. Submit this pdf file individually.
	An illustration of all processes and their pipe connections involved in processes.cpp. You should show: (1) How are processes (parent, child, grandchild, great grant child, etc.) connected? (2) The name of each process. (3) How is each pipe connected (input and output)?and (4) What file descriptors are opened or closed? • Correct: 4pts • Wrong: 2pts, or • No illustrations: 0pts	4	