CptS 370
Program 1: System Calls in C/++

Due Date: Friday, 18 Sep 2020, 11:59 p.m.

---

0. Attention

This series of programming assignments is a step-by-step implementation of an OS simulator in Java. This first assignment, however, is more focused on Linux system programming and therefore entails coding in C/C++. Please note that you must work independently on all programming assignments. The instructor will never tolerate any academic dishonesty. Please make good use of lab hours if you find yourself struggling with this assignment.

1. Purpose

This assignment intends (1) to familiarize you with Linux programming using several system calls such as fork, execlp, wait, pipe, dup2, and close, and (2) to help you understand that, from the kernel's view point, the *shell* is simply viewed as an application program that uses system calls to spawn and to terminate other user programs. You will become further familiar with the shell concept via the *ThreadOS* operating system simulator in Assignment 2.

2. Shell

This section explains the behavior and the language syntax of the Unix/Linux *shell*.

Command interpretation

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*, etc. With *shell* scripting, users can create customized automation shell tasks.

The behavior of *shell* simply repeats:

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

The prompt symbols frequently used include for example:

```
hostname$
```

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

1. The *shell* locates an executable file whose name is specified in the first string given from a keyboard input.
2. It creates a child process by duplicating itself.

3. The duplicated shell overloads its process image with the executable file.
4. The overloaded process receives all the remaining strings given from the keyboard input as argument and starts the 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.

```
hostname$ ./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 changes its current status rather than executing a user program. (Note that user programs are distinguished as external commands from the *shell* built-in commands.) For instance,

```
hostname$ cd public
```

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

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

```
hostname$ 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

      or

    - Spawning a new process and having it execute this external command
    - Waiting for the process to be terminated if the command is delimited by ';'.

I/O redirection and pipeline

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

```
hostname$ 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 convenience feature is pipeline.

```
hostname$ command1 | command2 | command3
```

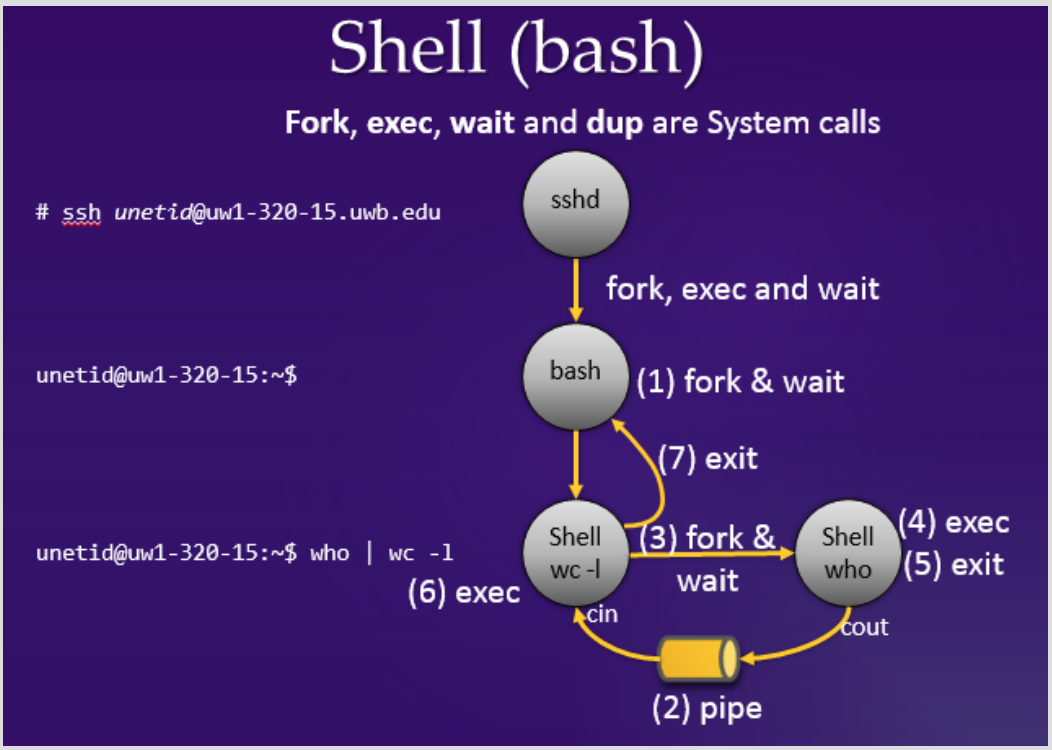
connects *command1*'s standard output to *command2*'s standard input, and also connects *command2*'s standard output to *command3*'s standard input. For instance,

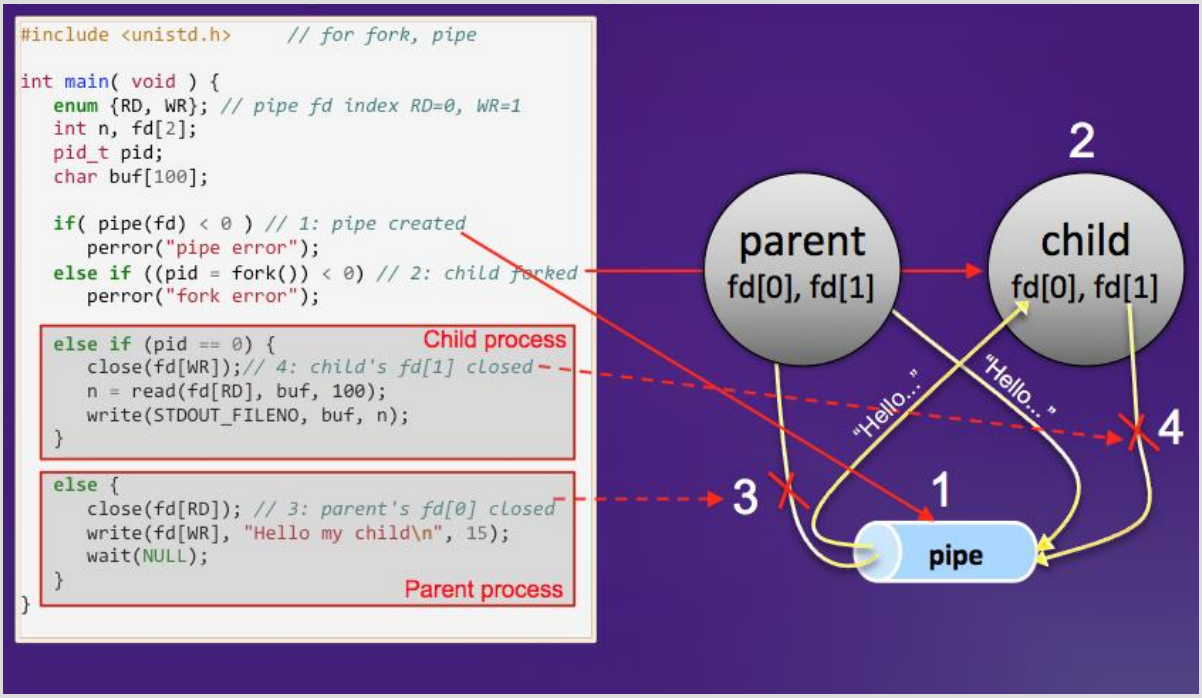
```
hostname$ who | wc -l
```

first 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. Next, *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 the current users.

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 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 kind of tricky. For three piped commands, the farthest offspring will execute the first command, the grand child will execute the 2nd, 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 diagrams describe how to execute "who | wc -l", and how to use the **pipe** system call.





3. Statement of Work

Linux System Programming

Code a C++ program, named *processes.cpp* that 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* using the following system calls:

1. **pid_t fork(void);** creates a child process that differs from the parent process only in terms of their process IDs.
2. **int execlp(const char *file, const char *arg, ..., (char *)0);** replaces the current process image with a new process image that will be loaded from *file*. The first argument *arg* must be the same as *file*.
3. **int pipe(int filedes[2]);** creates a pair of file descriptors (which point to a pipe structure), and places them in the array pointed to by filedes. *filedes[0]* is for reading data from the pipe, *filedes[1]* is for writing data to the pipe.
4. **int dup2(int oldfd, int newfd);** creates in newfd a copy of the file descriptor oldfd.
5. **pid_t wait(int *status);** waits for process termination
6. **int close(int fd);** closes a file descriptor.

For more details, type the **man** command from the *shell* prompt line. Use only the system calls listed above. Do not use the **system** system call. 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	Stdin	Stdout
Parent	wait for a child	no change	no change
Child	wc -l	redirected from a grand-child's stdout	no change
Grand-child	grep argv[1]	redirected from a great-grand-child's stdout	redirected to a child's stdin
Great-grand-child	ps -A	no change	redirected to a grand-child's stdin

4. What to Turn In

Program 1 must include:

1. Softcopy:
   processes.cpp
2. A sample of output when running your program vs. actual system calls:

```
$ processes kworker
$ ps -A | grep kworker | wc -l

$ processes sshd
$ ps -A | grep sshd | wc -l

$ processes scsi
$ ps -A | grep scsi | wc -l
```

3. Report:
   Explain the algorithm of your processes.cpp in brief, using flowcharts, plain language, or other method as appropriate.

5. FAQ

There is a FAQ document for Assignment 1 saved on Blackboard.