Report Documentation

Building files:
To build *processes.cpp* simply use the following command:
`g++ processes.cpp -o processes`

To build Shell.java simply use the following command:
`javac Shell.java`

General Overview:
- Create a *processes.cpp* that emulates a specified linux command, in order to learn about *fork()* and *pipe()* usage as well as *dup2()* usage.
- Create *Shell.java* that emulates the default *Shell.class* from the original ThreadOS. It should be able to load specified programs such as *PingPong.java* and take in a variable amount of arguments which may be separated by varying whitespace between commands and delimiters.

Program 1 Part 1: *processes.cpp*
This program is created with C++ and emulates a command line or terminal in interpretation of commands (specifically a Unix based command line). It seeks to create a process which then creates multiple processes underneath to carry out the necessary commands.

For each process that is created, *fork()* is used, returning the PID of the process in the making. The method *pipe()* is also used to create necessary pipes to pipe information from one command to another. For each pipe, an array of pipeFD (pipe file descriptor) is had for the *READ* and *WRITE* states present within a pipe. The function *execlp()* is used to execute the command that is passed in. The function *dup2()* is used to duplicate or overwrite the file descriptors to pass information between processes with pipes (either through *stdout* or *stdin* which are *READ* or *WRITE* states respectively). In my particular code, I used an *enum{READ,WRITE}* to make it easier to output to *stdout* or *stdin*.

For the command that is currently implemented in *processes.cpp* we have:
"ps -A | grep argv[1] | wc -l"
Which translates to: Show current process status with an all parameter flag, piped to a pattern search based on a keyword, piped to a word count with a long parameter flag. The actual `grep` pattern to search for isn't quite `argv[1]` but instead any process that is shown in "`ps -A`".

To implement these commands, we will have to use 3 *fork()* and 2 *pipe()* to execute the first command all the way to the third command. As *fork()* is done in a binary tree structure it will have to call first child then go all the way down to the bottom most child.

Program 1 Part 1 Algorithm:
To implement these 3 commands, I used 3 *fork()* functions and 2 *pipe()* functions to do so. This is to have 3 processes to be run, and to have information pass between the processes which is done so by the pipes.

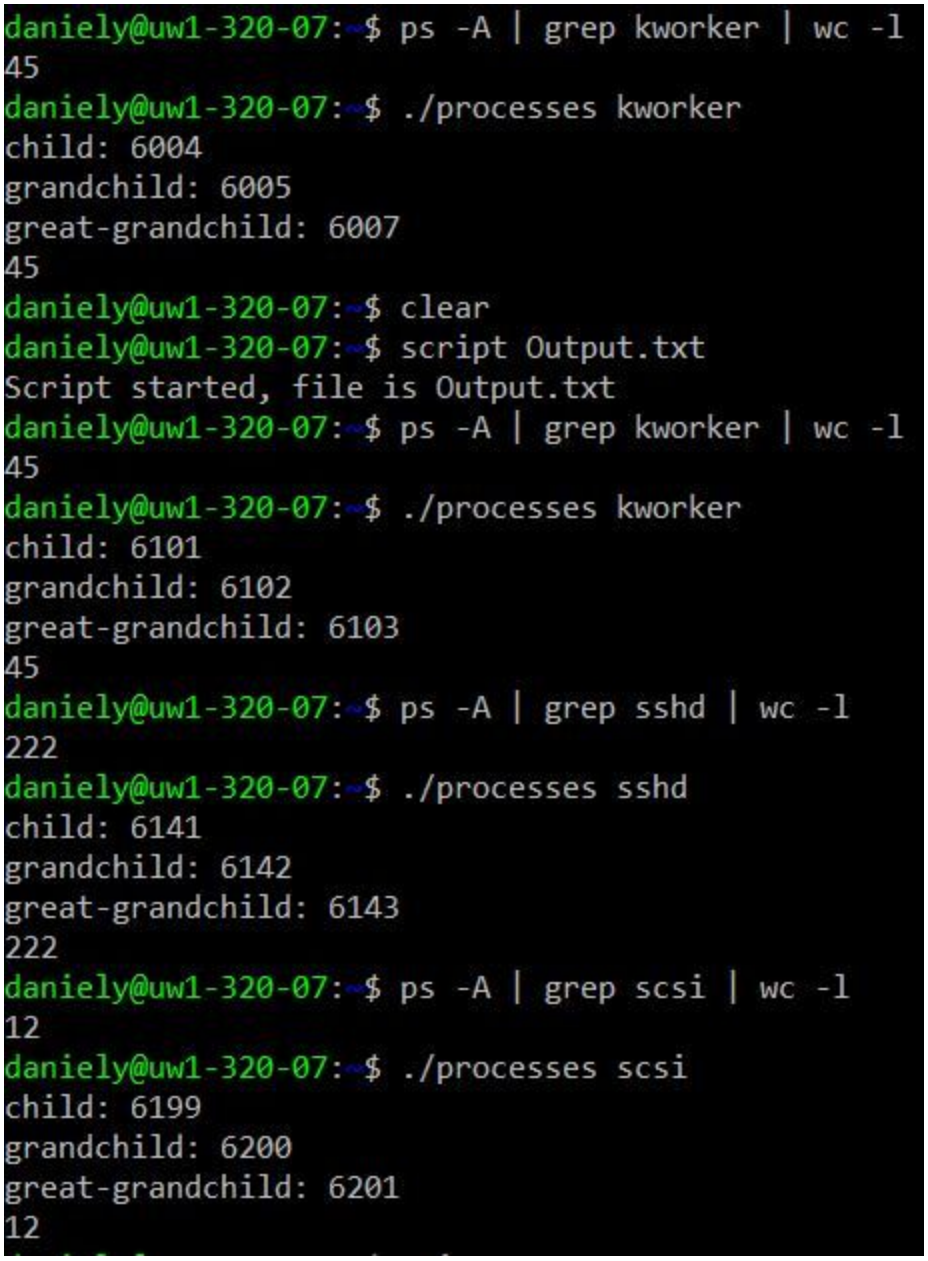
To start, the first *fork()* simply creates the parent and child. The second *fork()* creates the grandchild, and the third *fork()* creates the great-grandchild. This also lays down the hierarchy the processes are to run. As stated before *fork()* is done in a binary tree structure, as such the child will be the one that will need to be run first, then the grandchild, then the great-grandchild, and finally the parent.

As such we would then simply use *execlp()* three times for the three commands. We would have "`ps -A`" in child, then "`grep argv[1]`" in grandchild, then "`wc -l`" in great-grandchild. The parent doesn't need to do anything but simply *wait()* for all child processes to finish.

Once the command hierarchy was established, then the reading and writing between them, or piping, needs to be done as well. Judging from the command order, the first command is piped to the second, the second is piped to the third. From this we can determine that the first command only needs to *WRITE* to the second command. While the second command needs to *READ* from the first command then *WRITE* to the third command. The third command just needs to *READ* from the second command. Here we can then make the pipe algorithm, where we would need 2 pipes to solve these reading and writing connections. The first pipe would be used to *WRITE* to the second pipe, which then does *READ* from the first pipe and *WRITE*. The third command then just does *READ* on the second pipe.

Testing for the program is shown below and in the accompanied Output.txt which has the associating script output. For testing, code such as `std::cout << "child: " << getpid() << std::endl` was utilized in order to print out the PID of each process. This can be seen in the accompanied screenshot as well as in the text script file. Testing was done on Linux Lab machines as "`ps -A`" would return different processes on home machine.

Program 1 Part 1 Testing:



Program Part 2: Shell.java

This program is created with Java and aims to emulate a Shell for the ThreadOS that was provided. The features that are implemented are based off of the default Shell.class that was provided in the demonstration for ThreadOS. Shell.java will load valid applications and execute them in either sequential (as denoted by ";") or concurrently (as denoted by "&").

Program Part 2 Algorithm:

After loading a DISK with ThreadOS and loading simple programs to check, Shell.java should be loaded with the following commands:

```
l Shell.java
```

Which is executed inside of the ThreadOS, bringing out the Shell command prompt. Given the Shell.java program is not incredibly large, everything is essentially contained into a single method, *run()*, which is a necessary interface method extended from Thread. The *Shell()* constructor is simply empty.

Inside of *run()*, variable initialization is first done with an `int runCount` and `HashSet currentProcesses`. `runCount` is made to count the number of times run() is used, to denote how many Shell processes are run. The HashSet is an unordered set to simply remember processes for concurrent running. After an infinite while loop is set to continually run the shell until "exit" is entered. Inside the while loop a split is done to split commands between the delimiter of ";" for sequential running, then for each command that is split, another for each loop is done to then run commands with the "&" delimiter. These two for each loops are done to run these commands in different manners. If the "&" for each loop is run, the processes are executing without *SysLib.join()* being utilized, as they're to be run concurrently. If the ";" for each loop is run, the processes are run with *SysLib,join()* to run sequentially. The HashSet is used to store concurrent processes, and remove them when done.

Testing for the program was done with the following command:
PingPong abc 30 ; PingPong xyz 20 ; PingPong 123 30
PingPong abc 30   ; PingPong xyz 20 & PingPong 123 30
PingPong abc 30 & PingPong xyz 20 ; PingPong 123 30
PingPong abc 30   & PingPong xyz 20 & PingPong 123 30
Where it first tests 3 simple sequential commands, then it tests 1 sequential with 1 concurrent, then it tests the same with the reversed order, and finally 3 concurrent processes test. Slight variations in spacing is done. This is shown in the testing pictures below and in the ShellText.txt script that was generated from the testing.
.

Program 1 Part 2 Testing: