DUE BEFORE: **End of your lab session**

CSE-381: Systems 2

Exercise #3

Max Points: 20

Prior to proceeding with the exercise, you should first save/rename this document using the naming convention `MUid_Exercise3.docx` (example: `raodm_Exercise3.docx`).

Objective: The objective of this exercise is to:

- Learn to create a new process using the `fork` system call.
- Run a different program using `fork` and `execvp` system calls.

Submission: Once you have completed this exercise, upload:

- This MS-Word document saved as a PDF file with the naming convention **`MUid_Exercise3.pdf`**.
- The C++ source program developed in Part 2 of this exercise (to run a different program) and named using the convention ***`MUid`***`_ex3_2.cpp` (example: `raodm_ex3_2.cpp`).

You may discuss the questions with your neighbors, TAs, or your instructor.

Name: **Noah Dunn**

First wait for your instructor to play the following video https://youtu.be/nwm7rJG90i8 to introduce some of the concepts associated with the `fork` and `exec` system calls used in this exercise. Once your instructor has covered the concepts then you can proceed with this lab exercise.

Part #1: Explore the operations of the `fork()` system call

Background: The `fork()` system call provides a mechanism to clone a running process. In Unix and Linux, `fork` is the mechanism (and only available mechanism) that is used to create new processes and run other programs. When a process forks a new child process is created and both parent and child processes are identical copies of each other and resume operation right after the call to `fork()`. The only difference is that the return value of `fork` is different in the parent and child. The child and parent have exactly the same virtual memory layout. However, they are mapped to different physical addresses so they are different processes!

Exercise: This exercise requires you to explore the results of forking a process and record various observations as directed below. Note that you do not need to use NetBeans for this part of the exercise.

1. Download the source code, named ex3_1.cpp and scp it to the Linux server.
2. View the source code and study the program to note the following:
   a. Note the loop in read() method to read data from a given input file stream and print number of characters read. Both parent and child processes call this method but with the same input stream (created in main). This causes parent and child to share the same input stream causing them to read different subset of data from the same file (due to context switching where processes take turns to run)
   b. Now note how the main() method:
      i. Opens the data file for reading.
      ii. Forks to create a new child process
      iii. Note that, both parent and child processes end-up reading data from the same file (passed to the read method) opened by the parent because the file streams are cloned by fork. **If the file was opened after the fork, then the streams will not be shared**.
      iv. Finally, the parent process waits for the child to finish using the waitpid method.
3. Compile the program on the Linux server as shown below:

```
$ g++ -g -Wall -std=c++14 ex3_1.cpp -o ex3_1
```

4. Run the program three times and copy-paste the output in the space below:
   a. Output from Run #1:

```
In parent: Pid = 30555, address = 0x7ffead688f58
In child: Pid = 0, address = 0x7ffead688f58
child: Number of characters read = 63267437
parent: Number of characters read = 63234520
The parent is waiting for child pid: 30555
Parent is done. Child process's exit code: 0
```

   b. Output from Run #2:

```
In parent: Pid = 30559, address = 0x7ffd2a2e1cf8
In child: Pid = 0, address = 0x7ffd2a2e1cf8
parent: Number of characters read = 63488441
child: Number of characters read = 63013516
The parent is waiting for child pid: 30559
Parent is done. Child process's exit code: 0
```

   c. Output from Run #3:

```
In parent: Pid = 30568, address = 0x7fff431bc758
In child: Pid = 0, address = 0x7fff431bc758
child: Number of characters read = 63111808
```

```
parent: Number of characters read = 63390149
The parent is waiting for child pid: 30568
Parent is done. Child process's exit code: 0
```

5. Let's analyze each aspect of the output
   a. What are the values for `pid` printed in the parent process (hint: it will never be zero)? What do you think these values signify?

      `Pid = 30555, Pid = 30559, Pid = 30568`
      As more processes are run, pid increases as it represents how many processes have been started since reboot. It is a unique identifier.

   b. What are the values for `pid` printed in the child process (hint: it will always be zero)? What do you think these values signify?

      Pid = 0, Pid = 0, Pid = 0
      Each of these are the first child processes generated to each of the parent processes, which is why they are all 0s.

   c. How can we use the return value of `fork` system call to distinguish between parent and child process after the `fork`?

      We can check if the value is 0, or anything but 0. If the value is 0, the process can be identified as the child. For anything else it must be the parent.

6. Let's now study the address for the `pid` variable printed by the parent and child processes. Notice that the memory address for `pid` variable is exactly the same in both parent and child processes, but the values stored in them are different! What? How is that possible? How can two variables in two different processes have exactly the same memory address but store completely different values?

   Although these addresses are equivalent in virtual memory, how the compiler chooses to assign the values in actual memory will be different, thus we do not have any sort of collision even if it appears as though there is one.

7. Let's do some validation of I/O operations by tallying up the number of characters read by the two processes
   a. Record the number of characters read by each process and the total number of characters read in the table below:

Run	**Number of characters read**

	Parent	**Child**	**Total (parent + child)**
1	63234520	63267437	126501957
2	63488441	63013516	126501957
3	63390149	63111808	126501957

b. Why are the number of characters read by parent and child processes different?

The parent and child processes take turns reading lines. Some lines will possess more/less characters than others depending on which process starts and which lines are read by process.

c. How come the total number of characters read is always the same and how can you verify that the total is correct?

Both parent and child processes are running in the same stream. The number of characters will be read until the end, and then both processes will stop. You can check this with a single program/process that reads the whole file char by char in order to check if they are the same.

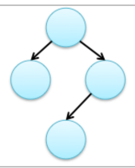
8. How is the exit code of the child process being obtained and printed?

The waitpid method takes an exit code as parameter, sets the value to the exit code inside of waitpid(), and then is printed outside in the main method body.

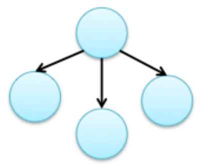
Part #2: Practice visualizing operation of fork

Background: Recollect that `fork` system call creates a clone of the exiting process and both parent and child continue running right after the `fork` system call returns. Using the return value of fork (*i.e.*, zero in child, and non-zero in parent), different process hierarchies can be constructed (using if-else statements appropriately).

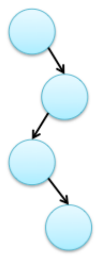
Convert the following process trees to corresponding C++ source code -- that is, the C++ source is implemented to create the process hierarchy shown in the figure. The first question has been completed to illustrate an example.



```
int main() {
    fork(); // First fork only in parent
    fork(); // Parent and child both fork
    return 0;
}
```



```
int main(){
   if(fork() !=0){
      if(fork() != 0){
         fork();
      }
   }
}
```



```
int main(){
if(fork() == 0){
      if(fork() == 0){
         fork();
      }
   }
}
```

Part #3: Run a `find` command using `execvp` method

Background: The `fork` system call creates a new process that is an identical clone of the parent process. In other words, `fork` does not enable running a completely different program. The task of running a completely different program is performed by the `execvp` system call. The `execvp` system call is one in a family of `exec` system calls.

Exercise: This exercise requires you to run the following find command by completing the program in the supplied starter code:

```
ls -l /usr
```

1. Run the above ls command in a terminal to see the expected output from this program.

2. Create a `NetBeans` project with the name `MUid_ex3_2.cpp` and copy-paste the supplied starter code in `ex3_2.cpp` to your source file. The starter code is bare bones but has all the necessary `#includes`

3. Using the examples from the slides copy-paste the `myExec` method that provides a convenient mechanism to execute a program.

4. Next, complete the `main` function run the specified command in the child process, and wait for the process to finish (in the parent).
   - Do not forget to use `waitpid` to wait for the child process to finish.

5. Run your program and copy-paste your output into the space below (your output should be identical to the output observed in step 1 in this part of the exercise):

```
 total 156
drwxr-xr-x   2 root root 69632 Aug 27 11:44 bin
drwxr-xr-x   2 root root  4096 Aug 20  2018 games
drwxr-xr-x  80 root root 20480 Jul 11 14:33 include
drwxr-xr-x  97 root root 20480 Aug 27 11:44 lib
drwxr-xr-x   3 root root  4096 Jul 11 14:34 lib32
drwxr-xr-x   3 root root  4096 Jul 11 14:34 libx32
drwxr-xr-x  13 root root  4096 Apr 24  2017 local
drwxr-xr-x   2 root root 12288 Aug 27 11:43 sbin
drwxr-xr-x 203 root root 12288 Jul 11 14:34 share
drwxr-xr-x  13 root root  4096 Sep 11 09:51 src
```

Submit files to Canvas

Upload just the following files to Canvas:

- Upload this document (duly filled with the necessary information) saved as a PDF file using the naming convention `MUid_Exercise3.pdf`.
- The C++ source program developed in Part 2 of this exercise (to run a different program) and named using the convention ***`MUid`***`_ex3_2.cpp` (example: `raodm_ex3_2.cpp`).

Upload each file individually onto Canvas. Do not upload zip/7zip/tar/gz or other archive file formats for your submission.