DUE BEFORE: **Deadline set on Canvas**

CSE-381: Systems 2

Exercise #4

Max Points: 15

Prior to proceeding with the exercise, you should first save/rename this document using the naming convention `MUid_Exercise4.docx` (example: `raodm_Exercise4.docx`).

Objective: The objective of this part of the exercise is to:

- Review the operations of `fork` system call.
- Run different programs using `fork` and `execvp/execlp` system calls.
- Review object-oriented programming in C++

Submission: Once you have completed this exercise, upload:

- This MS-Word document saved as a PDF file with the naming convention `MUid_Exercise4.pdf`.
- The C++ source files you modified in the different parts of this exercise.

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

Name: **RAObot**

It may be helpful to recollect concepts regarding `execlp` and `execvp` system calls by reviewing the following video https://youtu.be/7SZdWV1fLxA?t=582 (starting at the 9:40 minute mark) to recap the concepts associated with the `exec` family of syscalls used in this part of the exercise. Once you have reviewed the video then proceed with this exercise.

Part #1: Understanding the basics of `execvp` and `execlp` syscalls

Background: 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). The `execvp` system call requires the following 2 arguments:

1. The first argument is the name or path of the program to be executed
2. An array of C-strings for command-line arguments. Note: As per convention, the 1st command-line argument should always be exactly the same as the name/path of the program being executed.

Exercise: In this part of the exercise, we are going to build some strength on the basic understanding of the `execvp` system call:

1. What should be the first command-line argument passed as part of `execvp` system call? why?

   The first command-line argument should always be exactly the same as the command being executed. This is the convention.

2. Upon successfully running a given program, what is the return value of the `exec` system call?

   The `execlp` and `execvp` systems calls do not ever return after successfully starting to run another program. That is because our program is no longer running and is replaced with the newly start program. In other words, our program ceases to exist. Hence, there is nowhere for `execvp` or `execlp` to return to.

3. Why do we need to typically `fork` before using `exec`?

   The `exec` system call replaces the current process. So, if we don't `fork` first, then our program will not be running anymore after `exec`, that is, there not be any parent process! So, we will not be able to detect if the child process finished successfully and do operations. So, we always need to `fork`, check return value of `fork`, and call `exec` in the child process.

Exit code questions

4. What is exit code?

   In all operating systems, each process must (and does) return a value back to the OS indicating if the program ran successfully. This value is called the exit code. In C/C++ programs, the return value of the `main()` function is the exit code. In other languages, like Java, methods like `System.exit(exitCode);` is used to return the exit code.

5. How is the exit code value meant to be interpreted?

   Exit code is a number. A zero-exit code value indicates the program ran successfully. A non-zero exit code value indicates that the program had some runtime error.

6. What system call can a parent process obtain the exit code of a child process? **Show an example code fragment**.

   The `waitpid` system call's 2nd argument can be used to obtain the exit code of a child process as shown below:

   ```
   int childPid;  // Assume this variable has child-process's PID
   int exitCode = -1;
   // Wait until child process finishes & get exit code
   waitpid(childPid, &exitCode, 0);
   ```

Question on myExec helper method

7. What was the signature (name and parameters) of the helper method that was suggested to ease using the `execvp` system call (Hint: see: 16 minutes into the video)? What was the caution discussed about the helper method?

   The suggested helper method is `myExec(std::vector<std::string> args);` The caution discussed was that this method is designed to take a mutable list of command-line arguments so that they can be passed to the OS via the `execvp` system call. It is best not to modify this method but just copy-paste and use it.

Part #2: Programming with `execvp` method with C++ classes

Background: Object oriented programming provides a convenient approach to encapsulate information and provide controlled access to it. As a convention in C++, we use a split design for classes –

- The header (`.h`) file: The API of the class is defined in a header file (so it can be included in other source files).
- The source (`.cpp`) file: The methods in the class are implemented a separate source file.

Exercise: In this exercise you will be completing a given C++ class to streamline forking and running programs.

1. Start `NetBeans` and create a new Miami University C++ Project, **without a main file**, named `exercise4` on `os1.csi.miamioh.edu`.
2. Download the starter code, named `ChildProcess.h`, `ChlidProcess.cpp`, `serial_vs_parallel.cpp`, `main.cpp`, and `data.txt`. `scp` them it to the `exercise4` project directory on the Linux server.

   ```
   $ cd ~/Downloads
   $ scp ChildProcess.h ChildProcess.cpp serial_vs_parallel.cpp
   main.cpp data.txt
   MUID@os1.csi.miamioh.edu:~/NetBeansProjects/exercise4
   ```

3. Add all 4 files to your project. However, the starter code it will not compile yet as some methods are not yet implemented.

4. Briefly review the methods declared in `ChildProcess.h`.
5. Using the Javadoc/doxygen style comments in the header file, implement the methods in the source (`ChildProcess.cpp`) file. **Note**: You don't need to worry about `serial_vs_parallel.cpp` in this part.
6. Once you have correctly implemented the methods in `ChildProcess.cpp`, you will be able to run the program.
7. **Expected output from the program** (1 successful command with zero exit code and 1 unsuccessful program with non-zero exit code) is shown below:

```
__     __            _____            _    
\ \   / /           |  __ \          | |   
 \ \_/ /__  _   _   | |__) |___   ___| | __
  \   / _ \| | | |  |  _  // _ \ / __| |/ /
   | | (_) | |_| |  | | \ \ (_) | (__|   < 
   |_|\___/ \__,_|  |_|  \_\___/ \___|_|\_\

Exit code: 0
ls: cannot access '/blah': No such file or directory
Exit code: 512
Calling runProcesses()...
runProcesses() completed.
```

8. Let's ensure we are paying attention to API of the `int ChildProcess:wait() const` method
   a. What does the `const` at the end of this method indicate – as in what can/cannot this method do?

      Constant methods are methods can be called on both constant and non-constant objects. A constant method guarantees that it will never directly or indirectly (by calling other non-const methods) modify instance variables in the object, thereby ensuring the object is not mutated (*i.e.*, changed/modified).

      No, other languages like Java, Python, C# etc. do not have such a robust `const` methods – thereby preventing programmers from developing good APIs and robust solutions.

   b. Show an example of a C++ statement that would be invalid in this `wait` method.

      A constant method cannot change instance variables or call other non-constant methods. So, either of the following statements would be invalid inside the body of the this `wait` method:

      - `childPid = -1;    // Cannot change instance variables`
      - `forkNexec({"ls"});  // Cannot call non-const methods`

Part #3: Understanding serial vs. parallel execution

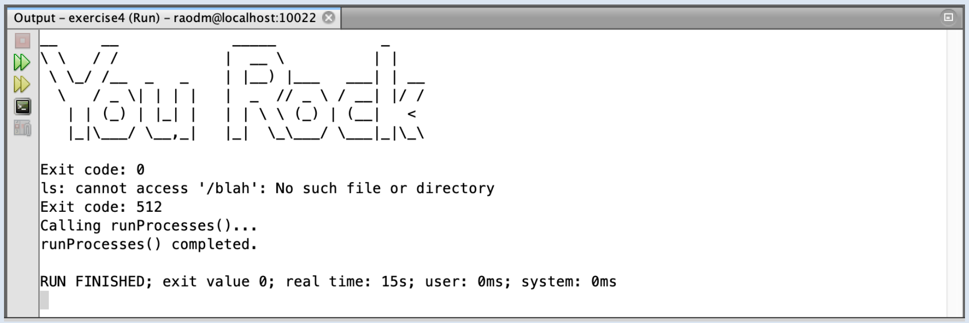
Background: In multiprocessing operating systems multiple processes can run simultaneously to take full advantage of the multi-core servers. Typically, a child process is started and we wait for the process to finish. This mode of operation is called serial or sequential processing, where only 1 child process runs at time. On the other hand, multiple processes can be run by first starting up several child processes (one after another) and then waiting for them to finish. This mode of operation is called parallel processing. Given sufficient hardware, parallel processing enables performing many tasks simultaneously. However, care must be taken so as not to overload the machine – too many processes can consume memory/resources and increase context-switching overheads, thereby negatively impacting overall performance.

Exercise: In this part of the exercise you are expected to covert a given serial program to run in parallel, via the following procedure.

1. Copy-paste the following starter code into the body of the `runProcesses` method in the `serial_vs_parallel.cpp` source file.

```cpp
void runProcesses() {
    // The command to be run -- this just sleeps for 5 seconds.
    const StrVec cmd = {"sleep", "5"};
    // Run the same process 3 times in serial fashion.
    ChildProcess cp;
    // 1st run
    cp.forkNexec(cmd);
    cp.wait();
    // 2nd run
    cp.forkNexec(cmd);
    cp.wait();
    // 3rd run
    cp.forkNexec(cmd);
    cp.wait();
}
```

2. Compile and run your program. You will notice that the program takes about 15 seconds to run. Copy-paste the output from the NetBeans terminal showing the runtime in the space below:



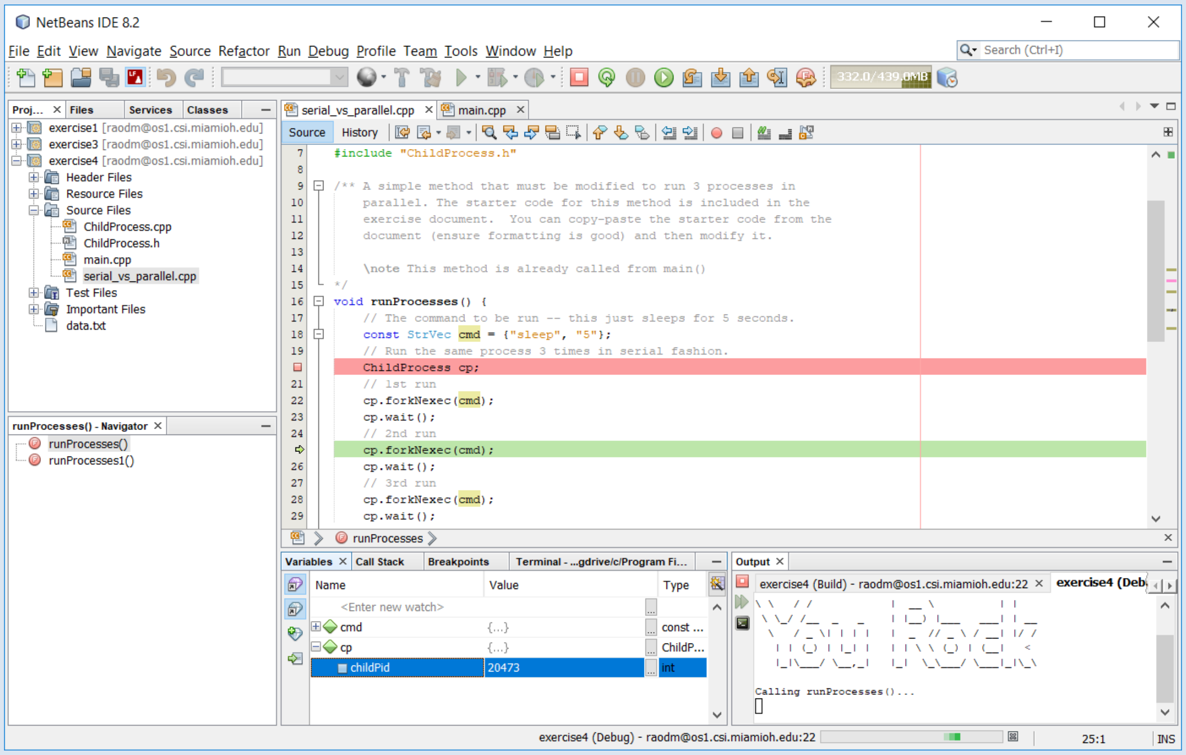
```
__     __            _____            _
\ \   / /           |  __ \          | |
 \ \_/ /__  _   _   | |__) |___   ___| | __
  \   / _ \| | | |  |  _  // _ \ / __| |/ /
   | | (_) | |_| |  | | \ \ (_) | (__|   <
   |_|\___/ \__,_|  |_|  \_\___/ \___|_|\_\

Exit code: 0
ls: cannot access '/blah': No such file or directory
Exit code: 512
Calling runProcesses()...
runProcesses() completed.

RUN FINISHED; exit value 0; real time: 15s; user: 0ms; system: 0ms
```

3. Let's see which lines of code take time to run by stepping through the debugger –
   a. Set a breakpoint at the 2nd line of the `runProcesses` method with the first call to the `forkNexec` method.
   b. Step through the code and observe which line of code (or method call) takes most time to run. Briefly describe why you think this line of code or method call takes so much time to finish?

Each of the `cp.wait()` calls take the most time to finish. Each call takes 5 seconds because the method waits for the child process running the "`sleep 5`" (sleep or do nothing for 5 seconds) command to finish.

c. Repeat stepping through the debugger and make a screenshot of the debugger showing the line of code that takes most time to run. Place screenshot below:

Place screenshot of debugger (as instructed above) in this space.



NetBeans IDE 8.2

File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help

Proj... Files Services Classes

- exercise1 [raodm@os1.csi.miamioh.edu]
- exercise3 [raodm@os1.csi.miamioh.edu]
- exercise4 [raodm@os1.csi.miamioh.edu]
  - Header Files
  - Resource Files
  - Source Files
    - ChildProcess.cpp
    - ChildProcess.h
    - main.cpp
    - serial_vs_parallel.cpp
  - Test Files
  - Important Files
  - data.txt

runProcesses() - Navigator

- runProcesses()
- runProcesses1()

serial_vs_parallel.cpp | main.cpp

```cpp
#include "ChildProcess.h"

/** A simple method that must be modified to run 3 processes in
    parallel. The starter code for this method is included in the
    exercise document.  You can copy-paste the starter code from the
    document (ensure formatting is good) and then modify it.

    \note This method is already called from main()
*/
void runProcesses() {
    // The command to be run -- this just sleeps for 5 seconds.
    const StrVec cmd = {"sleep", "5"};
    // Run the same process 3 times in serial fashion.
    ChildProcess cp;
    // 1st run
    cp.forkNexec(cmd);
    cp.wait();
    // 2nd run
    cp.forkNexec(cmd);
    cp.wait();
    // 3rd run
    cp.forkNexec(cmd);
    cp.wait();
```

Variables | Call Stack | Breakpoints | Terminal - ...gdrive/c/Program Fi...

Name	Value	Type
<Enter new watch>		
cmd	{...}	const ...
cp	{...}	ChildP...
childPid	20473	int

Output — exercise4 (Build) - raodm@os1.csi.miamioh.edu:22 | exercise4 (Deb

```
\ \   / /          |  __ \          | |
 \ \_/ /__  _   _  | |__) |___   ___| | __
  \   / _ \| | | | |  _  // _ \ / __| |/ /
   | | (_) | |_| | | | \ \ (_) | (__|   <
   |_|\___/ \__,_| |_|  \_\___/ \___|_|\_\

Calling runProcesses()...
```

exercise4 (Debug) - raodm@os1.csi.miamioh.edu:22 25:1 INS

4. Now modify the program to run the 3 processes in parallel. **Tip**: You will need 3 `ChildProcess` objects, 1 for each process.
5. Upon correctly converting to parallel operation, the program should run in 5 seconds.
6. Now, repeat the debugging operation to step through the code again to observe which line of code takes the most time
   a. Indicate which line of code (or method call) takes most time

   The first call to the `cp.wait()` calls take the most time to finish. Each call takes 5 seconds because the method waits for the child process running the "`sleep 5`" (sleep or do nothing for 5 seconds) command to finish.

   b. If the above method call takes time, then how come subsequent calls to the same method finish quickly?

The other 2 wait calls don't take much time because, by the time we call those methods, the child process has already finished running and hence the call to `waitpid` system call returns immediately.

a. Next, swap/change the order of calls to `waitpid` method in the revised parallel version (that is, wait for child 3, then child 2, then child 1). Does the runtime change? Why-or-why not?

In the parallel version, the order in which we wait for processes to finish does not matter. This is because if a child-process has finished running the wait method returns immediately, without taking much time at all. Hence, the runtime of the parallel version does not change because the overall runtime time depends on the slowest of the processes, which is 5 seconds.

b. Now, suppose that the serial version had 3 processes running 3 different commands, namely: `sleep 5`, `sleep 4`, and `sleep 3`. Then

Runtime of the **serial** version with the 3 processes would be	(5 + 4 + 3) == **12 seconds**
Runtime of the **parallel** version with the 3 processes would be	5 seconds (the slowest process determines overall runtime in parallel execution)

Part #4: Submit files to Canvas

Estimated time: < 5 minutes

Upload just the following files to Canvas via the CODE plug-in:

- `Scp` the 2 C++ files you modified (*i.e.*, `ChildProcess.cpp`, and `serial_vs_parallel.cpp`) from the Linux server to your local computer for submission.
- Upload this document (duly filled with the necessary information) saved as a PDF file using the naming convention `MUid_Exercise4.pdf`.
- The 2 C++ files you modified – `ChildProcess.cpp`, and `serial_vs_parallel.cpp`.
- Ensure you actually finish the submission process!

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