Homework 2

WORKING WITH FORK(), EXEC() AND WAIT()

The objective of this assignment is to get comfortable with the **`fork()`**, **`exec()`**, and **`wait()`** system calls

DUE DATE: Wednesday, February 19th @ 8:00 pm MT

Generative AI Use and Consequences
Use of AI tools such as ChatGPT, Claude, Github Co-Pilot, and/or their ilk to write or "improve" your code or written work at *any* stage is prohibited; this includes the ideation phase. It is your responsibility to ensure that you don't have the GitHub Co-Pilot extension installed in your IDE; assignment solutions generated by Co-Pilot aren't written by you. Turning in code or an essay written by generative AI tools will be treated as turning in work created by someone else, namely an act of plagiarism and/or cheating.

Ultimately, you will get out of the class what you put in. Simply copying and pasting code from generative AI tools is neither ethical nor does it contribute to your learning experience. There are multiple reasons why these generative AI tools are detrimental to your learning experience:

1. They rob you of the ability to think and learn the concepts for yourself. Solving problems is an essential step to gaining a solid understanding of the material.
2. You will struggle with the in-classroom quizzes and exams where you will not have access to these tools.
3. While we acknowledge that these tools are likely to become an important part of a software engineer's workflow in the future, you are much more likely to use these tools in an effective manner if you already have expertise in the relevant technical topics. Developing such expertise requires putting in the effort to learn these topics without the assistance of these tools.
4. These tools are prone to generating imperfect or even incorrect solutions, so trusting them blindly can lead to bad consequences.

Some helpful Infospaces videos for this assignment:

(1) https://infospaces.cs.colostate.edu/watch.php?id=277

(2) https://infospaces.cs.colostate.edu/watch.php?id=276

1 Description of Task

For this assignment you will be creating two programs: `Coordinator` and `Checker`

`Coordinator`: The coordinator is responsible for using the

(1) **`fork()`** command to launch another process
(2) **`exec()`** command to replace the program driving this process, while also supplying the arguments that this new program (`Checker`) needs to complete its execution.
(3) **`wait()`** command to wait for the completion of the execution of the process.

The `Coordinator` is responsible for launching 4 processes that it will load with the `Checker` program.

Each instance of the Checker will receive different arguments. To facilitate this, the Coordinator will take a total of five command line arguments and selectively pass them on to the Checker. The first argument is the divisor, followed by the dividends. For instance,

```
> coordinator 3 8 15 21 45
```

Would create 4 child processes that would check 8/3, 15/3, 21/3, and 45/3, respectively.

`Checker`: This program requires two arguments to complete its task. The Checker checks whether or not argTwo (the dividend) is divisible by argOne (the divisor) and prints out the result. Both these arguments are positive integers.

The two arguments that the `Checker` needs to perform its mathematical operation will be supplied to it by the `Coordinator;` the `Coordinator` is supplied these aforementioned arguments from the command line.

All print statements must indicate the program that is responsible for generating them. To do this, please prefix your print statements with the program name i.e. `Coordinator` or `Checker`. The example section below depicts these sample outputs.

A good starting point is to implement the functionality for the `Checker` program, and then write the code to manage its execution using the `Coordinator` program.

2 Requirements of Task

1. The Checker must accept two arguments, and the Coordinator must accept five command line arguments.
2. The Coordinator should spawn 4 processes using the **`fork()`** command and must ensure that it completes one full cycle of **`fork()`**, **`exec()`** and **`wait()`** for a given process before it moves on to spawning a new process.
3. Once it has used the **`fork()`** command, the Coordinator will print out the process ID of the process that it created. This can be retrieved by checking the return value of the **`fork()`** command.
4. Child-specific processing immediately following the **`fork()`** command then loads the Checker program into the newly created process using the **`exec()`** command. This ensures that the forked process is no longer a copy of the Coordinator. This **`exec()`** command should also pass 2 arguments to the Checker program. For this assignment, it is recommended that you use the **`execlp()`** command. The man page for **exec** (**`man 2 exec`**) will give details on the usage of the **`exec()`** family of functions.
5. When the Checker is executing it prints out its process ID; this should match the one returned by the **`fork()`** command in step 3.
6. The `Checker` then determines whether or not argTwo is divisible by argOne and prints this information.
7. If divisible, `Checker` should return a nonzero exit code. If not divisible, `Checker` should return 0. These correspond to the standard UNIX 'success' and 'failure' exit codes. Each exit code received by the Coordinator should be printed. You can use the **`WEXITSTATUS()`** macro to determine the exit status code (see **`man 2 wait`**).
8. Parent-specific processing in the `Coordinator` should ensure that the `Coordinator` will **`wait()`** for each instance of the child-specific processing to complete.

Figure 1 below depicts the assignment scenario. First, the Coordinator will complete one complete cycle of {fork, exec and wait} for each Checker process. Second, at a time only one Checker process will be active.

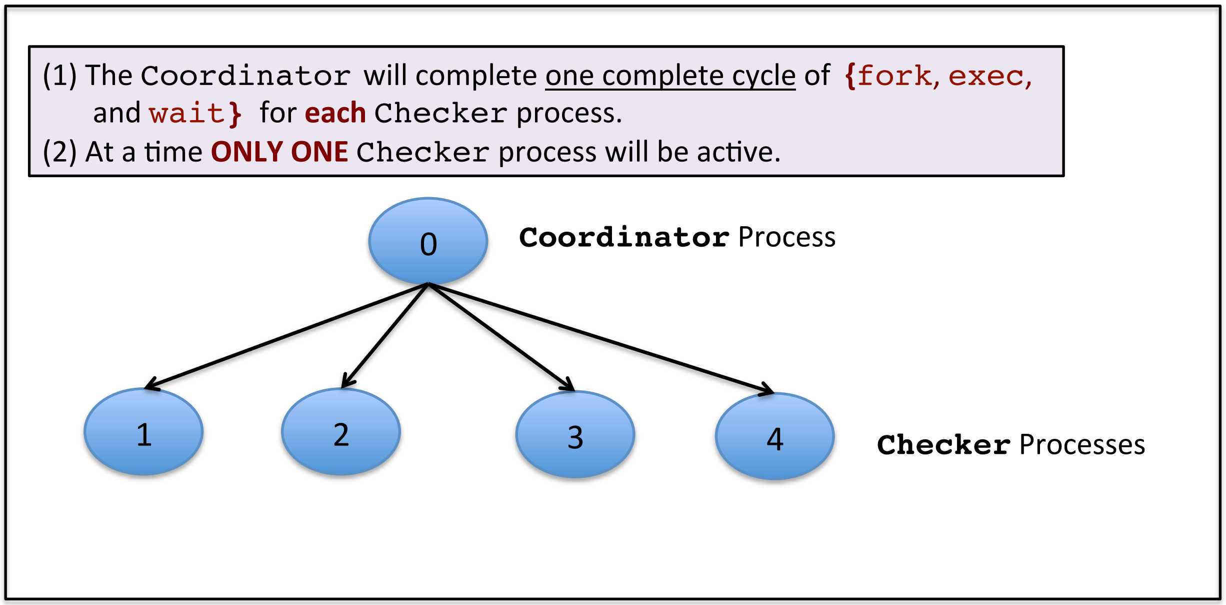
(1) The Coordinator will complete one complete cycle of {fork, exec, and wait} for each Checker process.
(2) At a time ONLY ONE Checker process will be active.
0
Coordinator Process
1
2
3
4
Checker Processes

Figure 1: Visual representation of the assignment goals

3 Example Output:

```
> ./coordinator 3 3 20 49 102
Coordinator: forked process with ID 23339.
Coordinator: waiting for process [23339].
Checker process [23339]: Starting.
Checker process [23339]: 3 *IS* divisible by 3.
Checker process [23339]: Returning 1.
Coordinator: child process 23339 returned 1.
Coordinator: forked process with ID 23340.
Coordinator: waiting for process [23340].
Checker process [23340]: Starting.
Checker process [23340]: 20 *IS NOT* divisible by 3.
Checker process [23340]: Returning 0.
Coordinator: child process 23340 returned 0.
Coordinator: forked process with ID 23341.
Coordinator: waiting for process [23341].
Checker process [23341]: Starting.
Checker process [23341]: 49 *IS NOT* divisible by 3.
Checker process [23341]: Returning 0.
Coordinator: child process 23341 returned 0.
Coordinator: forked process with ID 23342.
Coordinator: waiting for process [23342].
Checker process [23342]: Starting.
Checker process [23342]: 102 *IS* divisible by 3.
Checker process [23342]: Returning 1.
Coordinator: child process 23342 returned 1.
Coordinator: exiting.

> ./coordinator 7 32 49 846 22344
Coordinator: forked process with ID 23981.
Coordinator: waiting for process [23981].
Checker process [23981]: Starting.
Checker process [23981]: 32 *IS NOT* divisible by 7.
Checker process [23981]: Returning 0.
Coordinator: child process 23981 returned 0.
Coordinator: forked process with ID 23982.
Coordinator: waiting for process [23982].
Checker process [23982]: Starting.
Checker process [23982]: 49 *IS* divisible by 7.
Checker process [23982]: Returning 1.
Coordinator: child process 23982 returned 1.
Coordinator: forked process with ID 23983.
Coordinator: waiting for process [23983].
Checker process [23983]: Starting.
Checker process [23983]: 846 *IS NOT* divisible by 7.
Checker process [23983]: Returning 0.
Coordinator: child process 23983 returned 0.
Coordinator: forked process with ID 23984.
Coordinator: waiting for process [23984].
Checker process [23984]: Starting.
Checker process [23984]: 22344 *IS* divisible by 7.
Checker process [23984]: Returning 1.
Coordinator: child process 23984 returned 1.
Coordinator: exiting.
```

4 What to Submit

Assignments should be submitted through Canvas. E-mailing the codes to the Professor, GTA, or the class accounts will result in an automatic 1 point deduction.

Use the CS370 *Canvas* to submit a single .zip file that contains:

- All .c and .h files related to the assignment (please document your code),
- a Makefile that performs both a *make clean* as well as a *make all,*
- a README.txt file containing a description of each file and any information you feel the grader needs to grade your program.

Filename Convention: Your coordinator and checker must be named coordinator.c and checker.c respectively; you can name additional .c and .h files anything you want. The archive file should be named <FirstName>-<LastName>-HW2.zip . E.g. if you are Cameron Doe and submitting for HW2, then the zip file should be named Cameron-Doe-HW2.zip.

5 Grading

This assignment would contribute a maximum of 5 points towards your final grade. The grading will also be done on a 5 point scale. The points are broken up as follows:

0.5 points each for each of the tasks i.e. Task 1-8 (**4 points**)

1 point for getting the rest of the things right!

You are required to work alone on this assignment.

6 Late Policy

All assignments are due at 8:00 PM on the due date. There is a late penalty of 10% per-day for up to a maximum of 2 days.