EECE.4810/EECE.5730: Operating Systems

Spring 2020

Programming Project 1: Multi-Process Programming
Due **11:59 PM**, **Monday, 3/2/20**

1. Introduction

This project reviews fundamentals of multiprocessing. Your program will generate multiple processes using the UNIX `fork()` function, handle their return using `wait()`, and generate output from both parent and child processes to demonstrate the correctness of your approach.

This assignment is worth a total of 75 points. The given grading rubric in Section 3 applies to students in both EECE.4810 and EECE.5730.

2. Project Submission

Your submission must meet the following requirements:

- Your solution may be written in C or C++ (although there's very little in this assignment that requires object-oriented programming).
- Your code must run on the Linux machines in Ball 410, but you may write it elsewhere.
- You should submit a .zip file *(no other archive format)* containing the following files:
    - All source/header files you write. You can probably write your entire program in one file for this assignment, but if you use separate classes/structures, each class/structure should be defined in a header/source file pair.
    - A makefile that can be used to build your project
    - A README file that briefly describes the contents of your submission, as well as directions for compiling and running your code. The README file must contain the name(s) of everyone working on your submission.
- Programs must be submitted via Blackboard.
- You may work in a group of 2 students on this program (or work alone). If you are working in a group:
    - Contact Dr. Geiger as early as possible so your group can be set up in Blackboard.
    - As noted above, your README file should contain the names of all group members.

3. Specification and Grading Rubric

As noted above, the program should use `fork()` to start each new process and `wait()` to collect the return status (and possibly check the PID) of each.

Your assignment will be graded according to the rubric below; partial credit may be given if you successfully complete part of a given section. Please note that the grader may identify errors in your code not listed here because we haven't anticipated those errors occurring. A final rubric will be posted to explain any new errors found and assessed.

Your final submission should NOT include separate programs for each section—you should submit ONE program that fulfills the requirements of as many sections as possible.

	Points
Section A: At least 1 additional child process	15
Section B: ≥ 1 child processes, using loop to generate constant # of children	22
Section C: ≥ 1 child processes, # children based on command line argument	11
Section D: Program can differentiate between child processes as they finish	11
Section E: Each child process starts (same) new program	11
Section F: Each child process starts 1 of 5 new programs	5

Known deductions	
Submission is missing makefile	-3
Submission is missing README	-3
Children fork and wait instead of parent	-4
Test programs run sequentially	-5
Only 1 process starts new executable	-5
Program separates "child finished" message from actual wait	-10
Program waits for children in order using `waitpid()`	-1
Program incorrectly assumes processes finish in order	-5
Incorrectly assumed child PID = parent + child number	-5
Program unnecessarily uses shared memory	-5
Program doesn't associate finished child with start position	-5
Miscellaneous error (segmentation fault, compilation issues, etc)	-10

A detailed description for each section begins on the next page.

3. Specification and Grading Rubric (continued)

Section description:

A. (15 points) Your program creates at least one child process that is a copy of the original program. Both processes print at least one message indicating their PIDs, and the parent waits for the child to terminate, printing a message once it has done so.

B. (11 points) Your program creates more than one child process, with all children following the guidelines above (copy of the original program, prints a message indicating PID, parent waits for termination). In addition:

- All messages must contain a "child number" indicating when each child was created relative to the other processes. (The first is "Child 1," then "Child 2", etc.)
- The "child number" is related to the order in which the child processes are created—not necessarily the order in which they finish. "Child 1" is always the first child process you start, but it may not be the first child that finishes.
- The child processes should run simultaneously, not sequentially. In other words, you should start all child processes, then start using `wait()` to check for child processes finishing. You should not wait for the first child process to finish before starting the second.
- Your program uses a loop to create multiple children. Each child process prints messages as described above.
- Any program that uses a loop to create multiple children will get credit for this part; you may want to test the program with a fixed number of child processes before moving on to the next part, which requires user input.

C. 11 points: As above, your program uses a loop to create multiple children, but the number of child processes is based on a command line argument passed to your executable, not a constant limit. For example, if your executable is named "`proj1`", executing the command `./proj1 6` will run a version of your program that creates 6 child processes. Assume the maximum number of child processes is 25.

D. **11 points:** Your program is able to differentiate between different child processes finishing, printing not just the PID but also the "child number" described above.

- For example, when the first child completes, print a message saying, "`Child 1 (PID` *xxxxx*`) finished`", where *xxxxx* would be replaced by the actual PID. **The number and PID printed when each child finishes must match the numbers printed when that child is created.**

E. **11 points:** Your program starts multiple child processes as described above, but each child process starts a new program, replacing the address space of the parent with that of the new program.

- To earn these points, your child processes can all start the same program (I recommend test1, described below)

3. Specification and Grading Rubric (continued)

F. **5 points:** Each child process starts one new program from a set of five possible new programs. Source code for the new programs is on Blackboard. The test programs are:
   - test1.c: Prints values from 0 to 4, along with the square of each value.
   - test2.c: Calculates and prints the square root of the PID.
     - Since this program uses the math library, you must pass the –lm (lowercase L followed by m) option to gcc when compiling test2.c.
   - test3.c: Determines whether the PID is odd or even.
   - test4.c: Calculates and prints the number of digits in the PID.
   - test5.c: Uses a recursive quicksort function to sort an array of ten integers.

- **Please ensure (1) your executable names match the .c file names, without the .c ("test1", "test2", etc.) and (2) your program assumes all executables are in the same directory and test programs can therefore be started without a full path.** The path "./test1", for example, assumes "test1" is in the same directory as your executable.
- **You may NOT modify the source code for the test files**. Your program should generate correct output without any changes to those tests.

4. Hints

Program design: One approach to writing this program, particularly if you're struggling to get started, is to start with a relatively simple program that accomplishes one of the tasks in the grading rubric, then builds upon that program once you've established it works correctly.

When I initially wrote my solution for this assignment, I followed the steps below. You are not required to follow these steps—after all, you're submitting a single program that (hopefully) meets as many of the requirements in the rubric as possible.

Sample outputs from solutions at each step are shown in Section 5: Test Cases. **A successfully completed program will have output similar to what's shown on page 8 for Section F.**

- My first version created a single child process as a copy of the original program. Each process printed at least one message showing its PID, and the parent waited for the child to terminate, then printed a message indicating the child had completed.
- My next version created multiple child processes without a loop, printing messages at the start and end of each process, with the message containing both the process's PID and a "child number" indicating when each child was created relative to other children (the first child is "Child 1," then "Child 2", and so on.) Reminders from the grading rubric:
  - The child processes must run simultaneously, not sequentially.
  - "Child number" is related to the order in which child processes are created—not necessarily the order in which they finish.
- I modified the previous version to use a loop for child process creation, initially using a constant loop limit and then basing the limit on a command-line argument.

4. Hints (continued)

Program design (continued):

- My next version was able to differentiate which child finished and print both the PID and "child number" for each completed process. Previous versions only printed a PID as a process finished.
- Finally, I created a version in which each child process started one of the sample programs described in the grading rubric. Initially, every child process started a copy of "test1". I then modified my program so different children started different programs, with the children cycling through sample programs test1-test5.

Useful functions: The multiprocess examples covered in Lectures 2-3 should serve as a starting point for your program. The following additional functions may be useful:

- `pid_t getpid()`: Returns the process ID of the currently running process.
- `int atoi(char *str)`: Converts `str` to the corresponding integer value—for example, `atoi("33") = 33`.
- `int sprintf(char *str, const char *format, … )`: Prints the string specified by `format` and the following arguments to the string `str`. For example, if `x = 7`, `sprintf(s, "x = %d", x)` writes the string `"x = 7"` to the string `s`.

Tracking child processes: In order to ensure that the "child numbers" are generated correctly, you'll have to keep track of the number of child processes created as you do that. Doing so is much simpler if you ensure only one process (most likely the original parent process) can create child processes.

While you can certainly design a solution in which child processes are allowed to create other child processes, that solution may be needlessly complicated.

5. Test Cases

This section provides sample outputs for the different program versions described in Section 4: Hints. **Your goal is to write a program that generates similar output to the final test case (G)—the earlier cases simply give you an idea of how your program might behave if you follow the same iterative design process I did.** Your outputs will likely include different PIDs, and statements may be in a different order than the test cases (and in different program runs).

A. *Single child process:*

```
Parent pid is 4810
Started child with pid 4811
Child (PID 4811) finished
```

B. *Multiple child processes:* two child processes is the minimum required for this case.

```
Parent pid is 5730
Started child 1 with pid 5731
Started child 2 with pid 5732
Child (PID 5731) finished
Child (PID 5732) finished
```

C. *Number of child processes based on command-line argument:* output will be similar to Part B, with only (major) difference being number of children.

D. *Program can differentiate which child finishes:* example below assumes three children. The main difference between Part E and Parts B-D is each message indicating a child has finished contains both the PID and an identifier ("child 1") indicating the order in which that child was created.

```
Parent pid is 5769
Started child 1 with pid 5770
Started child 2 with pid 5771
Started child 3 with pid 5772
Child 2 (PID 5771) finished
Child 1 (PID 5770) finished
Child 3 (PID 5772) finished
```

5. Test Cases (continued)

Your goal is to write a program that generates similar output to the final test case (G)—the earlier cases simply give you an idea of how your program might behave if you follow the same iterative design process I did.

E. *All child processes start same executable:* the example below starts 3 copies of "test1". Note that, while each output from that program starts with "T1" to make it clear which test program is running, outputs from the different child processes are interleaved, making it difficult to tell which process is generating each output line.

```
Parent pid is 8556
Started child 1 with pid 8557
Started child 2 with pid 8558
Started child 3 with pid 8559
Running program test1 in process 8558
T1: i 0, i^2 0
T1: i 1, i^2 1
T1: i 2, i^2 4
T1: i 3, i^2 9
T1: i 4, i^2 16
Running program test1 in process 8557
T1: i 0, i^2 0
T1: i 1, i^2 1
Running program test1 in process 8559
T1: i 2, i^2 4
T1: i 0, i^2 0
T1: i 3, i^2 9
T1: i 1, i^2 1
T1: i 4, i^2 16
T1: i 2, i^2 4
T1: i 3, i^2 9
T1: i 4, i^2 16
Child 2 (PID 8558) finished
Child 3 (PID 8559) finished
Child 1 (PID 8557) finished
```

5. Test Cases (continued)

F. *Each child process starts one of five executables:* the example below shows a test run with 6 child processes. Note that child 1 and child 6 start the same executable ("test1").

THIS TEST CASE SHOWS THE FORMAT OF YOUR FINAL PROGRAM OUTPUT. ALL EARLIER CASES SIMPLY SHOW EXAMPLES OF POSSIBLE STEPS IN AN ITERATIVE DESIGN PROCESS.

```
Parent pid is 8574
Started child 1 with pid 8575
Started child 2 with pid 8576
Started child 3 with pid 8577
Started child 4 with pid 8578
Running program test1 in process 8575
T1: i 0, i^2 0
T1: i 1, i^2 1
T1: i 2, i^2 4
T1: i 3, i^2 9
T1: i 4, i^2 16
Started child 5 with pid 8579
Started child 6 with pid 8580
Child 1 (PID 8575) finished
Running program test3 in process 8577
T3: PID 8577 is odd
Running program test2 in process 8576
T2: sqrt of PID 8576 is 92.61
Running program test4 in process 8578
T4: PID 8578 has 4 digits
Child 3 (PID 8577) finished
Child 2 (PID 8576) finished
Child 4 (PID 8578) finished
Running program test1 in process 8580
T1: i 0, i^2 0
T1: i 1, i^2 1
T1: i 2, i^2 4
T1: i 3, i^2 9
T1: i 4, i^2 16
Child 6 (PID 8580) finished
Running program test5 in process 8579
T5: QS L[0-9]
T5: QS L[0-3]
T5: QS L[0-2]
T5: QS L[5-9]
T5: QS L[5-7]
T5: QS L[5-6]
T5: Final list = 1 2 3 4 5 6 7 8 9 10
Child 5 (PID 8579) finished
```