CSCI 4210 — Operating Systems
Lecture Exercise 2 (document version 1.1)

- This lecture exercise is due by 11:59PM ET on Wednesday, February 17, 2021
- This lecture exercise consists of practice problems and problems to be handed in for a grade; graded problems are to be done individually, so **do not share your work on graded problems with anyone else**
- For all lecture exercise problems, take the time to work through the corresponding video lecture(s) to practice, learn, and master the material; while the problems posed here are usually not exceedingly difficult, they are important to understand before attempting to solve the more extensive homeworks in this course
- As with the homeworks, you **must** use C for this assignment, and all submitted code **must** successfully compile via `gcc` with no warning messages when the `-Wall` (i.e., warn all) compiler option is used; we will also use `-Werror`, which will treat all warnings as critical errors
- All submitted code **must** successfully compile and run on Submitty, which uses Ubuntu v18.04.5 LTS and `gcc` version 7.5.0 (`Ubuntu 7.5.0-3ubuntu1~18.04`)

Practice problems

Work through the practice problems below, but do not submit solutions to these problems. Feel free to post questions, comments, and answers in our Discussion Forum.

1. What is the exact output of the `fork.c` example from the February 8 video lecture if we modify the code as shown below (adding `printf()` statements both before and after we call `fork()`)?

```
printf( "PID %d: BEFORE fork()\n", getpid() );
pid = fork();
printf( "PID %d: AFTER fork()\n", getpid() );
```

   To best answer this question, revise the diagram shown in the comments after the `main()` function.

2. What are the differences between the `waitpid()` and `wait()` system calls (covered in the February 10-11 video lectures)? What does the return value tell us? And what is the `wstatus` parameter used for?

3. Describe at least three reasons `fork()` could fail.

4. Review the `fork-lexec2.c` code posted along with this lecture exercise (also shown on the next page). Assuming no errors occur, determine exactly how many distinct possible outputs there could be. Show all possible outputs.

```
/* fork-lexec2.c */

/*
 * Lecture Exercise 2 -- Practice Problem 4
 *
 * How many distinct possible outputs are there?
 *
 * Show all possible outputs.
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{
  int t;
  int * q = &t;
  t = 19;
  pid_t p = fork();

  /* for Lecture Exercise 2, assume fork() succeeds */
  if ( p == -1 )
  {
    perror( "fork() failed" );
    return EXIT_FAILURE;
  }

  if ( p == 0 )
  {
    printf( "CHILD: happy birthday to me!\n" );
    printf( "CHILD: *q is %d\n", *q );
  }
  else /* p > 0 */
  {
    printf( "PARENT: child process created\n" );
    *q = 73;
    printf( "PARENT: *q is %d\n", *q );
  }

  return EXIT_SUCCESS;
}
```

Graded problems

Complete the problems below and submit via Submitty for a grade. Please do not post any answers to these questions. All work on these problems is to be your own.

1. Based on the `fork.c` example from the February 8 video lecture, write code to display the output shown below in the exact given order (i.e., do not allow interleaving to occur).

   Write all of your code in `lecex2-q1.c` for this problem.

   You must use both `fork()` and `waitpid()` to complete this problem. Each line of output that starts with `"PARENT"` must be displayed by the parent (original) process; likewise, each line of output that starts with `"CHILD"` must be displayed by the child process.

   ```
   bash$ ./a.out
   PARENT: okay, start here.
   CHILD: happy birthday to me!
   CHILD: i'm bored....self-terminating....good-bye!
   PARENT: child process terminated successfully.
   PARENT: sigh, i'm gonna miss that little child process.
   bash$
   ```

2. Copy your solution from above and modify it to have the parent (original) process create two child processes. Your program must display the output shown below in the exact given order. Again use both `fork()` and `waitpid()` to complete this problem.

   Write all of your code in `lecex2-q2.c` for this problem.

   As above, each line of output that starts with `"PARENT"` must be displayed by the parent (original) process. Further, each line of output that starts with `"CHILD A"` must be displayed by the *first* child process; and each line of output that starts with `"CHILD B"` must be displyaed by the *second* child process.

   Be sure you have one parent process with two child processes. **(v1.1)** Note that both child processes would **not** exist simultaneously (so create one, wait for it to terminate, then create the second one). Use shell commands `ps` and `grep` to be sure this is the case (and since the computer is faster than you, also consider using `sleep()` in your test/debugging code).

   ```
   bash$ ./a.out
   PARENT: okay, start here.
   CHILD A: happy birthday to me!
   CHILD A: i'm bored....self-terminating....good-bye!
   CHILD B: and happy birthday to me!
   CHILD B: see ya later....self-terminating!
   PARENT: both child processes terminated successfully.
   PARENT: phew, i'm glad they're gone!
   bash$
   ```

3. Review the `forked.c` code posted along with this lecture exercise (also shown on the next page). Do **not** change the `forked.c` code or submit this code to Submitty. Submitty will compile your own code file in with this given `forked.c` code.

   In the `forked.c` code, the parent process calls a `lecex2_parent()` function, whereas the child process calls a `lexec2_child()` function. Your task is to write these two functions in your own `fork-functions.c` code file. Each of these functions is described below.

   Only submit your `fork-functions.c` code file for this problem.

   - In the `lecex2_child()` function, you must open and read from a file called `data.txt`. Using any technique you would like, read/identify the sixth character in that file, then close the file.

     If all is successful, exit the child process and return the sixth character as its exit status.

     If instead an error occurs, display an error message to `stderr` and use the `abort()` library function to abnormally terminate the child process. **(v1.1)** Check the `man` page for `abort()` for more details.

   - In the `lecex2_parent()` function, you must wait for the child process to terminate by using `waitpid()`.

     If the child process terminated abnormally, display the following line of output and return `EXIT_FAILURE`:

     ```
     PARENT: child process terminated abnormally
     ```

     If instead the child process terminated normally, display the following line of output and return `EXIT_SUCCESS`:

     ```
     PARENT: child process reported '<char>'
     ```

     Note that `<char>` here represents the one-byte exit status received from the child process and should be displayed as a single character. As an example, if `'Q'` was the sixth character read by the child, the parent would output:

     ```
     PARENT: child process reported 'Q'
     ```

```c
/* forked.c */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

/* implement these functions in fork-functions.c */
int lecex2_child();
int lecex2_parent();

int main()
{
  int rc;

  /* create a new (child) process */
  pid_t p = fork();

  if ( p == -1 )
  {
    perror( "fork() failed" );
    return EXIT_FAILURE;
  }

  if ( p == 0 )
  {
    rc = lecex2_child();
  }
  else /* p > 0 */
  {
    rc = lecex2_parent();
  }

  return rc;
}
```