CSCI 4210 — Operating Systems
Lecture Exercise 2 (document version 1.0)
Process Creation/Management in C

- This lecture exercise is due by 11:59PM EDT on Wednesday, June 14, 2023
- 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 course content 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 assignments in this course
- 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 currently uses Ubuntu v20.04.6 LTS and `gcc` version 9.4.0 (`Ubuntu 9.4.0-1ubuntu1~20.04.1`)

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 if we modify the code by adding the following `printf()` statements both before and after the `fork()` call?

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

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

   Next, how does the output change if we redirect to a file as shown below, i.e., what is the exact contents of the `output.txt` file? As a hint, think about the buffering behavior.

   ```
   bash$ ./a.out > output.txt
   ```

2. What are the differences between the `waitpid()` and `wait()` system calls?

   What does the return value tell us?

   What is the `wstatus` parameter used for? How many bits are dedicated to each component encoded in this value?

3. Review the `fork-lecex2.c` code shown below. Assuming no runtime errors occur, how many distinct possible outputs could there be? Show all possible outputs by drawing a diagram.

```
int main()
{
  int t = 7;
  int * q = &t;
  pid_t p = fork();

  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 = 13;
    printf( "PARENT: *q is %d\n", *q );
  }

  return EXIT_SUCCESS;
}
```

   Remember that the parent (original) and child processes each indepenently reach the `return` statement in `main()`.

   Before we return from `main()` and exit each process, how many bytes are statically allocated for the given variables in the parent process? And in the child process?

   How many bytes are dynamically allocated in each process?

4. Modify the `fork-with-waitpid.c` example to have the parent process call `fork()` a second time after the `waitpid()` call, thereby creating a second child process.

   Have the second child process display the exit status of the first child process. To accomplish this, capture the exit status in the parent process before creating the second child process.

5. Modify the `fork-with-exec.c` example (once it becomes available) to execute your Homework 1 executable in the child process.

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. Write code to display the output shown below in the exact given order, i.e., do **not** allow interleaving to occur between the parent and child processes.

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

   You must call 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.

   The child process must call `open()` and `read()` to access the `lecex2-q1-input.txt` file. Specifically, this file contains a 4-byte `int` value in an endianness that matches the underlying architecture. (Do not use `scanf()` or `fscanf()` to read this binary file.)

   The child process returns this `int` value as its exit status.

   The parent process must then capture and display the exit status of the child process, indicated below as `<child-exit-status>`. And rather than have the parent process return `EXIT_SUCCESS`, return this captured value.

   ```
   bash$ gcc -Wall -Werror lecex2-q1.c
   bash$ ./a.out
   PARENT: start here
   CHILD: opened lecex2-q1-input.txt
   CHILD: read an int
   CHILD: returning the int
   PARENT: heat in <child-exit-status>
   bash$ echo $?
   <child-exit-status>
   bash$
   ```

   For this problem, use `waitpid()` to synchronize the parent and child processes. Here, synchronization means we are guaranteeing a specific order in which these two processes execute, thereby always producing the exact output shown above.

2. Review the `forked.c` code posted along with this lecture exercise (also shown on the next page). In this code, the parent process calls the `lecex2_parent()` function, while the child process calls the `lecex2_child()` function.

   Your task is to write these two functions in your own `lecex2-q2.c` code file. Each of these functions is described further below.

   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 as follows:

   ```
   bash$ gcc -Wall -Werror forked.c lecex2-q2.c
   ```

   Only submit your `lecex2-q2.c` code file for this problem.

   (a) In the `lecex2_child()` function, you must open and read from a file called `lecex2.txt`. Using any technique you would like, read the `n`th character in that file.

   If all is successful, exit the child process and return the `n`th 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.

   Read the `man` page for `abort()` for more details.

   (b) In the `lecex2_parent()` function, use `waitpid()` to suspend the parent process and wait for the child process to terminate.

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

   ```
   PARENT: oh no, child process terminated abnormally!
   ```

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

   ```
   PARENT: child process successfully returned '<char>'
   ```

   Note that `<char>` here represents the 1-byte exit status received from the child process and should be displayed as a single character.

   As an example, if `'G'` was the `n`th character read by the child, the parent would output:

   ```
   PARENT: child process successfully returned 'G'
   ```

```c
/* forked.c */

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

/* implement these functions in lecex2-q2.c */
int lecex2_child( int n );
int lecex2_parent();

int main()
{
  int n = 7;  /* or some other value ... */
  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( n );
  }
  else /* p > 0 */
  {
    rc = lecex2_parent();
  }

  return rc;
}
```