CSCI 4210 — Operating Systems
Lecture Exercise 2 (document version 1.0)

- This lecture exercise is due by 11:59PM on Wednesday, February 5, 2025
- This lecture exercise consists of practice problems and problems to be handed in for a grade; graded problems are to be done individually, so **please do not share your work on graded problems with others**
- 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 homework 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 v22.04.5 LTS and `gcc` version 11.4.0 (`Ubuntu 11.4.0-1ubuntu1~22.04`)
- You will have **eight** penalty-free submissions on Submitty, after which points will slowly be deducted, e.g., `-1` on submission #9, etc.
- You will have at least **three** days before the due date to submit your code to Submitty; if the auto-grading is not available three days before the due date, the due date will be 11:59PM three days after auto-grading becomes available

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 terminal 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() );
   /* create a new (child) process */
   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.

   How does the behavior and output change if we remove the newline `'\n'` characters from the two `printf()` statements added above?

   Next, how does the output change if we redirect `stdout` to a file? As a hint, review the posted `fork-buffering.c` example.

2. Review the code shown below. Assuming no runtime errors occur, how many distinct possible outputs to the terminal could there be? Show all possible outputs by drawing a diagram.

   How does program behavior change if output is redirected to a file?

```
int main()
{
  int t = 6181;
  int * q = &t;
  pid_t p = fork();

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

  if ( p == 0 )
  {
    q = calloc( 13, sizeof( int ) );
    *q = 123;
    *(q + 3) = 333;
    printf( "CHILD: happy birthday to me!\n" );
    printf( "CHILD: *q is %d\n", *q );
    free( q );
  }
  else /* p > 0 */
  {
    printf( "PARENT: child process created\n" );
    *q += 113;
    printf( "PARENT: *q is %d\n", *q );
  }

  return EXIT_SUCCESS;
}
```

   Remember that the parent (original) and child processes each independently 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?

3. 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 the `wstatus` value? As a hint, refer to the `man` page for `waitpid()` and also use the `-E` flag of `gcc`.

4. Modify the `fork-with-exec.c` example to show a `man` page for the first command-line argument. As an example, you would mimic this shell command:

   ```
   bash$ man fork
   ```

   To run your code, use:

   ```
   bash$ ./a.out fork
   ```

   First implement using `execl()`, then review the `man` pages and replace `execl()` with `execv()`.

   Also add functionality to allow the user to specify which section to view, i.e., `1`, `2`, etc.

5. Modify the posted `fork-pass-by-fork.c` example to use the first command-line argument, i.e., `*(argv+1)`, as input string `data`. Next, have the parent process create enough child processes such that the first child process converts the first seven bytes to uppercase, the second child process converts the next seven bytes to uppercase, and so on through to the end of the given string.

   The output should be formatted as given in the `fork-pass-by-fork.c` example.

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.

No square brackets allowed! As with our previous assignments, use pointer arithmetic. Any line of code containing square brackets, including comments, will be **automatically deleted on Submitty** before compiling via `gcc`.

1. 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-q1.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 a hidden version of the given `forked.c` code as follows:

   ```
   bash$ gcc -Wall -Werror forked.c lecex2-q1.c
   ```

   Only submit your `lecex2-q1.c` code file for this problem.

   (a) In the `lecex2_child()` function, open and read from a file called `lecex2.txt`. Using `open()` and `read()`, read the `n`th character in that file, then close the file.

   If all is successful, exit the child process and return the `n`th character (i.e., one byte) as its exit status. Note that the first byte in the file is considered `n` = 1, the second is `n` = 2, etc.

   If 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 terminates abnormally, display the following line of output in the parent process and return `EXIT_FAILURE`:

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

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

   ```
   PARENT: child process exited with status '<char>'
   ```

   Note that `<char>` here represents the exit status received from the child process—display this as a single character.

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

   ```
   PARENT: child process exited with status 'G'
   ```

```c
/* forked.c */

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

/* implement these functions in lecex2-q1.c */
int lecex2_child( unsigned int n );
int lecex2_parent();

int main()
{
  unsigned int n = 7;  /* or some other non-negative 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;
}
```

2. Based on the `fork-with-waitpid.c` example, write code to create a child process, then have this child process also call `fork()`, thereby creating a "grandchild" process. Place your code entirely in `lecex2-q2.c`.

   The grandchild process must open the input file given as the first command-line argument, i.e., `*(argv+1)`, and count the number of *properly nested* pairs of `'{'` and `'}'` characters in the file.

   After processing the input file, the grandchild process returns the accumulated count as its exit status.

   Next, the child process doubles this count and returns it (to the original parent) as its exit status. Finally, the original (top-level parent) process outputs this returned value.

   If any errors occur, display a meaningful error message to `stderr` and return `-1` as a special sentinel value.

   A simple example is shown below.

```
bash$ cat simple.txt
int main() { printf( "LET'S GO LAKERS!\n" ); }
bash$ ./a.out simple.txt
GRANDCHILD: counted 1 properly nested pair of curly brackets
CHILD: doubled 1; returning 2
PARENT: there are 2 properly nested curly brackets
```

   A more complex example is shown below.

```
bash$ cat input.txt
{ /* print parent PID value */
{ printf( "PARENT: my PID is {%d}\n", getpid() ); } } /* :-}  :-{ */
bash$ ./a.out input.txt
GRANDCHILD: counted 3 properly nested pairs of curly brackets
CHILD: doubled 3; returning 6
PARENT: there are 6 properly nested curly brackets
```

   Another example is shown below, with all three lines of output written to `stderr`.

```
bash$ ./a.out nosuchfile.txt
GRANDCHILD: open() failed: No such file or directory
CHILD: rcvd -1 (error)
PARENT: rcvd -1 (error)
```

   Specifically, error messages must be one line of output and begin with `GRANDCHILD:`, `CHILD:`, or `PARENT:` (the rest of the line can be anything you would like). This technique illustrates an example of propagating an error back through one or more parent processes.

What to submit

Please submit exactly two C source files called `lecex2-q1.c` and `lecex2-q2.c`. (Do not use any local header files.) These two files will be automatically compiled and tested against various test cases, some of which will be hidden.