Exercise 05

In this exercise, you get to write a multi-threaded program using the POSIX threads API (pthreads). You'll also get to see how execution order can be hard to predict and can vary from one execution to the next. This is something we'll need to appreciate when we start talking about race conditions and synchronization.

On the course homepage, you'll find a partial implementation of a program called *interleaving.c*. The main thread of your program runs a loop for 100,000 iterations using the write() system call to print out a newline at the end of each iteration.

Add code to create three new threads on each iteration of this loop, then join with them all before printing the newline. Each thread will use a different start routine to print different characters to standard output. One thread will print an 'a' then a 'b'. Instead of using the C standard output library, your thread should use the write() system call to print each character; make a separate call to write() to print each character. We could easily print both characters with a single call to write(), but making the two calls will help illustrate the execution orders we're getting for these threads.

Using the same technique, another thread will print 'c' then 'd', and the third thread will print 'e' then 'f'. After printing its two characters, each thread will just terminate.

We're using separate calls to write() to print each character so that we can see evidence of varied execution order among the threads. On each iteration of the loop in main(), we should get an output line with six characters, but the order of these characters will tell us something about the order in which the threads ran. For example, maybe we'll get a line “abcdef”. That tells us the first thread printed its two characters, then the second thread, then the last thread. Since each character is printed individually, we could get an order like “acebdf”. This tells us each thread got to print its first character, then each one got to print its second character.

With each thread running such a short sequence of statements and then terminating, you might think we would always get the same output. Let's see what we actually get. Compile the program and then run it as follows. When you compile, be sure to link using the -lpthread flag, so you get the functions in the pthreads API. This command takes the output of the program, sorts it and then uses the uniq command to discard duplicate lines. This should leave us with a copy of just each unique output line written by the program.

```
shell$ ./interleaving | sort | uniq
```

Examine your output and see what you get. I got around 30 different execution orders when I ran this program on an EOS linux machine. It looks like the remote machines are are all 2-core machines now. More cores will give you more interesting results. You get more interesting behavior from the CPU scheduler and you can even get parallel execution of the threads.

When you're done, submit the completed source file, interleaving.c using the exercise_05 assignment on Moodle.