**CPSC 261: Lab 7**

**Lab 7 - Thread Synchronization**

* [Goals](http://www.ugrad.cs.ubc.ca/~cs261/2013w2/labs/lab7.html#goals)
* [Introduction](http://www.ugrad.cs.ubc.ca/~cs261/2013w2/labs/lab7.html#introduction)
* [Overview](http://www.ugrad.cs.ubc.ca/~cs261/2013w2/labs/lab7.html#overview)
* [The Details](http://www.ugrad.cs.ubc.ca/~cs261/2013w2/labs/lab7.html#details)
* [Provided Materials](http://www.ugrad.cs.ubc.ca/~cs261/2013w2/labs/lab7.html#provided)
* [Handing It In](http://www.ugrad.cs.ubc.ca/~cs261/2013w2/labs/lab7.html#handing)

**Goals**

The goals of this lab are to:

* Help you understand thread synchronization in C programs using pthreads.

**Introduction**

This lab will help you explore the more advanced thread management primitives in Linux including mutexs and condition variables.

**Overview**

This lab has just one part:

* You must write C programs that implement the Dining Philosophers problem. You can read about the Dining Philosophers problem on the web. Use Google, the Wikipedia article is good enough.

Your philosophers should take little naps after every bit of exertion. That is, they nap each time before starting to eat, nap after getting one fork before trying to get the other fork, nap after getting both forks, and nap after putting the forks down before trying to eat again. Each of these naps should be a random length of time specified by a defined symbol named NAPTIME whose value should be defined to be 25000 usec. Napping can then be accomplished by:

usleep(random() % NAPTIME)

I want you to implement the problem with 5 philosophers and 5 forks. Following good programming practice, your solution should use a constant NPHILOSOPHERS (defined to be 5) to determine how many philosophers there are, and should create and use a different number of philosophers if that constant is changed. The philosophers will be numbered 0 through 4, the forks will be numbered 0 through 4, and the relationship between forks and philosophers will be:

Left fork

The left fork for philosopher i is numbered i

Right fork

The right fork for philosopher i is numbered (i + 1) mod 5

Each of your philosophers will try to eat a certain number of times, and then will be done. That number should be specified using another defined constant named NMEALS. When a philosopher has eaten that many times, it will be done and its thread should exit.

You will want to be able to get your philosophers to print out what they are doing. If the defined constant VERBOSE is 1, then you should print out a message each time a philosopher does something interesting (gets a fork, eats, puts down a fork, exits).

I'm not going to tell you precisely how you must implement this synchronization task using threads, but here are some ideas to get you started:

* + You'll probably want to have a data structure for each fork (a C struct). It will probably need a flag to indicate whether it is free or in use, a lock (pthread\_mutex\_t) to synchronize access to the free flag, and a condition variable (pthread\_cond\_t) so that threads can wait for access to the fork. A function to initialize these objects would seem like a good idea.
  + I found it useful to have a function that given a philosopher id (0 .. 4) would return the index of his left fork, and another one to get the index of his right fork.
  + You'll probably want a function to pick up a fork and another one to put one down.
  + You'll need a function that is the body of the thread that represents each philosopher. You will create NPHILOSOPHERS threads, each of them will run this function.

You will also implement three different synchronization strategies:

Left fork first and hold

The philosophers pick up the left fork first, and then pick up the right fork. If they can't get the either fork immediately, they just wait for it to be available.

Resource ordering

The philosophers obey a global resource order. That is, they look at the two forks that they need to acquire, and pick up the lowest numbered fork first. Except for that, they behave as in the Left fork first and hold synchronization strategy.

Random

The philosophers try to pick up a random fork first, and then try to pick up the second fork. If a philosopher is not able to acquire both forks without blocking, then it gives up, puts whatever forks it has back on the table, and starts over.

**The Details**

* Step 1 - Write a program in a file phil.c that implements the *Left fork first and hold* synchronization strategy.
* Step 2 - Copy your phil.c program to a new one named phil-ordered.c and modify it so that it implements the *resource ordering* synchronization strategy.
* Step 3 - Copy your phil.c program to a new one named phil-random.c and modify it so that it implements the *random* synchronization strategy.

**Provided Materials**

You should use subversion to check out the lab7 directory from your repository, in the same way that you did for the previous labs.

Your lab7 directory should contain the following files:

* Makefile

The provided Makefile contains instructions for compiling and linking the programs phil.c, phil-ordered.c, phil-random.c,

You will need to create all of the other files.