# CprE 308 Laboratory 3: Concurrent Programming with the `pthread` Library

Department of Electrical and Computer Engineering
Iowa State University

## 1 Submission

Fill out the provided lab report template and submit it as a PDF on Canvas. Feel free to adjust the answer boxes if needed.

- 20 pts - A summary of what you learned in the lab session. This should be no more than two paragraphs. Try to get at the main idea of the exercises, and include any particular details you found interesting or any problems you encountered.
- 80 pts - A write-up of each experiment in the lab (as dictated by the report template). For output, take screenshots or copy-paste results from the terminal, and summarize when asked. For program code, include comments that explain the important steps within the program. Include all relevant details.

## 2 Introduction

In this lab you will learn about concurrent programming using pthreads. Some suggested resources for pthreads:

1. https://pubs.opengroup.org/onlinepubs/007908799/xsh/pthread.h.html
2. Manual pages for the pthread functions (`man <function name>`)
3. Section 4 in this lab description
4. Example programs `t1.c`, `t2.c`, and `t3.c`, available on Canvas

Section 3.1 provides a step-by-step tutorial for creating a simple multi-threaded program. In Section 3.2, you will learn how to synchronize among threads that access the same data. You'll work with three example programs (available on Canvas) to learn about mutexes, condition variables, and to practice with a producer-consumer problem.

## 3 Exercises

### 3.1 Creating a simple multi-threaded program

Before beginning this exercise, look over the manual pages for `pthread_create()` and `pthread_join()`. Learn about how they work before you try to use them. Then create a program as follows:

- Open a new C file and add an include for `pthread.h`. Also include `stdio.h`.

- Create two functions called thread1 and thread2. These should have prototypes that allow them to be used with `pthread_create()`. For example:

  ```
  void * thread1(void * ptr);
  ```

  For the implementation, have the two functions print "Hello from thread 1" and "Hello from thread 2," respectively.

- Write a `main()` function as follows:
  - Declare two variables `t1` and `t2` with type `pthread_t`
  - Call `pthread_create()` twice to create both threads `t1` and `t2`
  - After creating the threads, print "Hello from the main thread"

- To compile the program, include `-lpthread` in the `gcc` command. For example:

  ```
  $ gcc -o ex1 ex1.c -lpthread
  ```

  This option will link the pthread library during the compilation process.

Run the program and observe the output. What happens? Well, we created the threads, but we forgot to use the `pthread_join` function to allow them to finish before main terminates.

1. (10 pts) Add a `sleep(5)` statement in the beginning of both thread1 and thread2. Compile and run the program. Do the messages get printed? Why or why not?
2. (5 pts) Add two `pthread_join()` calls to join `t1` and `t2` just before the print statement in `main()`. Compile and run the program. Do the threads' messages get printed? Why or why not?
3. (5 pts) Include your code with comments explaining the usage of `pthread_create()` and `pthread_join()`.

## 3.2 Thread synchronization

In this experiment you will learn how to provide mutual exclusion and synchronization to threads that share data. In pthreads, mutexes are used for mutual exclusion, and condition variables are used as synchronization mechanisms.

### 3.2.1 Mutex

We will first look at code examples that illustrate how threads use mutexes to achieve mutual exclusion in critical regions of code. Look over the manual pages for `pthread_mutex_lock` and `pthread_mutex_unlock`. Then, download `t1.c` from Canvas and study the code.

1. (5 pts ) Compile and run `t1.c`. What is the output value of v?
2. (15 pts) Delete the `pthread_mutex_lock` and `pthread_mutex_unlock` statements in both increment and decrement threads. Recompile and rerun `t1.c`. What is the output value of v? Explain why the output is either the same as, or different than, before.

### 3.2.2 Condition Variables

Look over the manual pages of `pthread_cond_signal` and `pthread_cond_wait`, then examine `t2.c`. The program uses two threads to print "hello world." The thread that prints "world" waits for the other thread to finish printing "hello." This is achieved using a condition variable.

- Modify the program by adding another thread (and function) called `again`. Use a second condition variable to synchronize the three threads so that they print out the statement "Hello World Again!"

- To implement this correctly, you must understand why the "done" flag is necessary. Think about the case where the hello function runs first and sends the signal before world is waiting (you can use a sleep statement to force this case). Note that a signal is not received unless a thread is already waiting for it. Could the world thread sleep forever? When you make your changes, take this problem into account.

1. (20 pts) Include your modified code with comments labeling what you added or changed.

### 3.3 Modified Producer-Consumer Problem

Download `t3.c` from the class website. The goal of this program is to run a group of consumers and a single producer in synchronization. The program will start one producer thread, which runs the function `producer`, and many consumer threads, each of which runs the function `consumer`.

Note that this producer-consumer problem is slightly different than what we covered in class. The producer should produce items only when the number of items in the supply has reached zero. Until this happens, the producer waits. The producer produces 10 items each time. Each consumer thread waits until there is at least one item of supply remaining to consume. It then consumes one item of supply, and exits. When there are no more consumer threads remaining to consume items, the producer also exits.

The program is incomplete. Your task is to fill in the code for the producer and make the program run as described. An outline for the producer has been written for you, but you may modify it as needed. The code for the consumer has already been completed. Note the existence of `pthread_cond_broadcast`, which will signal *all* threads waiting on a condition variable.

1. (20 pts) Include the code for your producer function with comments labeling the key parts.

## 4 Hints

### 4.1 Creating pthreads

The function used to create a pthread is:

```
int pthread_create(pthread_t * thread, pthread_attr_t * attr,
    void * (*start_routine)(void *), void * arg);
```

This will start a thread using the function pointed to by `start_routine`, which is a function pointer. You can think of a function pointer as the starting address of a function. The name of a function in C will act as a function pointer. Note the `pthread_create`'s prototype calls for a function pointer to a function that takes a void pointer and returns a void pointer. The argument of "`start_routine`" is passed separately as the fourth parameter, "arg". If `start_routine` needs more than one argument, "arg" should point to a structure that encapsulates all of the arguments. The code below shows how to typecast the structure to and from a void pointer so the `pthread_create` function can be used correctly.

```
#include <pthread.h>

struct two_args {
    int arg1;
    long arg2;
};

void *foo(void * p) {
    int local_arg1;
    long local_arg2;
    struct two_args * local_args = p;
    local_arg1 = local_args->arg1;
    local_arg2 = local_args->arg2;
    /* continue work */
}

int main() {
    pthread_t t; /* t: identifier for thread */
    struct two_args* ap; /* a pointer to the arguments for "foo" */
    ap = malloc(sizeof(struct two_args));
          /* sizeof returns the number of bytes in the structure */
    ap->arg1 = 1;
    ap->arg2 = 2;

    pthread_create(&t, NULL, foo, (void*)ap);
    /* continue work */
    /* DON'T FORGET TO CALL free() */
}
```

### 4.2 Condition Variables

- Condition variables allow us to synchronize threads by having them wait until a specific condition occurs. In t2.c, the "world" thread waits until the "hello" thread activates the condition. The `pthread_cond_wait` function automatically releases the mutex passed into it, and waits until some other thread signals the corresponding condition variable. When the condition variable is signaled, `pthread_cond_wait` function automatically contends to re-lock the mutex. Assuming that the mutex can be locked, the thread continues executing. If the mutex cannot be locked, the thread waits for an unlock to occur.
- A helpful construct for waiting might look like this:

  ```
  pthread_mutex_lock(&mutex)
  while (can't proceed)
      pthread_cond_wait(&cond, &mutex)
  //Do something...
  pthread_mutex_unlock(&mutex)
  ```

  You must have a mutex locked before waiting. Also, note that while the thread is waiting, the mutex is unlocked. It will be locked again when the thread resumes.
- You will find that there are two ways to signal threads waiting on a condition variable: `pthread_cond_signal`, and `pthread_cond_broadcast`. The latter will allow all threads currently waiting to continue, one after the other. A condition variable signal will be received by a thread only if it is already waiting on that condition variable.

### 4.3 Compilation

You need to explicitly link the thread library in the `gcc` or `g++` command using the option `-lpthread`.

```
$ gcc -lpthread -o threads threads.c
```