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

Department of Electrical and Computer Engineering
Iowa State University

1 Submission

Fill the given `308-lab3-report-template.doc` and submit it through Canvas. Feel free to adjust the answer boxes if needed.

- 10 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. Each experiment has some items you need to include. For output, screen-capture or copy-paste results from your terminal and summarize when necessary. For program code, include comments that explain the important steps within the program. Include all relevant details.

Due: One week after your lab session.

2 Introduction

In this lab session, we will learn about concurrent programming using the `pthread` library in C. You can learn about pthread functions in the following ways:

- Visit website: `https://pubs.opengroup.org/onlinepubs/007908799/xsh/pthread.h.html`
- Use the `man <function name>` command in bash
- Section 4 provides some examples of how to use `pthread` function calls
- Three example programs, `t1.c`, `t2.c`, and `t3.c` can be downloaded from Canvas.

Section 3.1 provides a step-by-step tutorial for you to create a simple multi-threaded program using the `pthread_create()` and `pthread_join()` function calls.

In Section 3.2, you will learn how to synchronize among threads that access the same data. You'll work with the 3 example programs `t1.c`, `t2.c`, and `t3.c` to learn about Mutex, Conditional Variables, and practice with a Producer-Consumer problem.

3 Excercises

3.1 Creating a simple multi-threaded program

Before you start this exercise, read the manual pages of functions `pthread_create()` and `pthread_join()`. Learn about what they do and how to use them. You can also read the three given programs (t1.c, t2.c, t3.c) for more examples.

Create your program with the following steps:

- Create a new C file and include the `pthread.h` header.
- Create two simple functions called `thread1` and `thread2`. For example:

  ```
  void* thread1() { printf("Hello from thread 1\n");}
  void* thread2() { printf("Hello from thread 2\n");}
  ```

- In your main() function:
  - Declare two variables `t1` and `t2` with type `pthread_t`
  - Call the `pthread_create()` function twice to create both threads t1 and t2
  - After creating the 2 threads, print out ”Hello from the main thread”
- To compile the program, include `-lpthread` in the gcc command. For example:

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

  This option will link the pthread library to our program.

1. **(10 pts)** Add a sleep(5) statement in the beginning of both thread1 and thread2 functions. Compile and run the program. Can you see their messages getting printed? Why?
2. **(5 pts)** Add two `pthread_join()` calls for t1 and t2 just before the printf statement in main. Compile and run the program. What is the output now? Why?
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 synchronize threads that share a common data source. Mutexes and conditional variables are used as synchronization mechanisms for pthreads.

3.2.1 Mutex

First, read the manual pages of `pthread_mutex_lock` and `pthread_mutex_unlock`. Then **download t1.c** from Canvas and study its code. In this program, threads use mutexes to exclusively access critical sections.

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 the same, or different.

3.2.2 Conditional Variable

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

(20 pts) Modify the program by adding another thread called "tid_again" and a function called "again". Create a second conditional variable "done_world" to synchronize the three threads so that they print out "Hello World Again!". Include your modified code with comments labeling what you added or changed.

NOTE: To implement this correctly, you must understand why the "done" variable 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 someone is waiting for it first. Could the world thread sleep forever? When you make your changes, take this into consideration.

3.3 Modified Producer Consumer Problem

Download t3.c and briefly examine its code. 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".

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. When there are no more consumer threads remaining to consume items, the producer must exit.

Each consumer thread waits until there is at least one item of supply remaining to consume. It then consumes one item of supply, and then exits.

(20 pts) The program is incomplete - the producer function is not fully implemented. Your task is to fill in the code for the producer and make the program run as described. Include your modified code with comments labeling what you added or changed.

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. If the type is void, how are we going to pass data into the thread? The argument of "`start_routine`" is separately passed through "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 */
}
```

4.2 Conditional 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. Once the `pthread_cond_wait` statement is active, it automatically releases the mutex and waits until it receives an active signal. When the condition receives its signal, the `pthread_cond_wait` function is able to lock the newly available 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)
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 only if another thread is already waiting.

4.3 Compilation

You need to pass the thread library on the `gcc` or `g++` command line with the option `-lpthread`.

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