CprE 308 Laboratory 3: Concurrent Programming with the pthreads Library

Department of Electrical and Computer Engineering
Iowa State University

1 Submission

Fill out the given lab *report-template* and submit as PDF on Canvas.

- **(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. Each experiment has a list of items you need to include. For output, take screenshots or copy-paste results from your terminal and summarize when necessary. Include all relevant details and add comments in your code to explain important steps.

2 Instructions

In this lab you will learn about concurrent programming using `pthread`. To get started:

1. Read the *entire* lab description first!
2. Manual pages of all different functions that you encounter (e.g.: `man <function name>`).
3. Description of each `pthread` function can be found at:
   `https://pubs.opengroup.org/onlinepubs/007908799/xsh/pthread.h.html`
4. Refer to Section 4 in this lab description for further hints on programming.
5. Sample programs: `t1.c`, `t2.c` and `t3.c` are available on Canvas.

3 Exercises

3.1 Programming with `pthread`

In this exercise we will step through a simple program using `pthread`. You may have a look at the manual pages for `pthread_create()` and `pthread_join()` to learn more before you use them.

- Open a new file (`ex1.c`) and include the `pthread.h` along with `stdio.h`.
- Create thread functions:
  - Create two function prototypes called `thread1` and `thread2` (e.g.: "`void* thread1();`"). These will be the methods that our pthreads will run.

  - Have the two functions print "Hello from thread1" and "Hello from thread2" respectively.
  - You may refer to the example code in Section 4, or in `t1.c`, and `t2.c`.

- Code the `main` function
  - Create two variables (`i1` and `i2`) of the datatype `pthread_t`
  - Call `pthread_create()` function twice to create both threads `i1` and `i2`
  - Add a print statement "Hello from main" at the end of the `main` function

- Compile the program using the `-lpthread` option.

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

  This option compiles the code linked with the `pthread` library.

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) To make sure the `main` terminates before the threads finish, add a `sleep(5)` statement in the beginning of the thread functions. Can you see the threads' output? Why or why not?
2. (5 pts) Add two `pthread_join()` statements just before the `printf` statement in `main`. Recompile and rerun the program. Can you see is the output? 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 synchronize threads that share a common data source. Mutexes and conditional variables are used as synchronization mechanisms for pthreads.

3.2.1 Mutex

We will first look at code examples that illustrate how threads use mutexes to exclusively access critical area. Look over the manual pages for `pthread_mutex_lock` and `pthread_mutex_unlock`. Download `t1.c` from Canvas and examine its contents.

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` statement in both increment and decrement threads. Recompile and rerun `t1.c`. W hat is the output value of v? Explain why the output is the same, or different.

3.2.2 Conditional Variable

Next download t2.c. This is a "Hello World" program using threads. The program uses threads to print "hello world". The thread that prints "world" waits for the other thread to finish printing "hello". This is achieved using condition variables. Look into the manual pages for `pthread_cond_signal` and `pthread_cond_wait`.

- Modify the program by adding another thread (and routine) called "again" Use a second conditional 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 someone is waiting for it first. Could the world thread sleep forever? When you make your changes, take this problem into account.

1. (10 pts) Include your modified code with your lab submission and comment on what you added or changed. Label this “t2.c”.

3.3 Modified Producer Consumer Problem

Next, you will work on a producer-consumer type problem that is different from what we discussed in class. So, please read the description carefully.

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”.

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.

Your task is to fill in the code for the producer. The code for the consumer has already been filled in.

1. (20 pts) Include your modified code with your lab submission and comment on 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
```