

Cpr E 308 Lab 3: Concurrent Programming with the pthreads Library  
Department of Electric and Computer Engineering  
Iowa State University  
Fall 2012

## Submission

Include the following in your submission:

- 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.
- A write-up of each experiment in the lab. Each experiment has a list of items you need to include. For output, cut and paste results from your terminal and summarize when necessary. Include all relevant details.
- Submit your summary, write-ups, and modified source code on blackboard. Your submission should be a single archive called *lab03-lastname-section.tar.gz*, where *lastname* is your last name and *section* is your section letter. The following command will create your archive:

```
tar -czvf lab03-lastname-section.tar.gz folder/
```

## Resources

1. Manual pages of all different functions that you encounter.

2. Read the following website:

[http://www.llnl.gov/computing/tutorials/pthreads/man/pthread\\_cond\\_wait.txt](http://www.llnl.gov/computing/tutorials/pthreads/man/pthread_cond_wait.txt)

You might want to check other commands in that Web site as well.

3. Refer to section 4 for further hints on programming

## Exercises

### 1. Programming with pthreads

In this exercise you will learn about concurrent programming using pthreads. We will step through a simple program using pthreads:

- Open a new file and include the `pthread.h` file. Create two new function prototypes called `thread1` and `thread2` (i.e. `void* thread1( ) ;`). These will be the methods that our pthreads will run. Look at the example code in Section 4, or in `t1.c`, and `t2.c`.
- Code the main function. Make two instances of the data type `pthread_t`. Call them `"i1"` and `"i2"`.
- Create the threads using the `pthread_create` function. Create both threads, passing similar arguments as shown in the example in the help section. The last argument may be passed a value of `NULL`, since we don't pass any arguments to our `thread1` and `thread2` functions.
- After the main function write the `thread1` and `thread2` functions. Just have each function print a message to the screen identifying itself (i.e. `printf("Hello,I am thread 1\n");`). Add a print statement in the end of the main function (i.e. `printf("Hello,I am the main process\n");`).
- Compile the program using the `-lpthread` option (e.g. `gcc -o ex1 ex1.c -lpthread`) to compile with the pthread library.

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

- (6 pts ) *To make sure the main terminates before the threads finish, add `sleep(5)` statement in the beginning of the thread functions. Can you see the output of threads? Why?*
- (2 pts ) *Add the two `pthread_join` statements just before the `printf` statement in main. Pass a value of `NULL` for the second argument. Recompile and rerun the program. What is the output? Why?*
- (2 pts ) *Include your commented code with your submission – label it "pthread example".*

## 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.

### 2.1 Mutex

We will first look at code examples that illustrate how threads use mutexes to exclusively access critical area. Download `t1.c` from the class website and examine its contents.

- (2 pts ) Compile and run `t1.c`, what is the output value of `v`?
- (8 pts) Delete the `pthread_mutex_lock` and `pthread_mutex_unlock` statement 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.

## 2.2 Conditional Variable

Next download `t2.c` from the class website. 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.

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

## 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.


- ( 20 pts ) Include your modified code with your lab submission and comment on what you added or changed. Label this “`t3.c`”.

## 4. Hints

### 4.1 Creating pthreads

The syntax of pthread creation is as follows:

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



Points to  
argument for  
start\_routine

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 pthread\_create has 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;  
    int agr2  
};  
  
void *foo(void * p)  
{  
    /* one way to retrieve the arguments is to cast the pointer back to the  
    correct data type so the arguments can be accessed. */  
    int local_arg1;  
    int local_arg2;  
    struct two_args * local_args = p;  
  
    local_arg1 = local_args->arg1;  
    local_arg2 = local_args->arg2;  
}  
  
main()  
{  
    pthread_t t; /* t: identifier for thread */  
    struct two_args* ap ; /* a pointer to the arguments for "foo" */  
  
    /* sizeof returns the number of bytes in the structure */  
    ap = (struct two_args*) malloc(sizeof(struct two_args));  
    ap->arg1 = 1;  
    ap->arg2 = 2;  
  
    /* pthread takes a function pointer to a function that takes a pointer  
    and returns a pointer. Since there is no way to know the type in  
    advance, the type is set to void and left for the programmer to cast */  
    pthread_create(&t, NULL, foo, (void*)ap);  
    ...  
}
```

## 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/g++` command line with the option `-lpthread`. Ex: `gcc -lpthread -o threads threads.c` –