CSCI 4100 – Assignment 3 – Threads and Locks

# Instructions

For this assignment you will be implementing a bounded buffer using threads and locks and testing it with multiple senders and receivers. I have provided structures and prototypes for your bounded buffer in the file bbuff.h. You will need to implement the functions I have specified in a file called bbuff.c.

There are two structures:

* bb\_msg is a structure that represents a simple message. It has two fields: t\_id represents the id number of the thread that is sending the message and m\_id represents the id number of the message itself.
* bbuff is a structure that represents the buffer itself. Note that it uses a predefined value BUFFER\_SIZE to represent the capacity of the buffer.

There are three functions having to do with messages:

* bb\_init\_msg is used to initialize a message with a given thread id and message id. It is passed a pointer to a message that has already been allocated in memory.
* bb\_copy\_msg is used to copy the contents of one message (the source) into another message (the destination). Both messages are passed as pointers and have already been allocated in memory.
* bb\_display\_msg is used to display the contents of a message along with the id of the receiving thread to standard output. This can be done using the printf function, which uses format codes to insert values into a string. For example:
* printf("num1 is %d and num2 is %d.", num1, num2);
* will display the following if num1 has a value of 5 and num2 has a value of 8:
* num1 is 5 and num2 is 8.

There are three functions having to do the bounded buffer:

* bb\_init is used to initialize a bounded buffer. The in and out variables are initialized to zero. See the section on **POSIX Threads and Locks** for how to initialize the lock.
* bb\_send is used to send a message to the buffer. It should do the following:
  1. Acquire the buffer lock.
  2. While the buffer is full, release the lock, then acquire it again so that a receiving thread can access the buffer.
  3. Copy the message from the bb\_msg structure provided to the correct location in the buffer using the bb\_copy\_msg function.
  4. Increment the buffer’s in member variable.
  5. Release the buffer lock.
* bb\_receive is used to receive a message from the buffer. It should do the following:
  1. Acquire the buffer lock.
  2. While the buffer is empty, release the lock, then acquire it again so that a sending thread can access the buffer.
  3. Copy the message from the correct location in the buffer to the bb\_msg structure provided using the bb\_copy\_msg function.
  4. Increment the buffer’s out member variable.
  5. Release the buffer lock.

## POSIX Threads and Locks

The standard thread library for UNIX-based systems is **POSIX Threads** (aka **Pthreads**.) To use this library you will need the following include statement:

#include <pthread.h>

To create and use a thread you must do the following:

1. Declare a variable of type pthread\_t.
2. Write a function whose argument and return value are both of type void \* (this means the function can take and return pointers to anything.)
3. Create a structure that can hold all of the information that the thread will need to execute, and create an instance of this structure containing the information for this thread.
4. Start the thread using the following function call:

* pthread\_create(&my\_thread, NULL, my\_function, &args);
* where my\_thread is the thread variable from step 1, my\_function is the name of the function from step 2, and args is the structure from step 3.

1. Wait for the thread to complete using the following function call:

* pthread\_join(my\_thread, NULL);

The POSIX thread library also has support for locks that can guarantee mutually exclusive access to shared data. To create and use a lock you must do the following:

1. Create a variable of type pthread\_mutex\_t.
2. Initialize this variable using the following function call:

* pthread\_mutex\_init(&my\_lock, NULL);

1. When the currently running thread needs to acquire the lock, use the following function call:

* pthread\_mutex\_lock(&my\_lock);

1. When the currently running thread needs to release the lock, use the following function call:

* pthread\_mutex\_unlock(&my\_lock);

## Testing the Bounded Buffer

You must also write code to test your bounded buffer with multiple sending and receiving threads. This functionality should be implemented in a file called thread\_tester.c. I have provided a structure and three prototypes to help you do this.

The t\_args structure is used to store all of the arguments that will be required for a sending or receiving thread to get started. It contains the following data members:

* t\_id is the id number for the thread.
* num\_msgs is the number of messages for the thread to send or receive.
* buffer is a pointer to the buffer that the thread is to use in order to send or receive messages.

A structure of type t\_args can be initialized using the t\_args\_init function. This function sets all of the fields of the t\_args structure using the given parameters.

The send\_msgs function is executed by a sending thread. It must do the following:

1. Cast the void pointer args to a pointer to a t\_args structure using the following statement:

* struct t\_args \* real\_args = (struct t\_args \*) args;

1. Declare a bb\_msg structure. (Note that in C you have to include the keyword struct when declaring a structure.)
2. Send the number of messages specified in the t\_args structure by repeatedly doing the following:
   1. Initialize the bb\_msg structure with the appropriate thread id and message id using bb\_init\_msg.
   2. Send the message to the buffer in the t\_args structure using bb\_send.
3. Return NULL.

The receive\_msgs function is executed by a receiving thread. It must do the following:

1. Cast the void pointer args to a pointer to a t\_args structure.
2. Declare a bb\_msg structure.
3. Receive the number of messages specified in the t\_args structure by repeatedly doing the following:
   1. Receive the message from the buffer in the t\_args structure using bb\_receive.
   2. Display the message along with the thread id from the t\_args structure using bb\_display\_msg.
4. Return NULL

Your main function should do the following:

1. Declare a pthread\_t variable for each of the senders and the receivers.
2. Declare a bbuff structure and initialize it using bb\_init.
3. Declare a t\_args structure for each of the sending and receiving threads and initialize them using t\_args\_init.
4. Start each of the sending threads using pthread\_create, the send\_msgs functions, and the appropriate t\_args structure.
5. Start each of the receiving threads using pthread\_create, the receive\_msgs function, and the appropriate t\_args structure.
6. Wait for each of the sending and receiving threads to complete using pthread\_join.

# Compiling and Running Your Code

To compile this code you should use the gcc compiler on like you did for Assignments 1 and 2, only this time you will need to include all of your .c files as command line arguments and explicitly tell the compiler to link with the POSIX thread library:

gcc -o thread\_tester thread\_tester.c bbuff.c -lpthread

Here is an example of a run of the code using 3 senders and 4 receivers and a total of 24 messages:

./thread\_tester  
[sending thread: 1, message 0, receiving thread: 0]  
[sending thread: 1, message 4, receiving thread: 0]  
[sending thread: 1, message 1, receiving thread: 1]  
[sending thread: 1, message 6, receiving thread: 1]  
[sending thread: 1, message 7, receiving thread: 1]  
[sending thread: 0, message 0, receiving thread: 1]  
[sending thread: 0, message 1, receiving thread: 1]  
[sending thread: 2, message 0, receiving thread: 1]  
[sending thread: 1, message 3, receiving thread: 3]  
[sending thread: 2, message 1, receiving thread: 3]  
[sending thread: 2, message 2, receiving thread: 3]  
[sending thread: 0, message 2, receiving thread: 3]  
[sending thread: 1, message 5, receiving thread: 0]  
[sending thread: 2, message 3, receiving thread: 0]  
[sending thread: 0, message 4, receiving thread: 0]  
[sending thread: 0, message 5, receiving thread: 0]  
[sending thread: 0, message 3, receiving thread: 3]  
[sending thread: 0, message 6, receiving thread: 3]  
[sending thread: 1, message 2, receiving thread: 2]  
[sending thread: 0, message 7, receiving thread: 2]  
[sending thread: 2, message 4, receiving thread: 2]  
[sending thread: 2, message 5, receiving thread: 2]  
[sending thread: 2, message 6, receiving thread: 2]  
[sending thread: 2, message 7, receiving thread: 2]

Note that the order of these operations is nondeterministic, so it will probably be different when you test it.

# What to Hand In

Your source files should have comments at the top listing your name, CSCI 4100, Assignment 3, and a brief explanation of what the program does. Put your files into a zip file then upload the zip file to D2L in the dropbox called Assignment 3.

If you don’t have access to a zip archiver you can use the following commands to create a zip archive in Linux:

$ mkdir yourlastnameAssign3  
$ mv thread\_tester.c bbuff.c bbuff.h yourlastnameAssign3  
$ zip -r yourlastnameAssign3.zip yourlastnameAssign3

This will create a zip file called yourlastnameAssign3.zip which you can upload to D2L.