CSCI 4100
Assignment 6
Threads and Locks

Learning Outcomes

Implement a bounded buffer using threads and locks.

Required Reading

Saltzer & Kashoek 5.2

Instructions

For this assignment you will be implementing a bounded buffer using threads and locks and testing it with multiple senders and receivers.

The Bounded Buffer

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. It is based on the bounded buffer with locks from Section 5.2 of the PoCSD textbook. Note that it uses a predefined value `BUFFER_SIZE` to represent the number of messages that can be stored in 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 below 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.
5. 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);
   ```

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

   ```
   pthread_mutex_lock(&my_lock);
   ```

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

   ```
   pthread_mutex_unlock(&my_lock);
   ```

Testing the Bounded Buffer

In addition to the bounded buffer code above 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;
   ```

2. Declare a `bb_msg` structure. (Note that in C you have to include the keyword `struct` when declaring a structure.
3. Send the number of messages specified in the `t_args` structure by repeatedly doing the following:
   (a) Initialize the `bb_msg` structure with the appropriate thread id and message id using `bb_init_msg`.
   (b) Send the message to the buffer in the `t_args` structure using `bb_send`.
4. 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:
   (a) Receive the message from the buffer in the `t_args` structure using `bb_receive`.
   (b) 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`. The number of messages sent or received by each thread is up to you, but the total number of messages sent should equal the total number of messages received.
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`. Note that in order for the threads to run concurrently you must "create" all of the threads before you "join" any of the threads.

Compiling and Running Your Code

To compile this code you should use the `gcc` compiler on the Linux server like you did for Assignment 3, 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 -lpthread -o thread_tester thread_tester.c bbuff.c
```

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]
```

What to Hand In

Download the source files to your local machine, put them into a zip file then upload the zip file to D2L in the dropbox called Assignment 6.

If you don't have access to a zip archiver you can use the following commands to create a zip archive on the linux server:

```
$ mkdir prog6
$ mv thread_tester.c bbuff.c bbuff.h prog6
$ zip -r prog6.zip prog6
```

This will create a zip file called `prog6.zip` which you can download to your local machine.