

Write a POSIX/LINUX program that will implement the bounded buffer problem. We must do a couple of things to make this thing work under multi-threading. First, busy waits won't work, so we have to do something else. And it's really a waste to leave one buffer segment empty while the algorithm is running. So we will use a count variable, but will provide protection for the count variable.

Start your program like this.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
```

We need a variable to protect our Count variable. Declare the variable this way:

```
pthread_mutex_t CountMutex;
```

Rather than using a busy wait, we will have threads go to sleep, and be awakened when the condition they are waiting for becomes true. To do this we need condition variables. These variables allow a thread to go to sleep, and allow another thread to wake it back up again. We have a sender and a receiver, they can both go to sleep and wake up, so we will make two condition variables for them using the following syntax.

```
pthread_cond_t SenderCond;
pthread_cond_t ReceiverCond;
```

Set the buffer size to 10, using a #define. Our messages will be random integers obtained using the rand( ) function. Send 100 messages from the sender to the receiver. The main program (also known as the main thread) must initialize all variables, start both the sender and receiver threads, and then exit. Don't do any sending or receiving in the main thread.

### Initializing Variables.

Initialize your Mutex and Cond variables using the following statements. You have to do this before you can use them.

```
pthread_mutex_init(&CountMutex,NULL);
pthread_cond_init(&SenderCond,NULL);
pthread_cond_init(&ReceiverCond,NULL);
```

### Starting a new thread.

Before starting a new thread, we need to create a thread variable and a thread attributes variable by using the following three statements.

```
pthread_t ThreadID;
pthread_attr_t ThreadAttr;
pthread_attr_init(&ThreadAttr);
```

After creating the required variables, we use the following statement to start a new thread.

```
pthread_create(&ThreadID,&ThreadAttr,Sender,NULL);
```

In the above Sender must be a function declared as follows:

```
void * Sender(void * x);
```

### Protecting the counter variable.

To protect the counter variable, surround the sending and receiving code with Mutex calls as follows.

```
pthread_mutex_lock(&CountMutex);  
...  
Sending (Receiving) code  
...  
pthread_mutex_unlock(&CountMutex);
```

### Going to sleep.

When you detect a condition that your thread should wait for (Sender waits when  $Count \geq BufferSize$  and Receiver waits when  $Count=0$ ), use the following function call to put the current thread to sleep. This will release the control of CountMutex so the other thread can run.

```
pthread_cond_wait(&SenderCond,&CountMutex);
```

Before going to sleep, you should set the value of a global variable so that the Sender (for example) will be able to tell that the Receiver is sleeping.

### Waking up.

After sending a message, the sender must check to see if the receiver is sleeping. If so, it must wake up the receiver using the following function call. Do this BEFORE unlocking CountMutex. The receiver should do the same thing for the sender.

```
pthread_cond_signal(&ReceiverCond);
```

Output

To prove your program is working, print each message before it is sent and after it is received in a format like the following: "Sent 128384524" "Received 4848237293".

**TURN IN:** A printout of your code and a floppy or a CDR containing your source code. (Do a file transfer to Windows if you have to.)