Atomic Broadcast

March 6, 2014

Joe Houng

Abstract

With the concepts of semaphores, there are many ways to implement synchronization for a multi-thread problem. One being an atomic broadcast.

Introduction

Atomic Broadcast Problem:
Assume one producer process and n consumer processes share a buffer. The producer deposits messages into the buffer, consumers fetch them. Every message deposited by the producer has to be fetched by all n consumers before the producer can deposit another message into the buffer. Program a solution this problem in two ways:

Semaphore solution: Develop a solution to this problem using Pthreads with semaphors for synchronization.

Condition variable solution: Develop a solution to this problem using Pthreads with condition variables for synchronization.

Approach

Both solutions have the same kind of structure and share a few global variables. The global variables are :

```
int thread_count; //total number of consumer threads;
int message; //the message that the producer sets, it will be a random number.
int r; //the random number;
```

That structure being that all consumers are generated in a for loop and then using:

```
pthread_create(&thread_handles[thread], NULL, Consumer, (void*) thread);
```

I keep all the consumer threads in pthread variable thread_handles so that I can pthread_join all of them easily in the end so that the next message is not broadcasted till all threads are done. The third parameter is the Consumer function which takes in a thread number as a parameter for debugging purposes.

Inside the loop that creates all the consumer threads, I check for the condition that the loop iteration number is equal to the random variable r % thread_count. This way the producer thread would begin at some random part of the program so to further indicate my implementation is accurate.

```
pthread_create(&idp, NULL, Producer, NULL);
```

There really isn't much use to the idp thread name, I just put it there out of the habit of naming things. The function that this thread calls is Producer.

Where the Semaphore and Conditional Variable programs differ is in the pthread function calls.

Semaphore:

Global variables:

```
sem_t ProductNum; //keeps track of number of products produced, init at 0
sem_t AllConsumed; //track consumption, init at -(thread_count -1)
sem_t mutex; //initiated at 1
```

Consumer function:

First the mutex is locked. Then the sem_wait is called on ProductNum. Since sem_wait is called on ProductNum, the consumer thread will hold on to the mutex as he waits for the producer. Since the mutex isn't free no other consumers can try to sem_wait on ProductNum. After the waiting ends the AllConsumed semaphore is incremented. Incrementing this semaphore will eventually indicate that the producer can generate another message.

Producer function:

First the producer sets the global message variable so that the consumers can grab it. Then inside a for loop from 0 to the number of consumers, the ProductNum semaphore is incremented. This will allow previous and next total consumers consume till ProductNum = 0. Then the producer waits till the AllConsumed semaphore is = 1. The AllConsumed semaphore will always = 1 because of the way I initialized this semaphore and how many consumers there are.

Conditional Variables:

Global Variables:

```
int consumeOK; //indicates when the producer has produced
pthread_mutex_t mutex;
pthread_cond_t cv;
int count; //indicates when the producer can produce again.
```

Consumer function:

First I put the thread into a busy while loop till the integer variable consumeOK is equal to 1. I do this so that each thread that is ran before the producer will wait till the producer gives the consumeOK signal. Then the count integer variable is incremented, once the count variable is greater than the number of the consumer threads, the last consumer will signal the producer that it is ok to produce again.

Producer function:

First the producer is locked in a mutex, the message is set, then the consumeOK

variable is set to 1 to indicate to the existing consumer threads that it is ok to consume. Then, while the number of consumers that has consumed the message is less than the number of consumer threads, the producer will wait on the cv variable, and release the mutex. This way the producer will not go till enough consumer threads are through, and the last thread executes the pthread_cond_signal(&cv) signal.

Use of the mutex in the Conditional Variable solution may have not been needed since there is only one producer and the consumers will wait with or without the mutex. I included the mutex just for good measures.

Materials

- CF405 lab machine with 16 physical cores

Results

The results indicate that my solutions worked. All threads received and printed out the same message from the producer. All threads waited till the producer thread was ran and the message was set. And the producer did not send another message till all threads got the message. This was indicated by putting my program in a loop that ran 10 times, and at the beginning of each loop the main program output what itteration of the loop it was on.

Conclusion

In conclusion, I think the conditional variable solution was much easier to implement than the semaphore solution because conditional variables just make more sense to me.

Run Instructions

Use the makefile by running “make ABcastSem” for the semaphore solution, or “make ABcastCV” for the conditional variable solution.