Lab 8: Solving the Producer Consumer Problem with PThreads

In this lab, we use PThreads to study the canonical Producer Consumer Problem. The solutions we examine achieve synchronization through different mechanisms: busy waiting and condition variables.

In this lab we will ensure you see how to use some shared variables to both exchange information between a producer and consumer in a queue, as well as tracking the state of the information exchange queues.

- Queues
- State Variables
- Busy Waiting
- PThreads Mutexes
- PThreads Condition Variables

The goal of the laboratory experience is to give you an appreciation of the variations in behavior that can result from the concurrency present in a mutli-threaded application, and to give you some experience with how to control that concurrency to ensure correct computation behavior.

Lab Materials

1. Slides
2. Lab Files

Assignment

You need to complete the implementation of producer_consumer.c to work with an arbitrary number of producer and consumer threads. producer_consumer takes as its arguments the number of producer and consumer threads it will use. The producers (combined) should produce exactly QMAX integers, and the consumers (combined) should consume exactly QMAX integers.

The makefile as provided will build the **producer_consumer** executable when you type "bash> make" and as required when you use any of the targets running tests. The test targets provided run the application with all combinations of one consumer, one producer, five consumers, and five producers. The tests produce a narrativeX.raw file that provides the narative statements in the order produced by all the threads running concurrently. For example, this is the raw output for one producer and one consumer produced by our example solution code. The stater code has not concurrency control implemented, so its output will be quite different.

```
  P-1 C-1
************
    RAW
************
prod 0:  0.
prod 0:  1.
```

```
prod 0:  2.
prod 0:  3.
prod 0:  4.
prod 0:  FULL.
con 0:   0.
con 0:   1.
con 0:   2.
con 0:   3.
con 0:   4.
prod 0:  5.
prod 0:  6.
prod 0:  7.
prod 0:  8.
prod 0:  9.
con 0:   5.
con 0:   6.
con 0:   7.
con 0:   8.
con 0:   9.
prod 0:  10.
prod 0:  11.
prod 0:  12.
prod 0:  13.
prod 0:  14.
con 0:   10.
con 0:   11.
con 0:   12.
con 0:   13.
con 0:   14.
con 0:   EMPTY.
prod 0:  15.
prod 0:  16.
prod 0:  17.
prod 0:  18.
con 0:   15.
con 0:   16.
con 0:   17.
con 0:   18.
con 0:   EMPTY.
prod 0:  19.
prod 0:  20.
prod 0:  21.
prod 0:  22.
con 0:   19.
con 0:   20.
con 0:   21.
con 0:   22.
con 0:   EMPTY.
prod 0:  23.
prod 0:  24.
prod 0:  25.
prod 0:  26.
con 0:   23.
con 0:   24.
con 0:   25.
con 0:   26.
con 0:   EMPTY.
prod 0:  27.
prod 0:  28.
prod 0:  29.
prod 0: exited
```

```
con 0:   27.
con 0:   28.
con 0:   29.
con 0:  exited
```

The narrativeX.sorted file groups the narrative statements of each thread together, but keeps them in the order produced. The raw file shows how actions of different threads are interleaved, and the sorted file shows the order in which each thread experienced execution.

The producers note when they see the queue as FULL and the consumers note when they see it as EMPTY. Note the frequency with which consecutive FULL or EMPTY statements occur in the narrative for a single thread. This shows how often a thread is unblocked only to block again. You should run each test several times to see how much behavior varies from run to run. For example, this is the sorted output from our example solution code for 5 producers and 5 consumers.

```
  P-5 C-5
************
  Sorted
************
con 0:   0.
con 0:   1.
con 0:   2.
con 0:   3.
con 0:   4.
con 0:   21.
con 0:   22.
con 0:   23.
con 0:   24.
con 0:   25.
con 0:   26.
con 0:   27.
con 0:   EMPTY.
con 0:   28.
con 0:  exited

con 1:   5.
con 1:   6.
con 1:   7.
con 1:   8.
con 1:   9.
con 1:   10.
con 1:   11.
con 1:   EMPTY.
con 1:   29.
con 1:  exited

con 2:   12.
con 2:   13.
con 2:   14.
con 2:   15.
con 2:   16.
con 2:   17.
con 2:   18.
con 2:   EMPTY.
con 2:   19.
con 2:   20.
con 2:  exited
```

```
con 3:  exited

con 4:  exited

prod 0:  0.
prod 0:  1.
prod 0:  2.
prod 0:  3.
prod 0:  4.
prod 0:  FULL.
prod 0:  FULL.
prod 0:  FULL.
prod 0:  FULL.
prod 0:  8.
prod 0:  FULL.
prod 0: exited

prod 1:  FULL.
prod 1:  5.
prod 1:  FULL.
prod 1:  FULL.
prod 1:  FULL.
prod 1:  FULL.
prod 1:  9.
prod 1:  FULL.
prod 1:  10.
prod 1:  FULL.
prod 1:  17.
prod 1:  FULL.
prod 1:  19.
prod 1:  20.
prod 1:  21.
prod 1:  22.
prod 1:  23.
prod 1:  FULL.
prod 1:  24.
prod 1:  25.
prod 1:  FULL.
prod 1:  26.
prod 1:  27.
prod 1:  28.
prod 1:  29.
prod 1: exited

prod 2: exited

prod 3:  FULL.
prod 3:  FULL.
prod 3:  6.
prod 3:  FULL.
prod 3:  FULL.
prod 3:  FULL.
prod 3:  FULL.
prod 3:  16.
prod 3:  FULL.
prod 3:  18.
prod 3: exited

prod 4:  FULL.
prod 4:  FULL.
```

```
prod 4:  FULL.
prod 4:  7.
prod 4:  FULL.
prod 4:  FULL.
prod 4:  11.
prod 4:  12.
prod 4:  13.
prod 4:  14.
prod 4:  15.
prod 4:  FULL.
prod 4:  FULL.
prod 4:  FULL.
prod 4: exited
```

There are several points of interest here. First, consumers 3 and 4 and producer 2 never got to do any productive work, and also never blocked on a full or empty queue. The luck of the draw on this run simply had the scheduler not choose them until there was nothing left to do. Also, we can see that many producers were awakened ony to find the queue full. This is because we specify you use **pthread_cond_broadcast()** which signals all waiters on a conditional variable. IF there is only one slot in the queue open, only one can win the race and the others may run only to see they should sleep again. This is an example of the *thundering herd* problem.

There are several other patterns of behavior that you can observe in the output from various runs of the program. The output also varies with the parameters used for the **do_work()** routine, which simulates work done by each thread outside the critical section. The code as provided uses (5,50) for the consumer which says to execute the CPU bound inner loop 5 times and then block for 50 milliseconds. The producer sets both of these to zero, which obviously makes it more likely that the queue will become full. Try different settings. Even modest amounts of production overhead make it much less likely for the queue to fill.

When you're finished

After you have finished your implementation, you should go over the following questions, you may be quizzed over them:

1. Briefly describe the problem you are trying to solve and describe the design of your solution.
2. Discuss why the busy-wait solution is inefficient for threads running on the same CPU, even though it produces the "desired behavior".
3. Why are you confident your solution is correct? You will need to argue from your narrated output as to why your solution is correct. Note, your output will likely not match the output listed here exactly. Two successive runs of your application will probably not match even vaguely, due to random variations in how threads are scheduled. However, your report should discuss each of the following points and discuss how your output supports your discussion of each:

    - Are each producer and consumer operating on a unique item? Is each item both produced and consumed?
    - Do the producers produce the correct number of items and the consumers consume the correct number of items?
    - Does each thread exit appropriately?
    - Does your solution pass the above tests with the different numbers of producer and consumer threads in the Makfile tests?

You also need to tar up your lab for submission. For this step, you should use the 'tar' target included in the lab's Makefile. Change the STUDENT_ID variable in the Makefile to your student ID and type:

```
make tar
```

This tar file should then be turned into Blackboard.

< Back to the Lab Home Page