# Solving the Producer – Consumer Problem with PThreads

Michael Jantz
Dr. Prasad Kulkarni
Dr. Douglas Niehaus

# Introduction

- This lab is an extension of last week's lab.
- This lab builds on the concepts learned from the PThreads Intro lab and Dining Philosophers lab.
- Go ahead and make and tag the starter code for this lab:
  - tar xvzf eecs678-pthreads_pc-lab.tar.gz;
  - cd pthreads_pc; make; ctags -R
- Helpful man pages for today:
  - pthread_mutex_lock, pthread_mutex_unlock, pthread_cond_signal, pthread_cond_wait

# The Producer - Consumer Problem

- The producer – consumer problem is a common implementation pattern for cooperating processes or threads. Put simply, a **producer** produces information that is later consumed by a **consumer**. Traditionally, the problem is defined as follows:
  - To allow the producer and consumer to run concurrently, the producer and consumer must share a common buffer.
  - So that the consumer does not try to consume an item that has not yet been produced, the two processes must be synchronized.
  - If the common data buffer is bounded, the consumer process must wait if the buffer is empty, and the producer process must wait if the buffer is full.
- There are many examples of this problem in “real world” applications.
  - e.g., a build system may run multiple processes concurrently. The compiler process will produce assembly code to be read by the assembler process.
  - If you have done socket programming, you used a socket as the data buffer between your client and server processes. Essentially, the operating system handled your synchronization issues.

# An Instance of the PC Problem

- In producer_consumer.c, there is an instance of the PC problem.
- Producer threads and consumer threads are each created. They each share a bounded length FIFO queue.
- Producers place integers into the queue starting at 0 and ending at some predefined maximum (call it WORK_MAX). Producers announce each item they produce.
- Consumers remove one integer at a time from the queue, reporting each as it is consumed. Each consumers exits after WORK_MAX total removals by all consumers.

# The Proposed Solution

- In order to solve the PC problem, the producer_consumer.c as it was distributed "proposes" the following "solution":

```
void *producer (void *q)
{
  fifo = (queue *)q;
  while (1) {
    do_work(PRODUCER_CPU, PRODUCER_BLOCK);
    while (fifo->full && *total_produced != WORK_MAX) {
      printf ("prod %d:\t FULL.\n", my_tid);
    }
    if (*total_produced == WORK_MAX) {
      break;
    }
    item_produced = (*total_produced)++;
    queueAdd (fifo, item_produced);

    printf("prod %d:\t %d.\n", my_tid, item_produced);
  }
  return(NULL);
}
```

```
void *consumer (void *q)
{
   fifo = (queue *)q;
   while (1) {

      while (fifo->empty && *total_consumed != WORK_MAX) {
        printf ("con %d:\t EMPTY.\n", my_tid);
      }
      if (*total_consumed == WORK_MAX) {
        break;
      }
      queueRemove (fifo, &item_consumed);
      (*total_consumed)++;
      do_work(CONSUMER_CPU,CONSUMER_CPU);
      printf ("con %d:\t %d.\n", my_tid, item_consumed);
  }
  return(NULL);
}
```

- Essentially, when the queue is empty the consumer simply spins (as it has nothing to do), and when the queue is full the producer will spin (as *it* has nothing to do).

# Problems With the Solution

- One of the problems with this solution is that it contains a race condition. Suppose the scheduler created the following interleaving:

| Producer | Consumer |
|---|---|
| | Remove an item from the queue.<br>A check to see if the queue is empty shows that it is. |
| Add last items to the queue.<br>Set queue->empty = 0<br>i == WORK_MAX, producer exits. | |
| | Set queue->empty = 1.<br>Set queue->full = 0<br>Start spinning. |

# Imposing Mutual Exclusion

- The basic problem with this solution is that the critical section of code, those accessing the shared total_produced, total_consumed, and queue variables, are not executed atomically
    - For example, producer thread can preempt the consumer thread after the consumer has checked if the list is empty but before it has set the *fifo->empty* variable (and vice versa).
- Use *pthread_mutex_lock( )* and *pthread_mutex_unlock( )* to enforce mutual exclusion for the critical sections modifying the queue
- HINT: The mutex you should use has already been initialized and is called *mutex* in the queue structure. Also, your mutex calls should not surround the busy-wait loops yet.
- The modified solution protects the update of the queue itself but not the check of the queue->empty and queue->full flags, so it doesn't eliminate all problems of race conditions
- The solution also suffers from the problem of wasteful busy waiting.

# Another Synchronization Problem

- Suppose the scheduler created the following interleaving after mutual exclusion is imposed on the queue manipulation
- Producer 2 will be adding an item to a queue that is full

| Producer 1 | Producer 2 |
|---|---|
| | A check to see if the queue is full shows that it is not. |
| A check to see if the queue is full shows that it is not.<br>pthread_mutex_lock( )<br>A check on total_produced<br>Add an item to the queue. The queue is full now.<br>pthread_mutex_unlock( ) | |
| | pthread_mutex_lock( )<br>A check on total_produced<br>Add an item to the queue<br>pthread_mutex_unlock( ) |

# Busy Waiting Problem

- Even if the thread is lucky enough to avoid this race condition, we are still wasting a lot of cycles by forcing each thread to spin when the conditions required for them to continue are not met.
    - Many “FULL” and “EMPTY messages can be printed
    - Try 'bash> grep “EMPTY” narrative1.raw | wc'
        - There were 43,000 in our example run
- You can comment out the busy-wait printfs to see behavior differently
- Consider how many wasted cycles are executed. And this is *with* a blocking print statement!
    - Also there is still a race wrt the empty and full flags
- Condition variables were invented to help with these kinds of situations among others

# Signal and Wait

- The *pthread_mutex_lock( )* and *pthread_mutex_unlock( )* library calls are implemented using more primitive methods provided by the operating system (via the futex system call).
  - A process calling *wait(fred)* will attempt to acquire the mutex fred if it is available. If it is not available, the calling process will insert itself into a list of waiters associated with the mutex *fred*, and will *block* (i.e. remove itself from the scheduler's list of processes ready to run). This is essentially the operation of *pthread_mutex_lock*.
  - A process calling *signal(fred)* will *wakeup* a process in *fred's* waiters list if one is present (i.e. change its blocked state to ready and place it on the scheduler's ready list). This is essentially the operation of *pthread_mutex_unlock*.

# Condition Variables

- The question becomes, can we use *block* and *wakeup* to implement more precise and efficient synchronization?
- It turns out we can. And the POSIX standard provides a component called a *condition variable* that makes using these primitives convenient and intuitive.
  - A condition variable has its own list of waiters associated with it.
  - When a program reaches a point where it should wait for some condition to be true, it blocks and inserts itself into the condition variable's waiters list using *pthread_cond_wait( )*
  - When the condition is met, any process with access to the condition variable can wakeup a process on the condition variable's waiters list, *pthread_cond_signal( )*, or all waiters, *pthread_cond_broadcast( )*

# Library Calls

- *pthread_cond_wait( )* forces a thread to block until a certain condition has been signaled:
  - int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
  - This call atomically unlocks the associated mutex and waits for the condition variable *cond* to be signaled.
  - It requires that the associated mutex must be locked by the calling thread on entrance to this call. Before returning to the calling thread, pthread_cond_wait re-acquires the associated mutex.
- *pthread_cond_signal()* signals a particular condition variable:
  - int pthread_cond_signal(pthread_cond_t *cond);
    - This call restarts one of the threads that are waiting on the condition variable *cond.*
    - If no threads are waiting on *cond* nothing happens.
    - If several threads are waiting on *cond*, exactly one is restarted, but it is not specified which.
  - Use pthread_cond_broadcast(pthread_cond_t *cond) to wake all threads waiting on a particular condition variable
- Note: a mutex is used to ensure operations on the condition variable are atomic

# Modifying producer_consumer.c

- Modify producer_consumer.c to make use of the condition variable library calls described above to handle one producer and one consumer
- When you are through, the producer and consumer should not spin until the condition they are waiting on is met, but should actually block their own execution. They should be signaled to wake up when that condition is satisfied.
- As a hint, the starter code has initialized all the mutexes and condition variables you should need. You can use the same mutex as before to associate with the condition variables. The condition variables created and initialized are fairly obvious: *fifo->notFull*, and *fifo->notEmpty*.

# Output

- When you are through, the output of the printf statements will provide information by which you can verify the semantics specified on the previous slide are present
  - Makefile targets: test1, test2, test3 and test4 run with various numbers of producers and consumers
  - NarrativeX.raw gives the raw output of testX
  - NarrativeX.sorted gives output for each thread separately
- Raw output shows how execution of threads are interleaved
- Sorted output shows the sequence of actions by each thread
- The amount of working and blocking time associated with each item can affect how threads behave, and thus how their execution is interleaved
  - Change the settings and see how behavior changes, if it does
- Also, run a given test multiple times with the same setting and look for differences in behavior due to random chance and changing system conditions

# Lab Assignment

- Your final task for this lab is to experiment with producer_consumer using different numbers of threads and different work settings
- If you have modified the producer and consumer routines to use the mutex and condition variables correctly, the program should work correctly for arbitrary numbers of threads
- How will you ensure the producers (consumers) produce (consume) the correct number of items? Consider how the integer pointer to the number consumed and the number produced is used.

# Testing

- *producer_consumer* takes as its arguments the number of producer and consumer threads it will use.
- Test your program for different combinations of producers and consumers: several producers and 1 consumer, 1 producer and several consumers, several producers and consumers.
  - The testX makefile targets are a guide, but try other things as well
  - You will have to create your own raw and sorted files for new tests
- Examine the raw and sorted output for a given test and see what you can deduce about behavior

# Conclusions

- Producer-Consumer is an extremely simple canonical problem which arises in a wide range of situations
- Yet, as simple as it is, there are a number of interesting features and a wide range of behavior
- One important part of this assignment is to look for small inconsistencies or other features of a behavior narrative that indicate unexpected scenarios
- Another important aspect is to note how variable the behavior of different runs under the same settings can be
    - Concurrency is subject to a lot of random variation