Solving the Producer – Consumer Problem with PThreads

Michael Jantz

Dr. Prasad Kulkarni

Dr. Douglas Niehaus

Edited by: Ahmet Soyyigit

Introduction

- If you have not done the PThreads Intro lab, you should do that first before attempting to this one.
- 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:
  - sem_overview, sem_wait, sem_post

The Producer - Consumer Problem

- The producer – consumer is a problem for cooperating processes or threads which share a common buffer for communication. Basically, a **producer** produces data and puts it in a common buffer that is consumed by a **consumer**. There can be multiple producers and consumers sharing the same common buffer.
- The solution to producer-consumer problem should allow us:
  - To run producer and consumer threads concurrently in sync. Therefore, it resolves synchronization problems that arise from race conditions.
  - To block a consumer when it tries to consume data from an empty buffer. This consumer should be resumed when the buffer becomes non-empty.
  - To block a producer when it tries to put data to a buffer that is full. This producer is resumed when the buffer becomes non-full.
  - As a result, producer(s) pass data to consumer(s) via the common buffer.
- 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.

Lab file

- In producer_consumer.c, there is an instance of the Producer-Consumer problem.
- Producer and consumer threads are created in the main function. They share a bounded length FIFO queue whose type is **queue (**a struct type**)**.
- Each producer places an integer into the queue every iteration until WORK_MAX integers are placed in total. They *notify* consumers whenever an item is placed.
- Each consumer remove an integer from the queue every iteration until WORK_MAX iterations are removed in total. They *notify* producers whenever an item is removed.

The Proposed Solution

- In order to solve the 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. A check to see if the queue is empty shows that it is.
Add last items to the queue. Set queue->empty = 0 i == WORK_MAX, producer exits.	
	Set queue->empty = 1. Set queue->full = 0 Start spinning.

Imposing Mutual Exclusion

- The basic problem with this solution is that the critical section of the code where threads access 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).
- A solution: we can use a single *mutex* to protect everything. For producer, it would be like this:

```
while (1) {
    do_work(PRODUCER_CPU, PRODUCER_BLOCK);
    pthread_mutex_lock(mutex);
    while (buffer it full){
        pthread_mutex_unlock(mutex);
        sleep for some time before trying again
        pthread_mutex_lock(mutex);
    }
    produce and put item into the queue
    pthread_mutex_unlock(mutex);
}
```

Although this solution ensures mutual exclusion, the problem of spinning still remains. We need a more CPU friendly solution instead of busy waiting.

Busy Waiting Problem

- Let’s go back and think about the initial solution.
- Even if the thread is lucky enough to avoid busy waiting, 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’ after execution.
- 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

Solution

- The problem of spinning can be solved by using condition variables or semaphores. We will focus on the semaphore solution in this lab.
- Semaphores are simply unsigned integers which are increased and decreased atomically.
- A thread can decrease a semaphore with the sem_wait function, and increase it with the sem_post.
- When a thread wants to decrease a semaphore that is zero at the moment, the sem_wait function will cause this thread to be end up being blocked by the OS. This blocked thread will be put in a queue of blocked threads waiting for the same semaphore. Each semaphore has its own waiting queue.
- When some other thread increase the same semaphore with sem_post, the thread at the head of queue will become ready to run. When it starts running, it will decrease the semaphore and return from the sem_wait call.

Solution

- As semaphores can be any unsigned integer, we can use them to represent the number of empty slots and the number of items inside the queue.
- This way, we will achieve our two important goals:
  - Getting the status of the buffer atomically
  - Avoiding busy waiting if the condition for producing or consuming is not yet satisfied
- Apart from these, accessing to queue data also needs to be done atomically. This is why we still need the mutex.
- Therefore, we should protect shared data operations with mutex while avoiding busy waiting with semaphores.

Library Calls

- int sem_init(sem_t *sem, int pshared, unsigned int value);
  - Used to initialize a semaphore object. For our case, pshared will be 0 as we are not going to share the semaphore between processes. The value can be used to represent the amount of emptiness or fullness of a buffer.
- int sem_wait(sem_t *sem);
  - This function will decrease the semaphore if it is higher than zero, and the execution of the thread will continue. If the semaphore is zero, calling thread will be blocked until it is put to ready queue of the scheduler by the OS. Please look at the slide 9 for explanation.
- int sem_post(sem_t *sem);
  - This function will increase the semaphore and the OS will wake up a thread waiting for this semaphore, if there is any.

A side info: As you might have noticed, a mutex is a semaphore which can be either one or zero.

Modifying producer_consumer.c

- Modify producer_consumer.c to make use of the semaphore and mutex library calls to solve the problem.
- 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 waken up by the OS when that condition is satisfied.
- As a hint, the starter code has initialized all the mutex and semaphores you should need EXCEPT the initial value of the semaphores. One of your task is to determine these values.

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 produce the correct number of items(also same for consumers)?

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 several 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 a 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 behavior of different runs can vary under the same settings.
  - Concurrency is subject to a lot of random variation