ECE 252: Systems Programming and Concurrency Spring 2024

Lecture 15 — The Producer-Consumer Problem

Prepared by Jeff Zarnett, taught by Seyed Majid Zahedi

Classical Synchronization Problems: Producer-Consumer

Various operating systems textbooks provide a few “classical problems”: some scenarios that are phrased in real-world terms but meant to be an analogy for a problem that operating systems will deal with. These standard or classic problems are used to test any newly-proposed synchronization or coordination scheme. The solutions make use of semaphores as the basis for mutual exclusion. We are going to examine three of them in-depth: the producer-consumer problem, the readers-writers problem, and the dining philosophers problem.

The most common synchronization problem is the producer-consumer problem, also sometimes called the bounded-buffer-problem. Two processes share a common buffer that is of fixed size. One process is the producer: it generates data and puts it in the buffer. The other is the consumer: it takes data out of the buffer. This problem can be generalized to have p producers and c consumers, but for the sake of keeping the explanation simple, for now we will have just one of each [Tan08].

There are a couple of rules to be aware of. It is not possible to write into a buffer that is already full; if the buffer has capacity N and there are currently N items in it, the producer cannot write into the buffer and must wait until there is space. It is similarly not possible to read from an empty buffer; if the buffer has zero elements in it, the consumer cannot read from the buffer and must wait until there is something in there.

To keep track of the number of items in the buffer, we will have some variable `count`. This is a variable shared between more than one thread, and therefore access to this should be controlled with mutual exclusion. Let us assume the maximum number of elements in the buffer is defined as `BUFFER_SIZE`.

If busy-waiting is permitted, that is, we do not care if we are wasting CPU time, we can get away with one mutex, which we can call `mutex`. Each of the producer and consumer threads very likely run in an infinite loop on their own, but the code below is the sufficient to explain one iteration.

Producer

```
[produce item]
added = false
while added is false
   wait( mutex )
   if count < BUFFER_SIZE
      [add item to buffer]
      count++
      added = true
   end if
   post( mutex )
end while
```

Consumer

```
removed = false
while removed is false
   wait( mutex )
   if count > 0
      [remove item from buffer]
      count--
      removed = true
   end if
   post( mutex )
end while
[consume item]
```

While this accomplishes what we want, it is inefficient. Let’s add a new rule that says we want to avoid busy-waiting. Thus, when the producer is waiting for space it will be blocked and just as the consumer will be when the consumer is waiting for an element. To accomplish this, we will need two general semaphores, each with maximum value of `BUFFER_SIZE`. The first is called `items`: it starts at `0` and represents how many spaces in the buffer are full. The second is the mirror image `spaces`; it starts at `BUFFER_SIZE` and represents the number of spaces in the buffer that are currently empty.

Producer

```
1. [produce item]
2. wait( spaces )
3. [add item to buffer]
4. post( items )
```

Consumer

```
1. wait( items )
2. [remove item from buffer]
3. post( spaces )
4. [consume item]
```

The producer can continue to produce items until the buffer is full and the consumer can continue to consume items until the buffer is empty. This solution works okay, given two assumptions: (1) that the actions of adding an item to the buffer and removing an item from the buffer add to and remove from the "next" space; and (2) that there is exactly one producer and one consumer in the system. If we have two producers, for example, they might be trying to write into the same space at the same time, and this would be a problem.

To generalize this solution to allow multiple producers and multiple consumers, what we need to do is add another binary semaphore, `mutex` (initialized to 1), effectively combining the previous solution with the one before it:

Producer

```
1. [produce item]
2. wait( spaces )
3. wait( mutex )
4. [add item to buffer]
5. post( mutex )
6. post( items )
```

Consumer

```
1. wait( items )
2. wait( mutex )
3. [remove item from buffer]
4. post( mutex )
5. post( spaces )
6. [consume item]
```

This situation should be setting off some alarm bells in your mind. In the synchronization patterns examined earlier, we mentioned the possibility of deadlock: all threads getting stuck. The hint that we might have a problem is one `wait` statement inside another. Unfortunately, seeing this pattern is not necessarily a guarantee that deadlock is going to happen (that would be too easy). This is, however, a sign that we need to analyze the code to determine if there is a problem.

Reading through the psuedocode above, you should be able to reason that this solution will not get stuck. You may choose a strategy along the lines of "proof by contradiction" and try to come up with a scenario that leads to deadlock. If you are unable to find one, then you may have a suitable solution (though it might be best to have someone else check to be sure). This is not a substitute for a formal mathematical proof, but the logic in your analysis should be convincing. Consider an alternate solution:

Producer

```
1. [produce item]
2. wait( mutex )
3. wait( spaces )
4. [add item to buffer]
5. post( items )
6. post( mutex )
```

Consumer

```
1. wait( mutex )
2. wait( items )
3. [remove item from buffer]
4. post( spaces )
5. post( mutex )
6. [consume item]
```

This solution is very much like the one we are certain works, except we have swapped the order of the `wait` statements. As before, we need to analyze this code to determine if there is a problem. This solution does have the deadlock problem. Imagine at the start of execution, when the buffer is empty, the consumer thread runs first. It will wait on `mutex`, be allowed to proceed, and then will be blocked on `items` because the buffer is initially empty. The thread is blocked. When the producer thread runs, it waits on `mutex` and cannot proceed because the consumer thread is in the critical section there. So the producer is blocked and can never produce any items. Thus, we have deadlock. This situation could occur any time the buffer is empty.

If the above pseudocode were implemented it is not a certainty that there will be a deadlock every time. In fact, the code will probably work fine most of the time. Once, however, we have found one scenario that can lead to deadlock, there is no need to look for other failure cases; we can write off this solution and replace it with a better one.

But let's get to doing an actual example! We will take some time to analyze this solution and understand how we got from the psuedocode above to the actual code below.

Producer-Consumer Example

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <semaphore.h>

#define BUFFER_SIZE 20
sem_t spaces;
sem_t items;
int counter = 0;
int* buffer;

int produce() {
  ++counter;
  return counter;
}

void consume( int value ) {
  printf("Consumed %d.\n", value);
}

void* producer( void* arg ) {
  int pindex = 0;
  while( counter < 10000 ) {
    int v = produce();
    sem_wait( &spaces );
    buffer[pindex] = v;
    pindex = (pindex + 1) % BUFFER_SIZE;
    sem_post( &items );
  }
  pthread_exit( NULL );
}

void* consumer( void* arg ) {
  int cindex = 0;
  int ctotal = 0;
  while( ctotal < 10000 ) {
    sem_wait( &items );
    int temp = buffer[cindex];
    buffer[cindex] = -1;
    cindex = (cindex + 1) % BUFFER_SIZE;
    sem_post( &spaces );
    consume( temp );
    ++ctotal;
  }
  pthread_exit( NULL );
}

int main( int argc, char** argv ) {
  buffer = malloc( BUFFER_SIZE * sizeof( int ) );
  for ( int i = 0; i < BUFFER_SIZE; i++ ) {
    buffer[i] = -1;
  }
  sem_init( &spaces, 0, BUFFER_SIZE );
  sem_init( &items, 0, 0 );

  pthread_t prod;
  pthread_t con;

  pthread_create( &prod, NULL, producer, NULL );
  pthread_create( &con, NULL, consumer, NULL );
  pthread_join( prod, NULL );
  pthread_join( con, NULL );
```

```
  free( buffer );
  sem_destroy( &spaces );
  sem_destroy( &items );
  pthread_exit( 0 );
}
```

Parallelizing the Producer-Consumer Solution

Now suppose that we wanted to have ten producers and ten consumers. How do we get there from here?

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

#define BUFFER_SIZE 100
int buffer[BUFFER_SIZE];
int pindex = 0;
int cindex = 0;
sem_t spaces;
sem_t items;
pthread_mutex_t mutex;

int produce( int id ) {
  int r = rand();
  printf("Producer %d produced %d.\n", id, r);
  return r;
}

void consume( int id, int number ) {
  printf("Consumer %d consumed %d.\n", id, number);
}

void* producer( void* arg ) {
  int* id = (int*) arg;
  for(int i = 0; i < 10000; ++i) {
    int num = produce(*id);
    sem_wait( &spaces );
    pthread_mutex_lock( &mutex );
    buffer[pindex] = num;
    pindex = (pindex + 1) % BUFFER_SIZE;
    pthread_mutex_unlock( &mutex );
    sem_post( &items );
  }
  free( arg );
  pthread_exit( NULL );
}

void* consumer( void* arg ) {
  int* id = (int*) arg;
  for(int i = 0; i < 10000; ++i) {
    sem_wait( &items );
    pthread_mutex_lock( &mutex );
    int num = buffer[cindex];
    buffer[cindex] = -1;
    cindex = (cindex + 1) % BUFFER_SIZE;
    pthread_mutex_unlock( &mutex );
    sem_post( &spaces );
    consume( *id, num );
  }
  free( id );
  pthread_exit( NULL );
}

int main( int argc, char** argv ) {
  sem_init( &spaces, 0, BUFFER_SIZE );
  sem_init( &items, 0, 0 );
  pthread_mutex_init( &mutex, NULL );
```

```
  pthread_t threads[20];

  for( int i = 0; i < 10; i++ ) {
    int* id = malloc(sizeof(int));
    *id = i;
    pthread_create(&threads[i], NULL, producer, id);
  }
  for( int j = 10; j < 20; j++ ) {
    int* jd = malloc(sizeof(int));
    *jd = j-10;
    pthread_create(&threads[j], NULL, consumer, jd);
  }
  for( int k = 0; k < 20; k++ ){
    pthread_join(threads[k], NULL);
  }
  sem_destroy( &spaces );
  sem_destroy( &items );
  pthread_mutex_destroy( &mutex );
  pthread_exit( 0 );
}
```

References

[Tan08] Andrew S. Tanenbaum. *Modern Operating Systems, 3rd Edition*. Prentice Hall, 2008.