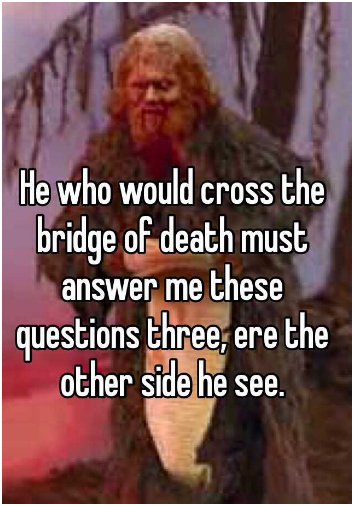
Lecture 28 — The Producer-Consumer Problem

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

April 7, 2024

Monty Python and the Holy Compiler



The producer-consumer problem, the readers-writers problem, and the dining philosophers problem.

Produce and Consume

First: 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.

Rules:

- The buffer is of capacity `BUFFER_SIZE`.
- Cannot write into a full buffer
- Cannot read from an empty buffer

To keep track of the number of items in the buffer, we will have some variable `count`.

This is a shared variable, so we need a mutex for it.

Are We There Yet?

If busy-waiting is permitted, we can get away with one mutex.

Shown below is one loop iteration for each of the producer & consumer.

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]
```

No Busy-Waiting

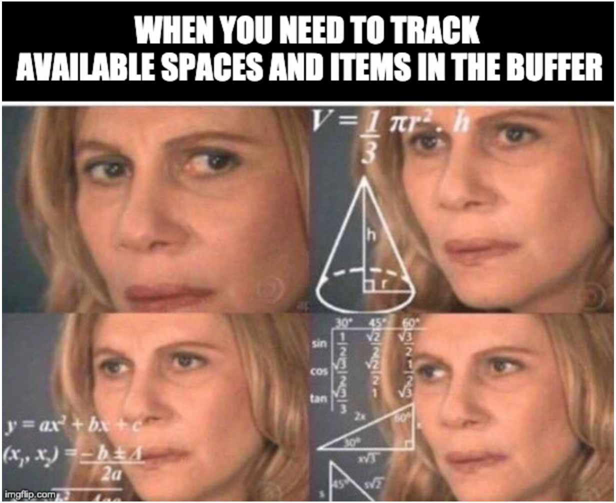
While this accomplishes what we want, it is inefficient.

Let’s add a new rule that says we want to avoid busy-waiting.

The producer gets blocked if there are no available spaces.

The consumer gets blocked if there’s nothing to consume.

When You Lose Track of the Number of Sets...



Use Semaphores To Count

Use 2 general semaphores, each with maximum value of `BUFFER_SIZE`.

`items`: starts at 0 and represents how many spaces in the buffer are full.

`spaces`: starts at `BUFFER_SIZE` and represents the number of spaces in the buffer that are currently empty.

Producer-Consumer with Waiting

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]
```

Does this work?

Are there any implicit assumptions?

Assumptions made? I assume so...

(1) The actions of adding an item to the buffer and removing an item from the buffer add to and remove from the “next” space.

(2) 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.

Mmmmmmmulti-Consume!

To generalize this solution to allow multiple producers and multiple consumers, we need a `mutex`.

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]
```

Does this work?

Anything... worrying?

Cancel Red Alert

The hint that we might have a problem is one `wait` statement inside another.

But it doesn't guarantee a problem...

We should be able to reason through why there is (or isn't) a problem.

Alternative Solution: PC

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]
```

Does this work?

This solution does have the deadlock problem!

Imagine at the start of execution, the buffer is empty and the consumer runs first...

Do you see the problem now?

This could also happen with the producer.

Problems are Only Sometimes a Problem

If this solution were implemented, it wouldn't guarantee a deadlock occurs.

In fact, it probably works 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 replace this solution with a better one.

P-C 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);
}
```

P-C Example

```
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 );
}
```

P-C Example

```c
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 );
}
```

Before We Go On

Mutex Syntax

We should take a moment to learn about the syntax of the pthread mutex.

While it is possible, of course, to use a semaphore as a mutex, frequently we will use the more specialized tool for this task.

In fact, it’s generally good practice to use the more specialized tool.

The structure representing the mutex is of type `pthread_mutex_t`.

```
pthread_mutex_init( pthread_mutex_t *mutex, pthread_mutexattr_t *attributes )
```

`mutex`: the mutex to intiialize.

`attributes`: the attributes; `NULL` is fine for defaults.

Shortcut if you do not want to set attributes:

```
pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;
```

By default, the mutex is created as unlocked.

Lock and Unlock

```
pthread_mutex_lock( pthread_mutex_t *mutex )
pthread_mutex_trylock( pthread_mutex_t *mutex ) /* Returns 0 on success */
pthread_mutex_unlock( pthread_mutex_t *mutex )
```

Unlock is self-explanatory.

`pthread_mutex_lock` is blocking.

`pthread_mutex_trylock` is nonblocking.

Trylock will come up again soon when we look at another classical synchronization problem.

Destroy the Mutex

```
pthread_mutex_destroy( pthread_mutex_t *mutex )
```

Destroy is also self-explanatory.

An attempt to destroy the mutex may fail if the mutex is currently locked.

Attempting to destroy a locked one results in undefined behaviour.

Multiple Producer-Consumer Example

```c
#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);
}
```

Multiple Producer-Consumer Example

```
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 );
}
```

Multiple Producer-Consumer Example

```
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 );
}
```