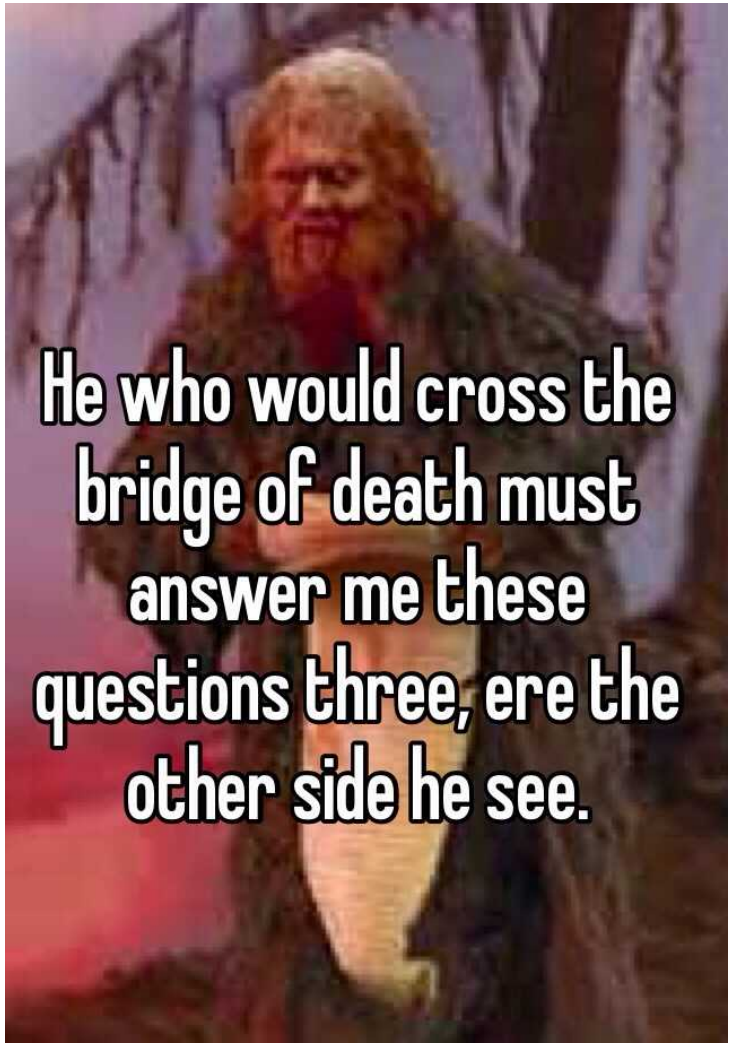
Lecture 15 — The Producer-Consumer Problem

By Jeff Zarnett and Seyed Majid Zahedi
jzarnett@uwaterloo.ca, smzahedi@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

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.

Consume, Obey

Rules:

- The buffer is of capacity `BUFF_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

```
 1. [produce item]
 2. added = false
 3. while added is false
 4.     lock( mutex )
 5.     if count < BUFF_SIZE
 6.         [add item to buffer]
 7.         count++
 8.         added = true
 9.     end if
10.     unlock( mutex )
11. end while
```

Consumer

```
 1. removed = false
 2. while removed is false
 3.     lock( mutex )
 4.     if count > 0
 5.         [remove item from buffer]
 6.         count--
 7.         removed = true
 8.     end if
 9.     unlock( mutex )
10. end while
11. [consume item]
```

No Busy-Waiting

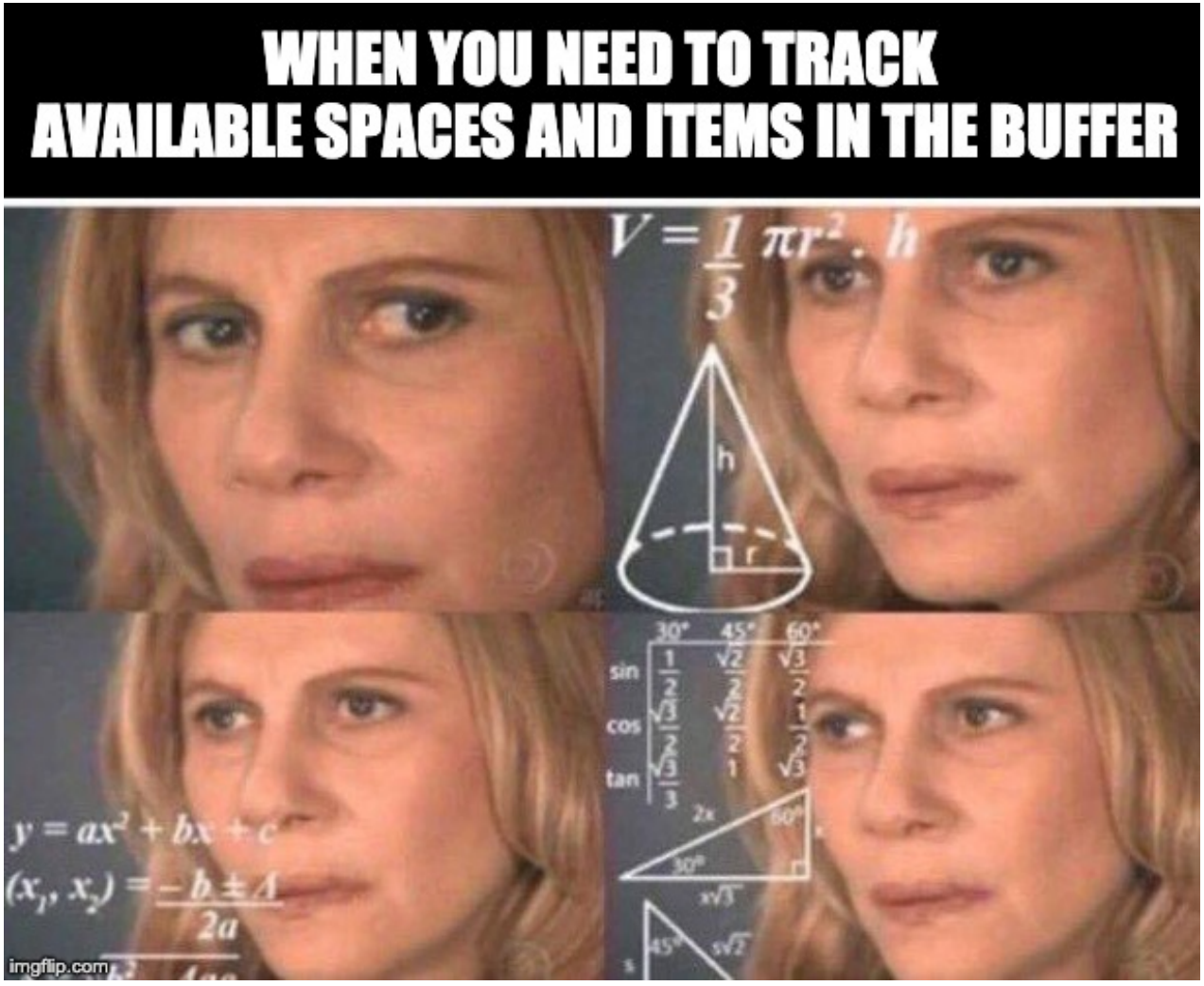
While this accomplishes what we want, it is inefficient.

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 `BUFF_SIZE`.

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

`spaces`: starts at `BUFF_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?

The Tiny Details...

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.

Multiple Producer-Consumer Example

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

#define BUFF_SIZE 100
int buffer[BUFF_SIZE];
int pindex = 0;
int cindex = 0;
sem_t spaces;
sem_t items;
sem_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 );
    sem_wait( &mutex );
    buffer[pindex] = num;
    pindex = (pindex + 1) % BUFF_SIZE;
    sem_post( &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 );
    sem_wait( &mutex );
    int num = buffer[cindex];
    buffer[cindex] = -1;
    cindex = (cindex + 1) % BUFF_SIZE;
    sem_post( &mutex );
    sem_post( &spaces );
    consume( *id, num );
  }
  free( id );
  pthread_exit( NULL );
}
```

Multiple Producer-Consumer Example

```
int main( int argc, char** argv ) {
  sem_init( &spaces, 0, BUFF_SIZE );
  sem_init( &items, 0, 0 );
  sem_init( &mutex, 0, 1 );

  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 );
  sem_destroy( &mutex );
  pthread_exit( 0 );
}
```

PC with Monitor

Use 1 mutex for mutual exclusion.

Use 2 CVs for waiting.

`full`: CV for producers to wait on when buffer is full.

`empty`: CV for consumers to wait on when buffer is empty.

Multi PC!

Producer

```
1. [produce item]
2. lock( mutex )
3. while count == BUFF_SIZE
4.    cond_wait( full, mutex )
5. end while
6. [add item to buffer]
7. count++
8. cond_signal( empty )
9. unlock( mutex )
```

Consumer

```
1. lock( mutex )
2. while count == 0
3.    cond_wait( empty, mutex )
4. end while
5. [remove item from buffer]
6. count--
7. cond_signal( full )
8. unlock( mutex )
9. [consume item]
```

Does this work?

Starvation: Where is My Ice Cream?

There is a subtle fairness issue, called starvation, which happens when a thread never gets a chance to run.

Consider a producer thread A that is waiting on the `full` CV.

Suppose that a consumer thread B arrives, consumes an item, and signals `full`.

At this point, another producer thread C arrives and goes to the ready queue.

The consumer thread B exits. The OS schedules C, which produces an item and makes the buffer full again.

The producer thread A runs next and has to wait again. This could continue infinitely, starving thread A.

Starvation: Where is My Ice Cream?

The probability of this happening is extremely low (close to zero!).

So, we might be fine with it and leave it as is.

However, if we want to make our code 100% starvation-free, our solution will involve a take-a-number approach.

Take A Number!

Take-a-number System

In this approach, before accessing the buffer, each incoming thread takes a number.

They then check the *now-serving* display to see if it's their turn. If not, they wait for their turn.

To implement this solution, we introduce `p_turn` and `c_turn` variables to track turns for producers and consumers, respectively.

We also introduce `next_p_turn` and `next_c_turn` variables to indicate the *now-serving* number for producers and consumers, respectively.

Solution 2

Producer

```
 1. [produce item]
 2. lock( mutex )
 3. my_turn = p_turn++
 4. while count == BUFF_SIZE ||
           next_p_turn < my_turn
 5.    cond_wait( full, mutex )
 6. end while
 7. [add item to buffer]
 8. count++
 9. next_p_turn++
10. cond_signal( empty )
11. unlock( mutex )
```

Consumer

```
 1. lock( mutex )
 2. my_turn = c_turn++
 3. while count == 0 ||
           next_c_turn < my_turn
 4.    cond_wait( empty, mutex )
 5. end while
 6. [remove item from buffer]
 7. count--
 8. next_c_turn++
 9. cond_signal( full )
10. unlock( mutex )
11. [consume item]
```

Does this work?

Analysis of Solution 2

Solution 2 is a step in the right direction, but it has a problem.

In particular, a signal could be delivered to a wrong thread—a thread whose turn has not yet come—who will wake up, check the conditions, and go to sleep again, wasting the signal.

When there is equal number of `cond_wait` and `cond_signal`, each wasted signal directly translates to a waiting thread that never wakes up!

IMMA GO SLEEP!

Solution 3

Producer

```
 1. [produce item]
 2. lock( mutex )
 3. my_turn = p_turn++
 4. while count == BUFF_SIZE ||
           next_p_turn < my_turn
 5.    cond_wait( full, mutex )
 6. end while
 7. [add item to buffer]
 8. count++
 9. next_p_turn++
10. cond_broadcast( empty )
11. unlock( mutex )
```

Consumer

```
 1. lock( mutex )
 2. my_turn = c_turn++
 3. while count == 0 ||
           next_c_turn < my_turn
 4.    cond_wait( empty, mutex )
 5. end while
 6. [remove item from buffer]
 7. count--
 8. next_c_turn++
 9. cond_broadcast( full )
10. unlock( mutex )
11. [consume item]
```

Does this work?

Analysis of Solution 3

This solution works.

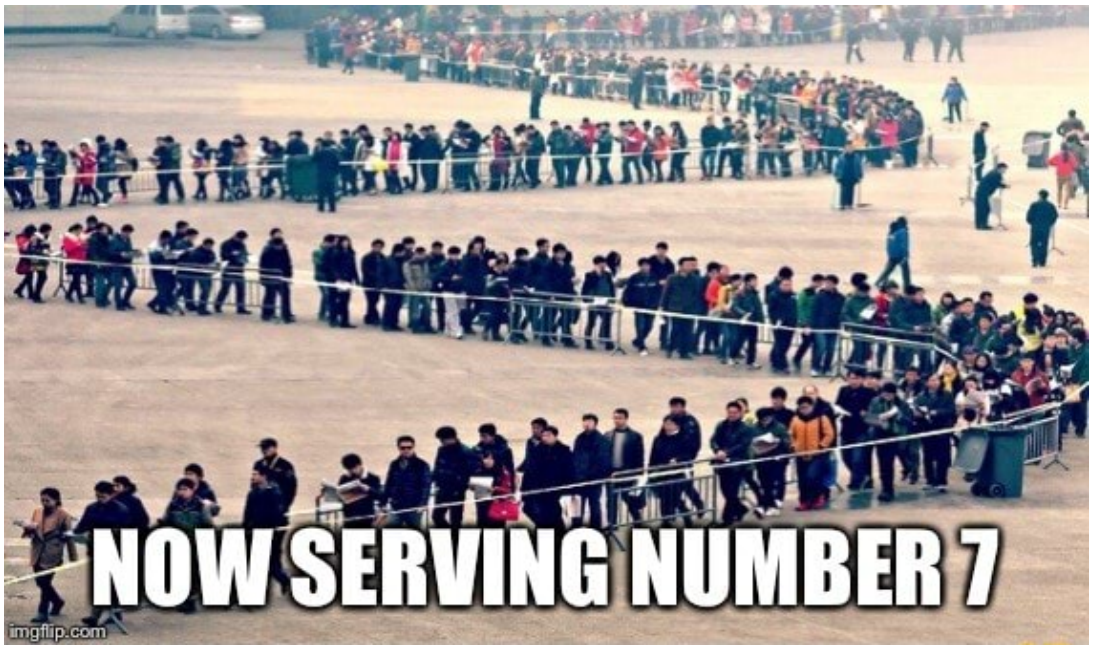
However, it is very inefficient in the sense that we wake everyone up, even though only one of them is allowed to proceed.

To properly solve the fairness issue and avoid inefficiency, threads need to send their signal to the right waiting thread.

We can achieve this by manually creating a FIFO list of waiting threads.

Each thread will have their own CV to wait on. Once a thread is done, they signal exactly the next thread in the line:

There Has To Be A Line!



Solution 4

Producer

```
 1. [produce item]
 2. lock( mutex )
 3. my_turn = p_turn++
 4. fifo_push( p_fifo, my_full )
 5. while count == BUFF_SIZE ||
          next_p_turn < my_turn
 6.    cond_wait( my_full, mutex )
 7. end while
 8. [add item to buffer]
 9. count++
10. next_p_turn++
11. fifo_pop( p_fifo )
12. if !fifo_is_empty( c_fifo )
13.    cond_signal(fifo_head(c_fifo))
14. end if
15. unlock( mutex )
```

Consumer

```
 1. lock( mutex )
 2. my_turn = c_turn++
 3. fifo_push( c_fifo, my_empty )
 4. while count == 0 ||
          next_c_turn < my_turn
 5.    cond_wait( my_empty, mutex )
 6. end while
 7. [remove item from buffer]
 8. count--
 9. next_c_turn++
10. fifo_pop( c_fifo )
11. if !fifo_is_empty( p_fifo )
12.    cond_signal(fifo_head(p_fifo))
13. end if
14. unlock( mutex )
15. [consume item]
```

Are we finally there?