Lecture 14 — Classical Synchronization Problems

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

April 9, 2017

Classical Problems

Various operating systems textbooks provide a few “classical problems”.

Scenarios phrased in real-world terms; an analogy for OS problems.

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:

- The producer-consumer problem
- The readers-writers problem
- The dining philosophers problem.

Producer-Consumer Problem

Most common: producer-consumer problem (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.

Producer-Consumer Rules

It is not possible to write into a buffer that is already full.

It is similarly not possible to read from an empty buffer.

Either situation can block a process.

Producer-Consumer Problem

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

This is a variable shared between more than one thread.

Therefore access to this should be controlled with mutual exclusion.

The maximum number of elements in the buffer is defined as `BUFFER_SIZE`.

Producer-Consumer Solution 1

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

Producer

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

Consumer

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

Producer-Consumer Solution 1 Analysis

While this accomplishes what we want, it is inefficient.

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

Thus, when the producer is waiting for space it will be blocked.

Same for the consumer when waiting for an element.

Producer-Consumer Solution 2

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 empty spaces.

Producer-Consumer Solution 2

Producer

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

Consumer

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

Producer-Consumer Solution 2 Analysis

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) 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.

General Producer-Consumer Solution

Now let us allow for multiple producers and multiple consumers.

Producer

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

Consumer

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

General Producer-Consumer Analysis

This situation should be setting off some alarm bells in your mind.

Recall the possibility of deadlock: all threads getting stuck.

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

Seeing this pattern is not a guarantee that deadlock is going to happen.

We must analyze the code to determine if there is a problem.

Checking for a Deadlock

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.
Mind you, someone else may be able to...

This is not a substitute for a formal mathematical proof, but the logic in your analysis should be convincing.

General Producer-Consumer Solution 2

Consider this alternate solution:

Producer

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

Consumer

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

General Producer-Consumer Solution 2 Analysis

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.

Try to think of a scenario where that happens.

General Producer-Consumer Solution 2 Analysis

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.

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. Deadlock!

This situation could occur any time the buffer is empty.

If the above code 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.

If 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.

The Readers-Writers Problem

Concurrent reading & modification of a data structure or record by > 1 thread.

A writer will modify the data; a reader will read it only without modification.

Unlike the producer-consumer problem, some concurrency is allowed:

1. Any number of readers may be in the critical section simultaneously.
2. Only one writer may be in the critical section (and when it is, no readers)

The Readers-Writers Problem

A writer cannot enter the critical section while any other thread is there.

While a writer is in the critical section, neither readers nor writers may enter.

This is very often how file systems work.

If any thread could read or write the shared data structure, we would have to use the general mutual exclusion solution.

The general mutual exclusion routine would prevent errors, but is a serious performance reduction.

Readers-Writers Solution 1

Let us keep track of the number of readers at any given time with `readers`.

We will need a way of protecting this variable from concurrent modifications, so there will have to be a binary semaphore `mutex`.

We will also need one further semaphore, `roomEmpty`.

A writer has to wait for the room to be empty (i.e., `wait` on the `roomEmpty` semaphore) before it can enter.

Readers-Writers Solution 1

Writer

```
1. wait( roomEmpty )
2. [write data]
3. signal( roomEmpty )
```

Reader

```
 1. wait( mutex )
 2. readers++
 3. if readers == 1
 4.     wait( roomEmpty )
 5. end if
 6. signal( mutex )
 7. [read data]
 8. wait( mutex )
 9. readers--
10. if readers == 0
11.     signal( roomEmpty )
12. end if
13. signal( mutex )
```

Readers-Writers Solution 1 Analysis

The first reader that arrives encounters the situation that the room is empty, so it "locks" the room (waiting on the `roomEmpty` semaphore).

That will prevent writers from entering the room.

Additional readers do not check if the room is empty; they just proceed to enter.

When the last reader leaves the room, it signals that the room is empty ("unlocking it" to allow a writer in).

This pattern is sometimes called the light switch.

Readers-Writers Solution 1 Analysis

The reader code has that situation that makes us concerned.

A wait on `roomEmpty` inside a critical section controlled by `mutex`.

With a bit of reasoning, we can convince ourselves that there is no deadlock.

A reader waits on `roomEmpty` only if a writer is currently in its critical section.

As long as the write operation takes finite time, eventually the writer will signal the `roomEmpty` semaphore and the threads can continue.

Deadlock is not a problem.

Readers-Writers Solution 1 Analysis

There is, however, a second problem that we need to be concerned about.

Suppose some readers are in the room, and a writer arrives.

The writer must wait until all the readers have left the room.

When each of the readers is finished, it exits the room.

In the meantime, more readers arrive and enter the room.

Readers-Writers Solution 1 Analysis

So even though each reader is in the room for only a finite amount of time, there is never a moment when the room has no readers in it.

This undesirable situation is not deadlock, because the reader threads are not stuck, but the writer (and any subsequent writers) is (are) going to wait forever.

This is a situation called starvation: a thread never gets a chance to run.

Starvation

Recall criterion 3 of properties we want in any mutual exclusion solution:

It must not be possible for a thread to be delayed indefinitely.

This problem is just as bad as deadlock in that if it is discovered, it eliminates a proposed solution as an acceptable option.

Even though starvation might only be an unlikely event.

We must therefore improve on this solution such that there is no longer the possibility that a writer starves.

Readers-Writers Solution 2

When a writer arrives, any readers should be permitted to finish their read.
No new readers should be allowed to start reading.

Eventually, all the readers currently in the critical section will finish.

The writer will get a turn, because the room is empty.

When the writer is done, all the readers that arrived after the writer will be able to enter.

Readers-Writers Solution 2

Writer

```
1. wait( turnstile )
2. wait( roomEmpty )
3. [write data]
4. signal( turnstile )
5. signal( roomEmpty )
```

Reader

```
 1. wait( turnstile )
 2. signal( turnstile )
 3. wait( mutex )
 4. readers++
 5. if readers == 1
 6.     wait( roomEmpty )
 7. end if
 8. signal( mutex )
 9. [read data]
10. wait( mutex )
11. readers--
12. if readers == 0
13.     signal( roomEmpty )
14. end if
15. signal( mutex )
```

Readers-Writers Solution 2 Analysis

Does this solution satisfy our goals of avoidance of deadlock and starvation?

Starvation is fairly easy to assess: the first attempt at the solution had one scenario leading to starvation and this solution addresses it.

You should be able to convince yourself that the solution as described cannot starve the writers or readers.

Readers-Writers Solution 2 Analysis

On to deadlock: the reader code is minimally changed from before

The writer has that dangerous pattern: two waits.

If the writer is blocked on the `roomEmpty` semaphore, no readers or writers could advance past the turnstile and no writers.

If the writer is blocked on that semaphore, there are readers in the room.

The readers will individually finish and leave (their progress is not impeded).

So the room will eventually become empty; the writer will be unblocked.

Readers-Writers Solution 2 Analysis

Note that this solution does not give writers any particular priority: when a writer exits it signals `turnstile` and that may unblock a reader or a writer.

If it unblocks a reader, a whole bunch of readers may enter before the next writer is unblocked and locks the turnstile again.

That may or may not be desirable, depending on the application.

In any event, it does mean it is possible for readers to proceed even if a writer is queued.

If there is a need to give writers priority, there are techniques for doing so.
But we will not examine them here.

The Dining Philosophers Problem

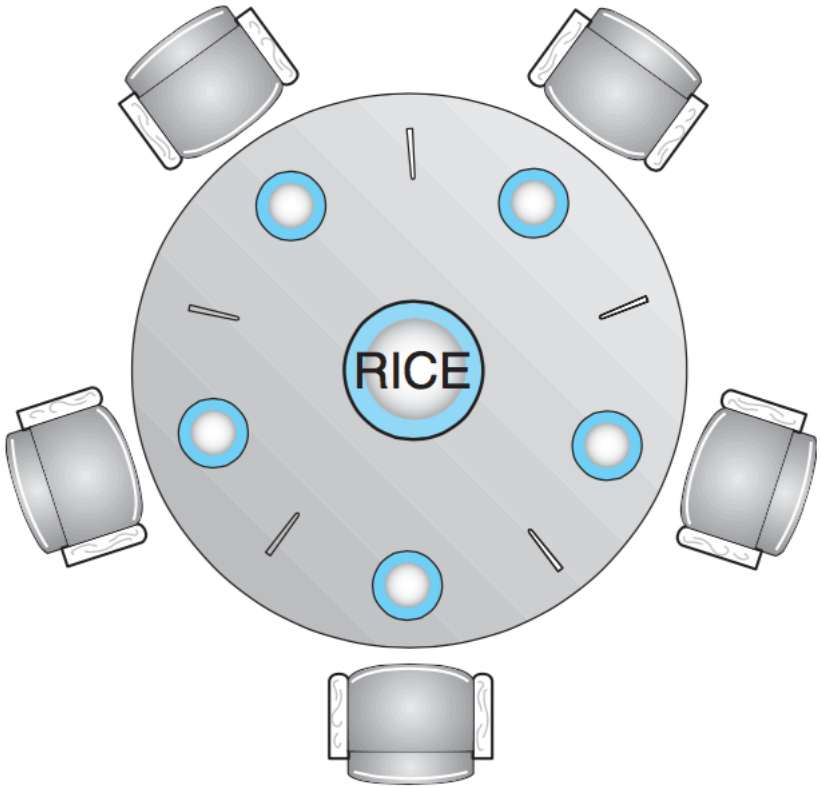
The dining philosophers problem was also proposed by Dijkstra in 1965.

The problem can have *n* philosophers, but problem is typically described as 5.

These five smart individuals spend their lives thinking, but every so often, they need to eat. They share a table, each having his or her own chair.

In the centre of the table is a bowl of rice, & it is laid with 5 single chopsticks.

The Dining Philosophers Problem



Dinner Time

When a philosopher wishes to eat, she sits down at her designated chair, and attempts to pick up the two chopsticks that are nearest.

Philosophers are polite and therefore do not grab a chopstick out of the hands of a colleague.

When a philosopher has both chopsticks, she may eat rice, and when she is finished, she puts down the chopsticks and goes back to thinking.

Dining with Semaphores

Suppose then that semaphores are the method for managing things.

Because only one person can be in possession of a chopstick at a time, each chopstick may be represented by a binary semaphore.

When the philosopher sits down he attempts to acquire the left chopstick, then the right, eats, and puts the chopsticks down.

This works fine, until all philosophers sit down at the same time. Each grabs the chopstick to his or her left.

None of them are able to acquire the chopstick to his or her right (because someone has already picked it up).

None of the philosophers can eat; they are all stuck. Deadlock.

Deadlock at the Dinner Table

This example makes it more clear why we call a situation where a thread never gets to run “starvation”.

If a philosopher is never able to get both chopsticks, that philosopher will never be able to eat.

Though I am not an expert on biology, I have it on good authority that people who do not eat anything end up eventually starving to death.

Even philosophers.

Saving the Philosophers

One thing that would guarantee that this problem does not occur is to protect the table with a binary semaphore.

This would allow exactly one philosopher at a time to eat, but at the very least, deadlock and starvation would be avoided.

Although this works, it is a suboptimal solution.

There are five seats and five chopsticks. Yet only one person is eating at a time.

No, the Pigeons are not for Eating

What if we limit the number of philosophers at the table concurrently to four?

The pigeonhole principle applies here: if there are *k* pigeonholes and more than *k* pigeons, at least one pigeonhole must have at least two pigeons.

Thus, at least one of the four philosophers can get two chopsticks.

Implementation is easy: a general semaphore with a max and initial value of 4.

Invite a Leftie

The problem above occurs because every philosopher tries to pick up the left chopstick first.

If some of them try to pick up the left and some pick up the right first, then deadlock will not happen.

This problem is a great basis to launch into a discussion about deadlock...