Lecture 14 — Synchronization Patterns

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

October 6, 2018

Synchronization Patterns

There are a number of common synchronization patterns that occur frequently and we can use semaphores to solve them.

These synchronization patterns are ways of co-ordinating threads or processes.

We have already examined serialization and mutual exclusion; there are more.

Throughout this section we will use pseudocode and something like “Statement A1” could be any valid statement in the program.

Signalling

Recall from earlier the example with Alice and Bob at the power plant.

This was postling.

Signalling can be used in general as a way of indicating that something has happened.

Suppose we have a semaphore named `sem`, initialized to 0.

Thread A

```
1. Statement A1
2. post( sem )
```

Thread B

```
1. wait( sem )
2. Statement B2
```

Analysis of Signalling

If *B* gets to the `wait` statement first, it will be blocked (as the semaphore is 0) and cannot proceed until someone posts on that semaphore.

When *A* does call `post`, then *B* may proceed.

If instead *A* gets to the `post` statement first, it will post and the semaphore value will be 1.

Then, when *B* gets to the `wait` statement, it can proceed without delay.

Regardless of the actual order that the threads run, we are certain that statement A1 will execute before statement B2.

Rendezvous

The rendezvous is an expansion of the post pattern so that it works both ways.

Two threads should be at the same point before either of them may proceed (they “meet up”).

Suppose we have:

Thread A

```
1. Statement A1
2. Statement A2
```

Thread B

```
1. Statement B1
2. Statement B2
```

Rendezvous

The desirable property is that A1 should take place before B2 and that B1 should take place before A2.

As each thread must wait for the other, two semaphores will be needed: one to indicate that *A* has arrived and one for *B*.

We will assign them the names `aArrived` and `bArrived` and initialize both to 0.

Rendezvous Solution 1

A first attempt at a solution:

Thread A

```
1. Statement A1
2. wait( bArrived )
3. post( aArrived )
4. Statement A2
```

Thread B

```
1. Statement B1
2. wait( aArrived )
3. post( bArrived )
4. Statement B2
```

Rendezvous Solution 1 Analysis

The problem here should be obvious: thread *A* gets to the `wait` statement and will wait until *B* posts its arrival before it can proceed.

Thread *B* gets to its `wait` statement and will wait until *A* posts its arrival before it will proceed.

Unfortunately, each thread is waiting for the other to post and neither of them can get to the actual `post` statement because they are both blocked.

Deadlock

Neither thread can proceed.

The situation can never be resolved, because there is no external force that would cause one or the other to be unblocked.

This is a situation called deadlock, and it is a subject that will receive a great deal of examination later on.

For now, an informal definition is: all threads are permanently stuck.

Obviously, this is undesirable.

Rendezvous Solution 2

What if instead, the threads reverse the order and post first before waiting?

Thread A

```
1. Statement A1
2. post( aArrived )
3. wait( bArrived )
4. Statement A2
```

Thread B

```
1. Statement B1
2. post( bArrived )
3. wait( aArrived )
4. Statement B2
```

Rendezvous Solution 2 Analysis

This solution works: if *A* gets to the rendezvous point first, it posts its arrival and waits for *B*.

If *B* gets there first, it posts its arrival and waits for *A*.

Whichever gets there last will post and unblock the other, before it calls `wait`.

It will be able to proceed directly; the first thread to arrive already postled.

Rendezvous Solution 3

A variation on this can also work where only one thread posts first and the other thread posts second.

Thread A

```
1. Statement A1
2. wait( bArrived )
3. post( aArrived )
4. Statement A2
```

Thread B

```
1. Statement B1
2. post( bArrived )
3. wait( aArrived )
4. Statement B2
```

Rendezvous Solution 3 Analysis

While this solution will not result in deadlock, it is somewhat less efficient than the previous: it may require an extra switch between processes.

As long as we are certain that deadlock will not occur, a solution is acceptable.

Nevertheless, the previous solution is provably better.

Mutual Exclusion

We saw previously the motivation and concept of mutual exclusion through messages in the linked list example.

The general form in pseudocode is of course:

Thread A

```
1. wait( mutex )
2. critical section
3. post( mutex )
```

Thread B

```
1. wait( mutex )
2. critical section
3. post( mutex )
```

The mutex semaphore is originally initialized to 1.

Whichever thread gets to the `wait` statement first will proceed immediately and not be blocked at all.

If the semaphore were initialized to 0 then neither thread could ever get to the `post` statement or ever get into the critical section (deadlock).

Symmetric vs. Asymmetric Solutions

Threads *A* and *B* are identical here.

This is a symmetric solution.

It is easier to make predictions about the behaviour of the threads when they all do the same thing.

If the different threads have different sections of code, they are asymmetric.

The symmetric solutions very often scale well.

Multiplex

In addition to the binary semaphore, we also discussed the general semaphore.

If the general semaphore is initialized to *n*, then at most *n* threads can be in the critical section at a time.

Example: restaurants have a certain number of tables and seats.

If more people wish to dine than there are seats available, those customers must wait until some seats become available (other customers leave).

Restaurants generally want to pack in as many seats as possible, but fire safety regulations set a maximum occupancy for a given space.

Multiplex

Suppose that the system has a problem that when too many concurrent database requests are happening.

The queries become slow and eventually time out.

A potential solution is to protect all database accesses with a binary semaphore, so only one database query can run at any time.

Analysis may reveal that this is too restrictive a policy; perhaps we can execute 5 queries concurrently without any slowdown.

Then initialize the semaphore with a value of 5, allowing at most 5 threads into the critical section at any time.

Multiplex

This is a symmetric solution, so it will work for arbitrarily many threads.

Thread *K*

```
1. wait( mutex )
2. critical section
3. post( mutex )
```

This looks exactly like the solution for mutual exclusion, as it should.

The only difference is how many threads can enter the critical section at a time.

Barrier

The barrier pattern is a generalization of the rendezvous pattern;
A way of having more than two threads meet up at the same point.

Given n threads, each of which knows that the total number of threads is n.

When the first $n - 1$ threads arrive, they should wait until the nth arrives.

As a solution we might consider a variable to keep track of the number of threads that have reached the appropriate point.

This variable is shared data; modification of it should be in a critical section.

Barrier

Thus we will have a semaphore, initialized to 1, called `mutex` to protect that counter.

Then we will have a second semaphore, `barrier` that will be the place where threads wait until the *n*th thread arrives.

Barrier Solution 1

Thread *K*

```
1. wait( mutex )
2. count++
3. post( mutex )
4. if count == n
5.     post( barrier )
6. end if
7. wait( barrier )
```

Barrier Solution 1 Analysis

When the *n*th thread arrives, it unlocks the barrier and then may proceed.

If there is more than one thread waiting at the barrier, the first thread will be unblocked when the *n*th thread posts on it.

There are no other `post` statements!

The other threads waiting are stuck, waiting for a post that never comes.

Barrier Solution 2

Barrier Solution 2

The *n*th thread to arrive should post $n - 1$ times:

Thread *K*

```
1. wait( mutex )
2. count++
3. post( mutex )
4. if count == n
5.     for i from 1 to  n
6.         post( barrier )
7.     end for
8. end if
9. wait( barrier )
```

Barrier Solution 2 Analysis

This allows all *n* threads to proceed (none get stuck), but it is less than ideal.

The thread that runs last is very likely the lowest priority thread.

When it posts on the semaphore, the thread that has just been unblocked will be the next to run.

Then the system switches back, at some later time, to the thread currently unblocking all the others.

Worst case, 2*n* process switches, when it could be accomplished with *n*.

Barrier Solution 3

Have each thread unblock the next:

Thread *K*

```
1. wait( mutex )
2. count++
3. post( mutex )
4. if count == n
5.     post( barrier )
6. end if
7. wait( barrier )
8. post( barrier )
```

Barrier Solution 3 Analysis: The Turnstile

This pattern (wait followed immediately by post) is called a turnstile.

The analogy should be familiar to anyone who has travelled by subway.
A turnstile allows one person at a time to go through.

A turnstile pattern allows one thread at a time to proceed through, but can be locked to bar all threads from proceeding.

Initially the turnstile in the above example is locked, and the *n*th thread unlocks it and permits all *n* threads to go through.

Barrier Solution 3 Analysis

In this solution we are reading the value of `count`, a shared variable, without the protection of a semaphore.

Is this dangerous?

Yes, but the alternative is, in this specific instance, worse.

Consider this instead:

Thread *K*

```
1. wait( mutex )
2. count++
3. if count == n
4.     post( barrier )
5. end if
6. wait( barrier )
7. post( barrier )
8. post( mutex )
```

Barrier Solution 3A Analysis

The problem here is deadlock once again.

The first thread waits on `mutex` and then goes to wait on the `barrier` semaphore.

At this point, the first thread is blocked.

When a second thread comes along, it will wait on `mutex` but can get no further because the first thread has not postled on it.

Barrier Solution 3A Analysis

The counter will be 1, but cannot get past 1.

The condition of `count` equalling *n* can never be true.

Thus, all the threads are stuck.

This is a common source of deadlock: blocking on a semaphore while inside a critical region.

Reusable Barrier

The barrier solution we have is good.

The way it is implemented now, `count` can increase but never decrease.

Once the barrier is open, it can never be closed again.

Programs very often do the same thing repeatedly, so a one-time use barrier is not ideal; it would be better to have a reusable barrier.

Idea: Decrement `count` after the rendezvous has taken place.

Reusable Barrier Solution 1

Thread *K*

```
 1. wait( mutex )
 2. count++
 3. post( mutex )
 4. if count == n
 5.     post( turnstile )
 6. end if
 7. wait( turnstile )
 8. post( turnstile )
 9. [critical point]
10. wait( mutex )
11. count--
12. post( mutex )
13. if count == 0
14.     wait( turnstile )
15. end if
```

Reusable Barrier Solution 1 Analysis

There are two problems with the above implementation.

Suppose thread $n - 1$ is about to execute line 4 and then there is a process switch and the nth thread comes to this point.

Both of them will find that `count` is equal to n and therefore both threads will post the turnstile.

The same problem occurs on line 13.

Reusable Barrier Solution 2

Thread *K*

```
 1. wait( mutex )
 2. count++
 3. if count == n
 4.     post( turnstile )
 5. end if
 6. post( mutex )
 7. wait( turnstile )
 8. post( turnstile )
 9. [critical point]
10. wait( mutex )
11. count--
12. if count == 0
13.     wait( turnstile )
14. end if
15. post( mutex )
```

Reusable Barrier Solution 2 Analysis

This solves the problem previously identified by putting the checks of `count` inside the critical section controlled by `mutex`.

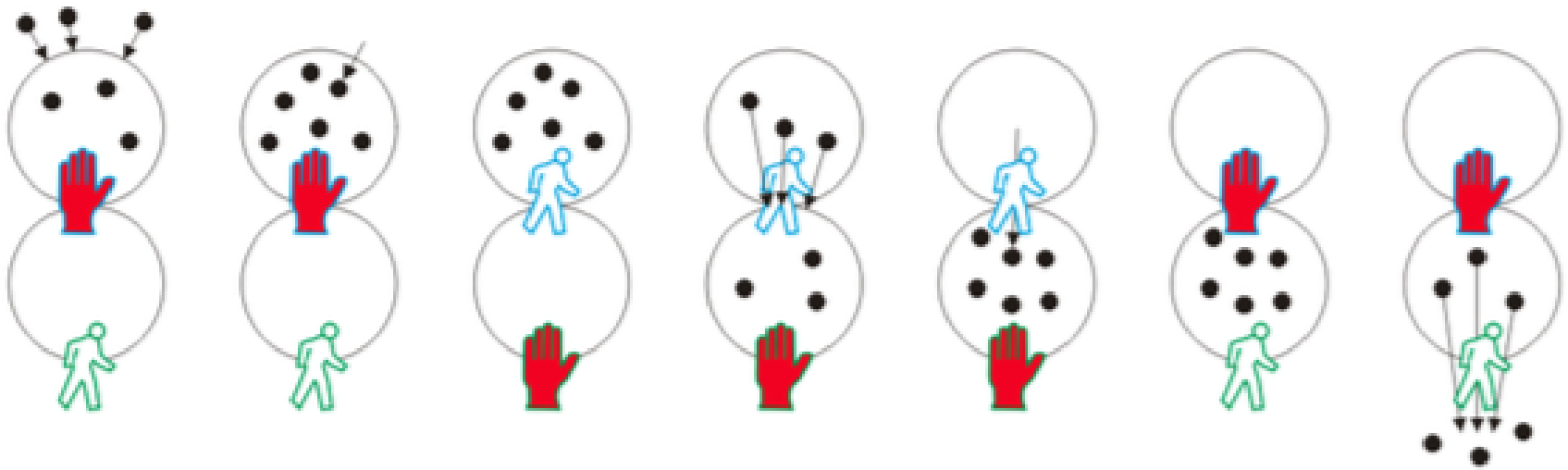
Suppose one particular thread gets through the second mutex but is running in a loop and gets back through the first mutex again.

This would be like one thread being one “lap” ahead of the others.

We can prevent this by having two turnstiles: first all threads wait at the first turnstile until the last gets there and lets them through.

Then all threads wait at a second turnstile until the last gets there and lets them all through again.

Reusable Barrier Visually



This solution can also be called a two-phase barrier because all threads have to wait twice: once at each turnstile.

Reusable Barrier Solution 3

Thread *K*

```
 1. wait( mutex )
 2. count++
 3. if count == n
 4.     wait( turnstile2 )
 5.     post( turnstile1 )
 6. end if
 7. post( mutex )
 8. wait( turnstile1 )
 9. post( turnstile1 )
10. [critical point]
11. wait( mutex )
12. count--
13. if count == 0
14.     wait( turnstile1 )
15.     post( turnstile2 )
16. end if
17. post( mutex )
18. wait( turnstile2 )
19. post( turnstile2 )
```