# Lecture 13 — Synchronization Patterns

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

April 27, 2015

# 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 signalling.

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. signal( 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 signals on that semaphore.

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

If instead *A* gets to the `signal` statement first, it will signal 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 signal 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. signal( aArrived )
4. Statement A2
```

**Thread B**

```
1. Statement B1
2. wait( aArrived )
3. signal( 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* signals its arrival before it can proceed.

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

Unfortunately, each thread is waiting for the other to signal and neither of them can get to the actual `signal` 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 signal first before waiting?

**Thread A**

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

**Thread B**

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

# Rendezvous Solution 2 Analysis

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

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

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

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

# Rendezvous Solution 3

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

**Thread A**

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

**Thread B**

```
1. Statement B1
2. signal( 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. signal( mutex )
```

**Thread B**

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