MASSEY UNIVERSITY
TE KUNENGA KI PŪREHUROA
UNIVERSITY OF NEW ZEALAND

159.341 Programming Languages, Algorithms & Concurrency

Synchronisation (Part 1)

Daniel Playne
d.p.playne@massey.ac.nz

Synchronisation

Reminder:
A process is **independent** if it is not affected by the execution of another process.

A **cooperating** process can affect (or be affected by) the execution of another process. When two processes share data with each other, they are cooperating.

Synchronisation

Processes and threads may cooperate/communicate through access to shared data or resources.

However, concurrent access to shared data may result in inconsistencies.

Synchronisation is necessary to ensure that shared resources are accessed in an appropriate way and that data consistency is preserved.

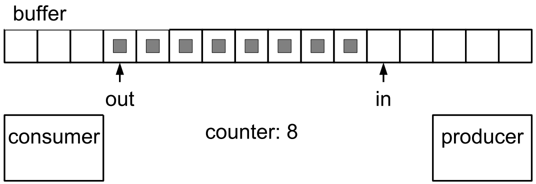
Synchronisation

Consider an example producer-consumer program that uses a buffer to communicate.

This solution will have a shared `buffer` of size `N`. The producer can place at most `N` items in the buffer.

The number of items currently in the buffer will be tracked with a shared `counter` which will be used to ensure the producer doesn't overfill the buffer and that the consumer never reads from an empty buffer.

Synchronisation



producer:

```
while(true) {
    // produce an item (next_item)
    while(counter == N) {}
    buffer[in] = next_item;
    in = (in + 1) % N;
    counter++;
}
```

consumer:

```
while(true) {
    while(counter == 0) {}
    next_item = buffer[out];
    out = (out + 1) % N;
    counter--;
    //consume item
}
```

What is wrong with this code?

Race Condition

This code has a **race condition** in it, specifically with the two instructions `counter++;` and `counter--;`.

If the value of `counter` is say 10 and both the producer and consumer concurrently execute these instructions, there will be three possibilities.

`counter` will have a value of 9, 10 or 11 (although clearly the correct value is 10).

Race Condition

The problem is that the instructions `counter--` and `counter++` are actually implemented (on a typical machine) as the following:

`counter++`

```
register1 = counter
register1 = register1 + 1
counter = register1
```

`counter--`

```
register2 = counter
register2 = register2 - 1
counter = register2
```

Race Condition

When these instructions execute concurrently, we can think of them following some sequential order with the instructions interleaved together (the order of each set of instructions is kept intact).

```
register1 = counter          {register1 = 10}
register2 = counter          {register2 = 10}
register1 = register1+1      {register1 = 11}
register2 = register2-1      {register2 = 9}
counter   = register1        {counter   = 11}
counter   = register2        {counter   = 9}
```

Race Condition

Another (just as possible) order:

```
register1 = counter        {register1 = 10}
register2 = counter        {register2 = 10}
register2 = register2-1    {register2 = 9}
counter   = register2      {counter   = 9}
register1 = register1+1    {register1 = 11}
counter   = register1      {counter   = 11}
```

Race Condition

Or if we are lucky:

```
register1 = counter          {register1 = 10}
register1 = register1+1      {register1 = 11}
counter   = register1        {counter   = 11}
register2 = counter          {register2 = 11}
register2 = register2-1      {register2 = 10}
counter   = register2        {counter   = 10}
```

Race Condition

Race conditions such as this one occur commonly in multithreaded programs and operating systems when multiple processes attempt to access the same shared resource.

Sections of code where threads or processes access some (potentially) shared resource or data are called **critical sections**. The programmer must identify these critical sections but once they have been identified there must be some way of protecting them against race conditions.

Critical-Section Problem

The **critical-section problem** involves designing a protocol such that when multiple processes that share access to some resource can never execute their critical sections at the same time.

This involves each process performing some actions to request permission to enter the critical section and some actions to signal the other processes when they have left the critical section.

Critical-Section Problem

A solution to the critical-section problem must satisfy the following requirements:

- **Mutual exclusion** - no two processes may be executing their critical sections at the same time.
- **Progress** - if no process is executing its critical section and some processes wish to enter their critical sections, only those processes participate in deciding which process should enter.
- **Bounded waiting** - there is an upper limit on the number of times other processes may enter their critical sections after a process has made a request to enter and before that request is granted.

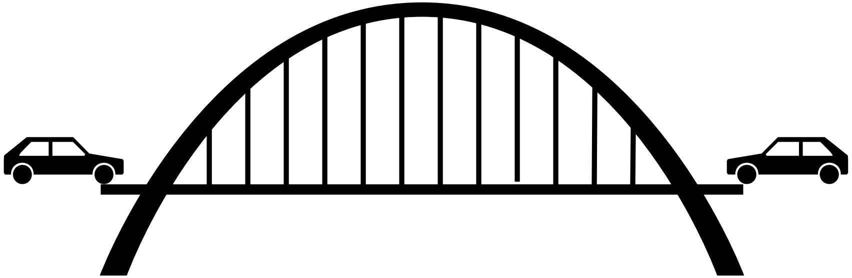
Critical-Section Problem

Consider the following thought experiment:

- One lane bridge covered in fog (can't see what is on the bridge)
- Two cars both want to cross the bridge (no reverse gear)
- Cannot tell how long a car will spend crossing the bridge
- Only ever one car going L or R at a time.
- Can set up lights on the bridge that can be seen through the fog.
- Only atomic operations are allowed.

Critical-Section Problem

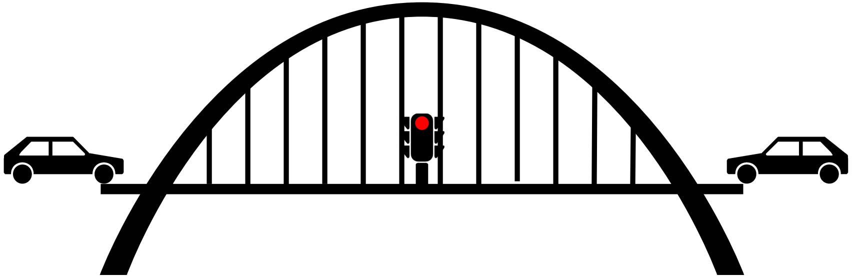
Can we design a protocol that allows the cars to cross the bridge safely and meet the requirements for the critical-section problem?



Bridge crossing.

Critical-Section Problem - Solution 1

Solution 1 - use a light indicating which car's turn it is next. When the light is green, the car on the left crosses, when it is red the car on the right crosses. Each car switches the light after they leave the bridge.


Solution 1 - Strict Alternation

Critical-Section Problem - Solution 1

Solution 1 - Strict Alternation

Left car:

```
while(true) {
    while(light != GREEN) {
        // wait
    }
    // cross bridge
    light = RED;
    // other tasks
}
```

Right car:

```
while(true) {
    while(light != RED) {
        // wait
    }
    // cross bridge
    light = GREEN;
    // other tasks
}
```

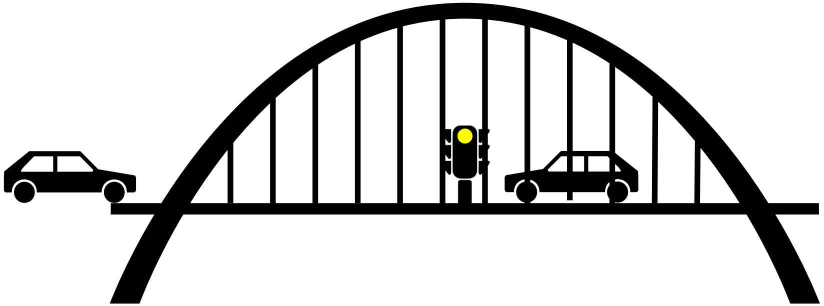
Critical-Section Problem - Solution 1

Solution 1 - Strict Alternation

- **Mutual exclusion** - yes, both cars cannot cross at the same time.
- **Bounded waiting** - yes, each car will have to wait for at most one other car.
- **Progress** - no, a car may have to wait even if there is no other car waiting.

Critical-Section Problem - Solution 2

Solution 2 - use the light to signal that a car is on the bridge. Before a car tries to cross the bridge it checks whether the light is on, if not it will turn it on and cross the bridge (remembering to turn it off after it's crossed).



Solution 2

Critical-Section Problem - Solution 2

Solution 2

Left car:

```
while(true) {
    while(light == ON) {
        // wait
    }
    light = ON;
    // cross bridge
    light = OFF;
    // other tasks
}
```

Right car:

```
while(true) {
    while(light == ON) {
        // wait
    }
    light = ON;
    // cross bridge
    light = OFF;
    // other tasks
}
```

Critical-Section Problem - Solution 2

What happens if the cars both arrive at the same time?

1. Left car arrives and checks the light (OFF)
2. Right car arrives and checks the light (OFF)
3. Left car sets light to ON
4. Right car sets light to ON
5. Both cars enter the bridge and **crash!**

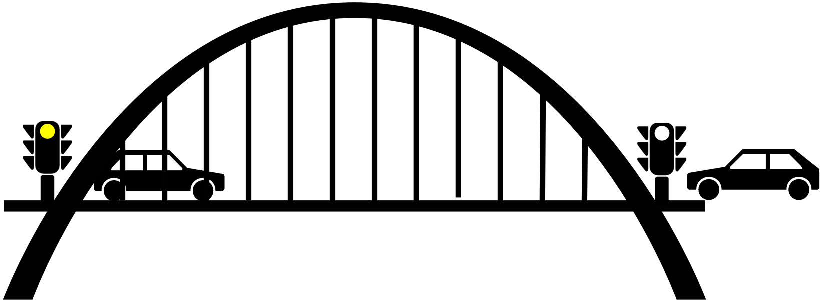
Critical-Section Problem - Solution 2

Solution 2 - Lock Variables

- **Progress** - yes, cars only wait for other cars either waiting or crossing the bridge.
- **Bounded waiting** - no, a car may be very unlucky and keep missing out on crossing the bridge.
- **Mutual exclusion** - no, two cars could end up trying to cross the bridge at the same time.

Critical-Section Problem - Solution 3

Solution 3 - could we use a separate light for each car? Each car turns its light on and then checks the other light. When the other light is off it will cross the bridge and then turn off its light.



Solution 3

Critical-Section Problem - Solution 3

Solution 3

Left car:

```
while(true) {
    light[LEFT] = ON;
    while(light[RIGHT] == ON) {
        // wait
    }
    // cross
    light[LEFT] = OFF;
    // other tasks
}
```

Right car:

```
while(true) {
    light[RIGHT] = ON;
    while(light[LEFT] == ON) {
        // wait
    }
    // cross
    light[RIGHT] = OFF;
    // other tasks
}
```

Critical-Section Problem - Solution 3

This approach solves the mutual exclusion problem - a car cannot enter the bridge at the same time as the other.

However it does introduction another problem:

1. Left car arrives and turns its light ON
2. Right car arrives and turns its light ON
3. Left car waits for right light to turn OFF
4. Right car waits for left light to turn OFF
5. Both cars wait forever!

Deadlock

This problem is called **deadlock**, both of the cars are waiting for the other car to finish crossing the bridge.

As neither car is actually crossing the bridge then both cars will sit there and wait forever.

Obviously humans are too impatient to wait forever but computers are more than happy to.

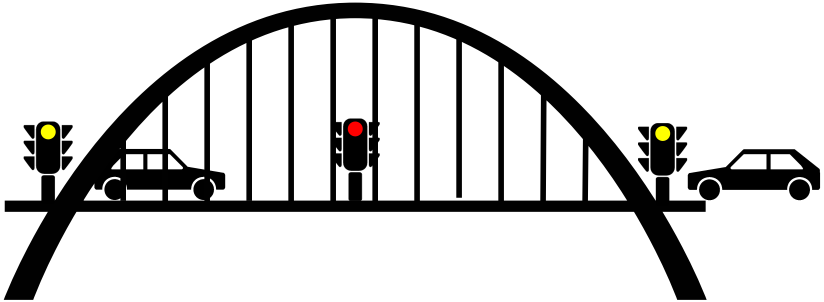
Critical-Section Problem - Solution 3

Solution 3 - Lock Variables

- **Mutual exclusion** - yes, the cars cannot both enter the bridge at the same time
- **Progress** - no, both cars can get stuck forever.
- **Bounded waiting** - no, as above.

Critical-Section Problem - Solution 4

Solution 4 - Petersons Algorithm - combination of solution 1 and solution 3.


Solution 4 - Peterson's Algorithm

Critical-Section Problem - Solution 4

Solution 4 - Peterson's Algorithm

Left car:

```
while(true) {
    light[LEFT] = ON;
    light[CENTRE] = RED;
    while(light[RIGHT] == ON &&
            light[CENTRE]==RED) {
        // wait
    }
    // cross
    light[LEFT] = OFF;
    // other tasks
}
```

Right car:

```
while(true) {
    light[RIGHT] = ON;
    light[CENTRE] = GREEN;
    while(light[LEFT] == ON &&
            light[CENTRE]==GREEN) {
        // wait
    }
    // cross
    light[RIGHT] = OFF;
    // other tasks
}
```

Critical-Section Problem - Solution 4

Each light registers the presence of a car wishing to cross the bridge and the central light makes the cars take turns at crossing.

The cars can only cross the bridge (enter the critical section) if the other light is off or its this cars turn. Both lights may be on at the same time but turn can only be `RED` or `GREEN` but not both.

Critical-Section Problem - Solution 4

For the `LEFT` car to enter the bridge, either `light[RIGHT] == OFF` or `light[CENTRE] == GREEN`.

If the `RIGHT` car comes along at the same time it will set `light[RIGHT] = ON` but then set `light[CENTRE] = GREEN` which means that the `LEFT` car should be crossing anyway.

A similar argument holds for the `RIGHT` car.

Critical-Section Problem - Solution 4

A car can only be prevented from entering the critical section (the bridge) if there is another car either crossing or waiting to cross. If there is no other car then it will immediately start crossing the bridge.

If the `LEFT` car arrives at the while loop and `light[RIGHT] == OFF` then the `LEFT` car can immediately start to cross. It will only wait if `light[RIGHT] == ON` and `light[CENTRE] == RED`.

Critical-Section Problem - Solution 4

A car will also have to wait for at most one other car to cross first. Assume the `RIGHT` car is waiting for the `LEFT` car to cross.

Once the `LEFT` car finishes crossing it will set `light[LEFT] = OFF`, if another car manages to arrive at the left side of the bridge it will set `light[LEFT] = ON` but then set `light[CENTRE] = RED` which means that the `RIGHT` car will get to cross the bridge.

Critical-Section Problem - Solution 4

Solution 4 - Peterson's Algorithm

- **Mutual exclusion** - yes, two cars cannot enter the bridge at the same time.
- **Progress** - yes, cars will only wait if there is another car either waiting or crossing.
- **Bounded waiting** - yes, each car has to wait for at most one other car.

Summary

- Race-conditions
- Critical-section problem
- Software solutions
- Peterson's Algorithm