

Liam O'Connor

CSE, UNSW (and data61)

Semester 2 2018

Recalling Definitions

Definition

Concurrency is an abstraction for the programmer, allowing programs to be structured as multiple **threads of control**, called *processes*. These processes may communicate in various ways.

Example Applications: Servers, OS Kernels, GUI applications.

Concurrent Programs

Consider the following concurrent processes, sharing a variable n .

var $n := 0$		
$p_1: \text{var } x := n;$	$q_1: \text{var } y := n;$	$r_1: \text{var } z := n;$
$p_2: n := x + 1;$	$q_2: n := y - 1;$	$r_2: n := z + 1;$

Typically concurrent semantics are approximated by **interleaving** steps from each process.

Question

What are the possible returned values?

A Sobering Realisation

How many scenarios are there for a program with n processes consisting of m steps each?

	$n = 2$	3	4	5	6
$m = 2$	6	90	2520	113400	$2^{22.8}$
3	20	1680	$2^{18.4}$	$2^{27.3}$	$2^{36.9}$
4	70	34650	$2^{25.9}$	$2^{38.1}$	$2^{51.5}$
5	252	$2^{19.5}$	$2^{33.4}$	$2^{49.1}$	$2^{66.2}$
6	924	$2^{24.0}$	$2^{41.0}$	$2^{60.2}$	$2^{81.1}$

$$\frac{(nm)!}{m!^n}$$

Volatile Variables

var $y, z := 0, 0$	
p_1 : var x ;	q_1 : $y := 1$;
p_2 : $x := y + z$;	q_2 : $z := 2$;

Question

What are the possible final values of x ?

What about $x = 2$? Is that possible?

It **is** possible, as we cannot guarantee that the statement p_2 is executed **atomically** — that is, as one step.

Typically, we require that each statement only accesses (reads from or writes to) at most **one** shared variable at a time. Otherwise, we cannot guarantee that each statement is one atomic step. This is called the **limited critical reference** restriction.

Atomicity

We would like to group multiple steps into one atomic step, called a *critical section*. This allows us to reduce the number of possible interleavings to ensure consistent results.

A sketch of the problem can be outlined as follows:

forever do <i>non-critical section</i> <i>pre-protocol</i> critical section <i>post-protocol</i>	forever do <i>non-critical section</i> <i>pre-protocol</i> critical section <i>post-protocol</i>
--	--

The non-critical section models the possibility that a process may do something else. It can take any amount of time (even infinite). Our task is to find a pre- and post-protocol such that certain *atomicity properties* are satisfied.

Desiderata

We want to ensure two main properties:

- **Mutual Exclusion** No two processes are in their critical section at the same time.
- **Eventual Entry** (or *starvation-freedom*) Once it enters its pre-protocol, a process will eventually be able to execute its critical section.

Question

Which is safety and which is liveness?

Mutex is safety, Eventual Entry is liveness.

First Attempt

We can implement **await** using primitive machine instructions or OS syscalls, or even using a busy-waiting loop.

var <i>turn</i> := 1	
forever do	forever do
<i>p</i> ₁ <i>non-critical section</i>	<i>q</i> ₁ <i>non-critical section</i>
<i>p</i> ₂ await <i>turn</i> = 1;	<i>q</i> ₂ await <i>turn</i> = 2;
<i>p</i> ₃ critical section	<i>q</i> ₃ critical section
<i>p</i> ₄ <i>turn</i> := 2	<i>q</i> ₄ <i>turn</i> := 1

Question

Mutual Exclusion? Yup!

Eventual Entry? Nope! What if *q*₁ never finishes?

Second Attempt

var wantp, wantq := False, False	
forever do	forever do
p ₁ <i>non-critical section</i>	q ₁ <i>non-critical section</i>
p ₂ await wantq = False;	q ₂ await wantp = False;
p ₃ wantp := True;	q ₃ wantq := True;
p ₄ critical section	q ₄ critical section
p ₇ wantp := False	q ₇ wantq := False

Mutual exclusion is violated if they execute in lock-step (i.e. p₁q₁p₂q₂p₃q₃ etc.)

Third Attempt

var <i>wantp, wantq</i> := False, False	
forever do	forever do
<i>p</i> ₁ <i>non-critical section</i>	<i>q</i> ₁ <i>non-critical section</i>
<i>p</i> ₂ <i>wantp</i> := True;	<i>q</i> ₂ <i>wantq</i> := True;
<i>p</i> ₃ await <i>wantq</i> = False;	<i>q</i> ₃ await <i>wantp</i> = False;
<i>p</i> ₄ critical section	<i>q</i> ₄ critical section
<i>p</i> ₇ <i>wantp</i> := False	<i>q</i> ₇ <i>wantq</i> := False

Now we have a **stuck state** (or **deadlock**) if they proceed in lock step, so this violates **eventual entry** also.

Fourth Attempt

var wantp, wantq := False, False	
forever do	forever do
p ₁ <i>non-critical section</i>	q ₁ <i>non-critical section</i>
p ₂ <i>wantp := True;</i>	q ₂ <i>wantq := True;</i>
p ₃ while wantq do	q ₃ while wantp do
p ₄ <i>wantp := False;</i>	q ₄ <i>wantq := False;</i>
p ₅ <i>wantp := True</i>	q ₅ <i>wantq := True</i>
od	od
p ₆ critical section	q ₆ critical section
p ₇ <i>wantp := False</i>	q ₇ <i>wantq := False</i>

We have replaced the **deadlock** with **live lock** (looping) if they continuously proceed in lock-step. Still potentially violates eventual entry.

Fifth Attempt

var <i>wantp</i> , <i>wantq</i> := False, False var <i>turn</i> := 1	
forever do p ₁ <i>non-critical section</i> p ₂ <i>wantp</i> = True; p ₃ while <i>wantq</i> do p ₄ if <i>turn</i> = 2 then p ₅ <i>wantp</i> := False; p ₆ await <i>turn</i> = 1; p ₇ <i>wantp</i> := True fi od p ₈ critical section p ₉ <i>turn</i> := 2 p ₁₀ <i>wantp</i> := False	forever do q ₁ <i>non-critical section</i> q ₂ <i>wantq</i> = True; q ₃ while <i>wantp</i> do q ₄ if <i>turn</i> = 1 then q ₅ <i>wantq</i> := False; q ₆ await <i>turn</i> = 2; q ₇ <i>wantq</i> := True fi od q ₈ critical section q ₉ <i>turn</i> := 1 q ₁₀ <i>wantq</i> := False

Reviewing this attempt

The fifth attempt (**Dekker's algorithm**) works well except if the scheduler pathologically tries to run the loop at $q_3 \cdots q_7$ when $turn = 2$ over and over rather than run the process p (or vice versa).

What would we need to assume to prevent this?

Fairness

The *fairness assumption* means that if a process **can** always make a move, it will **eventually** be scheduled to make that move.

With this assumption, Dekker's algorithm is correct.

Machine Instructions

There exists algorithms to generalise this to any number of processes (**Peterson's algorithm**), but they're outside the scope of this course.

What about if we had a single **machine instruction** to swap two values **atomically**, XC?

var common := 1	
var tp := 0	var tq := 0
forever do	forever do
p1 <i>non-critical section</i>	q1 <i>non-critical section</i>
repeat	repeat
p2 XC(<i>tp, common</i>)	q2 XC(<i>tq, common</i>);
p3 until tp = 1	q3 until tq = 1
p4 critical section	q4 critical section
p5 XC(<i>tp, common</i>)	q7 XC(<i>tq, common</i>)

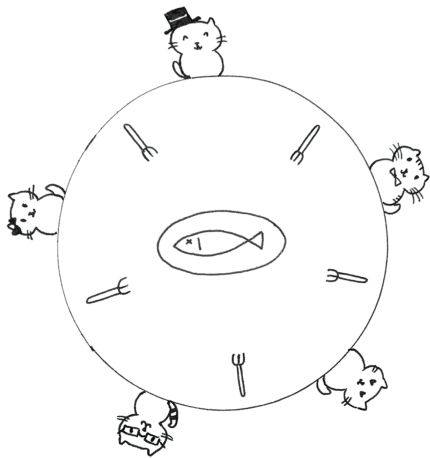
Locks

The variable *common* is called a *lock*. A lock is the most common means of concurrency control in a programming language implementation. Typically it is abstracted into an abstract data type, with two operations:

- *Taking* the lock — the first exchange (step p_2/q_2)
- *Releasing* the lock — the second exchange (step p_5/q_5)

<i>var lock</i>	
forever do	forever do
p_1 <i>non-critical section</i>	q_1 <i>non-critical section</i>
p_2 take (<i>lock</i>)	q_2 take (<i>lock</i>);
p_3 critical section	q_3 critical section
p_4 release (<i>lock</i>)	q_4 release (<i>lock</i>);

Dining Philosophers



Five philosophers sit around a dining table with a huge bowl of spaghetti in the centre, five plates, and five forks, all laid out evenly. For whatever reason, philosophers can eat spaghetti only with **two** forks^a. The philosophers would like to alternate between eating and thinking.

^aThis is obviously a poor adaptation of an old problem from the East where requiring two chopsticks is more convincing.

Looks like Critical Sections

```
forever do
  think
  pre-protocol
  eat
  post-protocol
```

Lets try using locks!

For philosopher $i \in 0 \dots 4$:

```
 $f_0, f_1, f_2, f_3, f_4$ 
```

```
forever do
  think
  take( $f_i$ )
  take( $f_{(i+1) \bmod 5}$ )
  eat
  release( $f_i$ )
  release( $f_{(i+1) \bmod 5}$ )
```

Deadlock is possible (consider lockstep).

Fixing the Issue

f_0, f_1, f_2, f_3, f_4	
Philosophers 0...3	Philosopher 4
forever do <i>think</i> take (f_i) take ($f_{(i+1) \bmod 5}$) <i>eat</i> release (f_i) release ($f_{(i+1) \bmod 5}$)	forever do <i>think</i> take (f_0) take (f_4) <i>eat</i> release (f_0) release (f_4)

We have to enforce a **global ordering** of locks.