Concurrency Appreciation

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: **var** $x := n$;	q_1: **var** $y := n$;	r_1: **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?

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	$2^{24.0}$	$2^{41.0}$	$2^{60.2}$	$2^{81.1}$

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?

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?

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.

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.

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	**forever do**
non-critical section	*non-critical section*
pre-protocol	*pre-protocol*
critical section	**critical section**
post-protocol	*post-protocol*

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	**forever do**
non-critical section	*non-critical section*
pre-protocol	*pre-protocol*
critical section	**critical section**
post-protocol	*post-protocol*

The non-critical section models the possibility that a process may do something else. It can take any amount of time (even infinite).

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	**forever do**
non-critical section	*non-critical section*
pre-protocol	*pre-protocol*
critical section	**critical section**
post-protocol	*post-protocol*

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?

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 *turn* := 1	
forever do	**forever do**
p_1 *non-critical section*	q_1 *non-critical section*
p_2 **await** *turn* = 1;	q_2 **await** *turn* = 2;
p_3 **critical section**	q_3 **critical section**
p_4 *turn* := 2	q_4 *turn* := 1

Question

Mutual Exclusion?

First Attempt

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

var *turn* := 1	
forever do	**forever do**
p_1 *non-critical section*	q_1 *non-critical section*
p_2 **await** *turn* = 1;	q_2 **await** *turn* = 2;
p_3 **critical section**	q_3 **critical section**
p_4 *turn* := 2	q_4 *turn* := 1

Question

Mutual Exclusion? Yup!

First Attempt

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

var *turn* := 1	
forever do	**forever do**
p_1 *non-critical section*	q_1 *non-critical section*
p_2 **await** *turn* = 1;	q_2 **await** *turn* = 2;
p_3 **critical section**	q_3 **critical section**
p_4 *turn* := 2	q_4 *turn* := 1

Question

Mutual Exclusion? Yup!
Eventual Entry?

First Attempt

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

var $turn := 1$	
forever do	**forever do**
p_1 *non-critical section*	q_1 *non-critical section*
p_2 **await** $turn = 1$;	q_2 **await** $turn = 2$;
p_3 **critical section**	q_3 **critical section**
p_4 $turn := 2$	q_4 $turn := 1$

Question

Mutual Exclusion? Yup!
Eventual Entry? Nope!

First Attempt

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

var *turn* := 1	
forever do	**forever do**
p_1 *non-critical section*	q_1 *non-critical section*
p_2 **await** *turn* = 1;	q_2 **await** *turn* = 2;
p_3 **critical section**	q_3 **critical section**
p_4 *turn* := 2	q_4 *turn* := 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 *non-critical section*	q_1 *non-critical section*
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

Second Attempt

var *wantp*, *wantq* := False, False	
forever do	**forever do**
p_1 *non-critical section*	q_1 *non-critical section*
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. $p_1q_1p_2q_2p_3q_3$ etc.)

Third Attempt

var *wantp*, *wantq* := False, False	
forever do	**forever do**
p_1 *non-critical section*	q_1 *non-critical section*
p_2 *wantp* := True;	q_2 *wantq* := True;
p_3 **await** *wantq* = False;	q_3 **await** *wantp* = False;
p_4 **critical section**	q_4 **critical section**
p_7 *wantp* := False	q_7 *wantq* := False

Third Attempt

var *wantp*, *wantq* := False, False	
forever do	**forever do**
p_1 *non-critical section*	q_1 *non-critical section*
p_2 *wantp* := True;	q_2 *wantq* := True;
p_3 **await** *wantq* = False;	q_3 **await** *wantp* = False;
p_4 **critical section**	q_4 **critical section**
p_7 *wantp* := False	q_7 *wantq* := 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 *non-critical section*	q_1 *non-critical section*
p_2 *wantp* := True;	q_2 *wantq* := True;
p_3 **while** *wantq* **do**	q_3 **while** *wantp* **do**
p_4 *wantp* := False;	q_4 *wantq* := False;
p_5 *wantp* := True	q_5 *wantq* := True
od	**od**
p_6 **critical section**	q_6 **critical section**
p_7 *wantp* := False	q_7 *wantq* := False

Fourth Attempt

var *wantp*, *wantq* := False, False	
forever do	**forever do**
p_1 *non-critical section*	q_1 *non-critical section*
p_2 *wantp* := True;	q_2 *wantq* := True;
p_3 **while** *wantq* **do**	q_3 **while** *wantp* **do**
p_4 *wantp* := False;	q_4 *wantq* := False;
p_5 *wantp* := True	q_5 *wantq* := True
od	**od**
p_6 **critical section**	q_6 **critical section**
p_7 *wantp* := False	q_7 *wantq* := False

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

Fifth Attempt

var *wantp*, *wantq* := False, False
var *turn* := 1

	forever do		**forever do**
p_1	*non-critical section*	q_1	*non-critical section*
p_2	*wantp* = True;	q_2	*wantq* = True;
p_3	**while** *wantq* **do**	q_3	**while** *wantp* **do**
p_4	**if** *turn* = 2 **then**	q_4	**if** *turn* = 1 **then**
p_5	*wantp* := False;	q_5	*wantq* := False;
p_6	**await** *turn* = 1;	q_6	**await** *turn* = 2;
p_7	*wantp* := True	q_7	*wantq* := True
	fi		**fi**
	od		**od**
p_8	**critical section**	q_8	**critical section**
p_9	*turn* := 2	q_9	*turn* := 1
p_{10}	*wantp* := False	q_{10}	*wantq* := 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?

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`?

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**
p_1 *non-critical section*	q_1 *non-critical section*
repeat	**repeat**
p_2 XC(*tp*, *common*)	q_2 XC(*tq*, *common*);
p_3 **until** *tp* = 1	q_3 **until** *tq* = 1
p_4 **critical section**	q_4 **critical section**
p_5 XC(*tp*, *common*)	q_7 XC(*tq*, *common*)

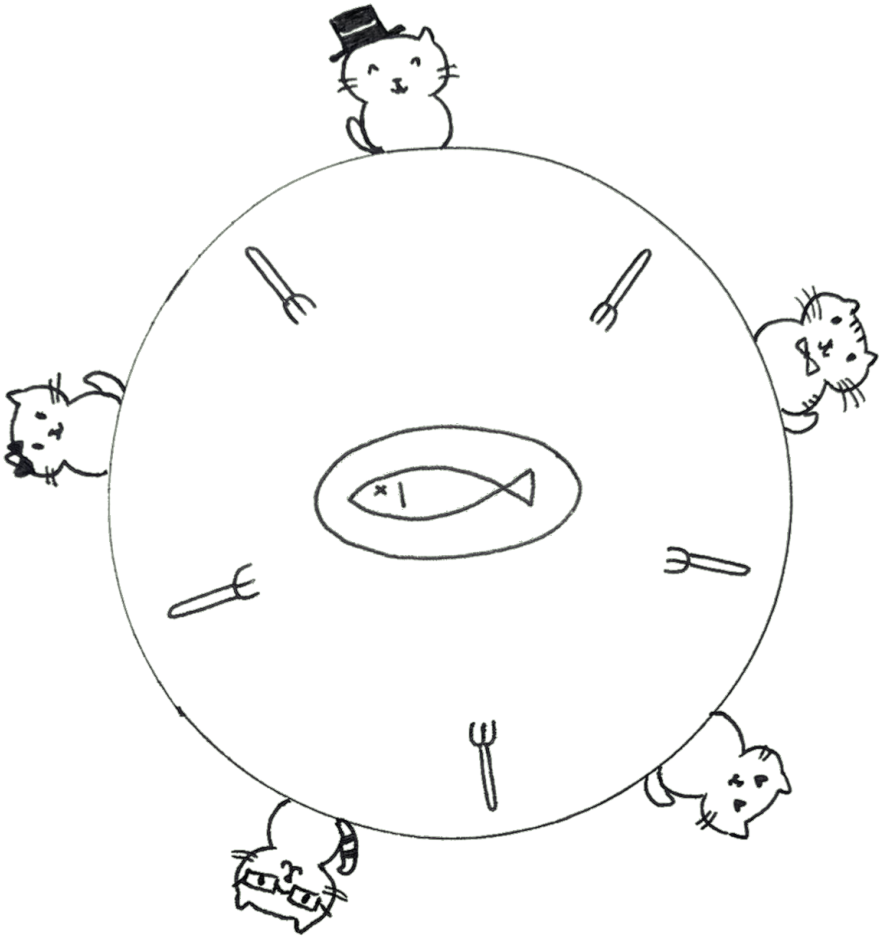
Locks

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)

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

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.

[a] This 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!

Looks like Critical Sections

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

Lets try using locks!

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

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**($f_{(i+1) \bmod 5}$)

Looks like Critical Sections

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

Lets try using locks!

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

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	**forever do**
think	*think*
take(f_i)	**take**(f_0)
take($f_{(i+1) \bmod 5}$)	**take**(f_4)
eat	*eat*
release(f_i)	**release**(f_0)
release($f_{(i+1) \bmod 5}$)	**release**(f_4)

Fixing the Issue

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

We have to enforce a global ordering of locks.