Concurrency Appr

Christine Rizkallah
CSE, UNSW

Term 3 2020

Definitions

Definition

Concurrency is an abstraction for the programmer, allowing programs to be structured as multiple thread unicate in various ways.

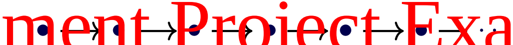
Example Applications: Servers, OS Kernels, GUI applications.

Anti-definition

Concurrency is **not** *parallelism*, which is a means to explo order to improve performance.

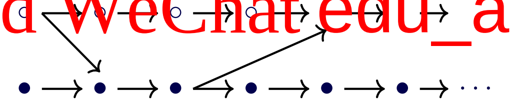
Sequential vs Concurrent

We could consider a *sequential* program as a sequence (or *total order*) of *actions*:



The ordering here i

A concurrent program is not a total order but a partial order.



This means that there are now multiple possible *interleavings* of these actions — our program is non-deterministic where the interleaving is selected by the scheduler.

Concurrent Programs

Consider the following concurrent processes, sharing a variable n.

		n 1;

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?

4	70	34650	2	2	2
5	252	$2^{19.5}$	2^{3}		
6	924	$2^{24.0}$	2^{4}		

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

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?

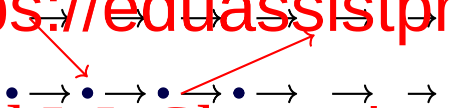
What about $x = 2$? Is that possible?

It is possible, as we cannot guarantee that the statement [illegible] mically — 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.

Synchronisation

In order to reduce the number of possible interleavings, we must allow processes to synchronise thei .



The red arrows are synchronisations.

Atomicity

The basic unit of synchronisation we would like to implement is to group multiple steps into one atomic step, called a *critical section*.

A sketch of the problem can be outlined as follows:

critical section	**c**
post-protocol	*p*

The non-critical section models the possibility that a process 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 Ex** e.
- **Eventual E** process will eventu

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* $:= 1$	
p_3 **critical section**	q_3
p_4 *turn* $:= 2$	q_4

Question

Mutual Exclusion? Yup!

Eventual Entry? Nope! What if q_1 never finishes?

Second Attempt

var *wantp*, *wantq* := False, False	
	e;
p_3 *wantp* := True;	q_3 *wantq* := True;
p_4 **critical section**	q_4
p_7 *wantp* := False	q_7

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	
3	3 e;
p_4 **critical section**	q_4
p_7 *wantp* := False	q_7

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_5 *wantp* := True	q_5
od	
p_6 **critical section**	q_6
p_7 *wantp* := False	q_7

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**
	n
p_5 $wantp :=$ False;	q_5
p_6 **await** $turn = 1$;	q_6
p_7 $wantp :=$ True	q_7
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 q_7 when $turn = 2$ over and over rather than run the proces

What would we ne

Fairness

The *fairness assumption* means that if a process can alwa 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?

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

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* th
- *Releasing*

var *lock*	
forever do	**for**
p_1 *non-critical section*	q_1
p_2 **take** (*lock*)	q_2
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,

l

er

eat

[a].

[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
```

```
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 $0 \ldots 3$	Philosopher 4
take$(f_{(i+1) \bmod 5})$ *eat* **release**(f_i) **release**$(f_{(i+1) \bmod 5})$	$_4$

We have to enforce a global ordering of locks.