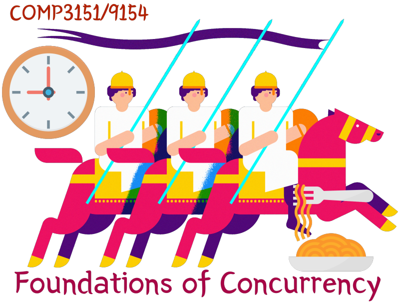


Hardware-Assisted Critical Sections

Johannes Åman Pohjola
CSE, UNSW
Term 2 2022

Where we are at

In the last lecture we introduced efficient algorithms for critical section solutions for N processes.

In this lecture, we will talk more about hardware-assisted critical sections and how they are used to implement a basic unit of synchronisation, called a *lock* or *mutex*.

Machine Instructions

Recall the exchange solution:

bit common ← 1	
bit tp ← 0	bit 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_5 XC(tq, common)

Machine Instructions

Now let's see the test and set solution:

$$\mathtt{TS}(x, y) \quad \equiv \quad x, y := y, 1 \text{ (atomically)}$$

bit common ← 0	
bit tp	bit tq
forever do	**forever do**
p_1 *non-critical section*	q_1 *non-critical section*
repeat	**repeat**
p_2 TS(tp, common)	q_2 TS(tq, common);
p_3 **until** tp = 0	q_3 **until** tq = 0
p_4 **critical section**	q_4 **critical section**
p_5 common ← 0	q_5 common ← 0

Locks

The variable *common* is called a *lock* (or *mutex*). 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*);

Architectural Problems

In a multiprocessor execution environment, reads and writes to variables initially only read from/write to cache.

Architectural Problems

In a multiprocessor execution environment, reads and writes to variables initially only read from/write to cache.

Writes to shared variables must eventually trigger a write-back to main memory over the bus.

Architectural Problems

In a multiprocessor execution environment, reads and writes to variables initially only read from/write to cache.
Writes to shared variables must eventually trigger a write-back to main memory over the bus.
These writes cause the shared variable to be cache invalidated.
Each processor must now consult main memory when reading in order to get an up-to-date value.

Architectural Problems

In a multiprocessor execution environment, reads and writes to variables initially only read from/write to cache.

Writes to shared variables must eventually trigger a write-back to main memory over the bus.

These writes cause the shared variable to be cache invalidated.

Each processor must now consult main memory when reading in order to get an up-to-date value.

The problem: Bus traffic is limited by hardware.

Architectural Problems

In a multiprocessor execution environment, reads and writes to variables initially only read from/write to cache.

Writes to shared variables must eventually trigger a write-back to main memory over the bus.

These writes cause the shared variable to be cache invalidated.

Each processor must now consult main memory when reading in order to get an up-to-date value.

The problem: Bus traffic is limited by hardware.

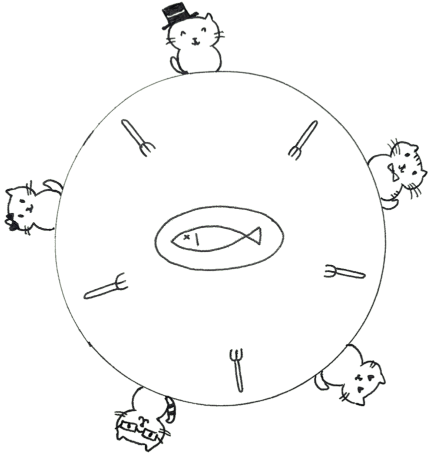
With these instructions...

The processes spin while waiting, writing to shared variables on each spin. This quickly causes the bus to become jammed, and can delay processes from releasing the lock (c.f. the *thundering herd* problem).

The solution?

Johannes will demonstrate in Promela the test-and-test-and-set solution (and a similar approach for exchange).

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 would be more convincing with chopsticks. Blame Tony Hoare.

Looks like Critical Sections

forever do

think

pre-protocol

eat

post-protocol

Looks like Critical Sections

forever do
    think
    pre-protocol
    eat
    post-protocol

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

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 $0 \ldots 3$	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 $0 \ldots 3$	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.

What now?

- Assignment 0 deadline is on Monday.
- Assignment 1 comes out next week! Please find a partner!
- Next week: We will look at semaphores and monitors.