Hardware-Assisted Critical Sections

Johannes Åman Pohjola
CSIRO’s Data61 (and UNSW)
Term 2 2021
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.
Recall the exchange solution:

<table>
<thead>
<tr>
<th>bit common ← 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit tp ← 0</td>
</tr>
<tr>
<td>forever do</td>
</tr>
<tr>
<td>p₁ non-critical section</td>
</tr>
<tr>
<td>repeat</td>
</tr>
<tr>
<td>p₂ XC(tp, common)</td>
</tr>
<tr>
<td>p₃ until tp = 1</td>
</tr>
<tr>
<td>p₄ critical section</td>
</tr>
<tr>
<td>p₅ XC(tp, common)</td>
</tr>
</tbody>
</table>

| bit tq ← 0    |
| forever do    |
| q₁ non-critical section |
| repeat         |
| q₂ XC(tq, common) |
| q₃ until tq = 1 |
| q₄ critical section |
| q₅ XC(tq, common) |
Now let’s see the test and set solution:

\[ \text{TS}(x, y) \equiv x, y := y, 1 \text{ (atomically)} \]
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
p_1  non-critical section
p_2  take (lock)
p_3  critical section
p_4  release (lock)
forever do
q_1  non-critical section
q_2  take (lock);
q_3  critical section
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.
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. The philosophers would like to alternate between eating and thinking.

\[\text{This would be more convincing with chopsticks. Blame Tony Hoare.}\]
Looks like Critical Sections

\[
\text{forever do}
\begin{align*}
\text{think} \\
\text{pre-protocol} \\
\text{eat} \\
\text{post-protocol}
\end{align*}
\]

Deadlock is possible (consider lockstep).
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($f_{(i+1) \mod 5}$)
  eat
  release($f_i$)
  release($f_{(i+1) \mod 5}$)
```

Deadlock is possible (consider lockstep).
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($f_{(i+1) \mod 5}$)
  eat
  release($f_i$)
  release($f_{(i+1) \mod 5}$)

Deadlock is possible (consider lockstep).
## Fixing the Issue

<table>
<thead>
<tr>
<th>$f_0, f_1, f_2, f_3, f_4$</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Philosophers 0...3</strong></td>
<td><strong>Philosopher 4</strong></td>
</tr>
<tr>
<td><strong>forever do</strong></td>
<td><strong>forever do</strong></td>
</tr>
<tr>
<td><em>think</em></td>
<td><em>think</em></td>
</tr>
<tr>
<td><em>take($f_i$)</em></td>
<td><em>take($f_0$)</em></td>
</tr>
<tr>
<td><em>take($f</em>{(i+1) \mod 5}$)_</td>
<td><em>take($f_4$)</em></td>
</tr>
<tr>
<td><em>eat</em></td>
<td><em>eat</em></td>
</tr>
<tr>
<td><em>release($f_i$)</em></td>
<td><em>release($f_0$)</em></td>
</tr>
<tr>
<td><em>release($f</em>{(i+1) \mod 5}$)_</td>
<td><em>release($f_4$)</em></td>
</tr>
</tbody>
</table>

We have to enforce a global ordering of locks.
Fixing the Issue

<table>
<thead>
<tr>
<th>$f_0, f_1, f_2, f_3, f_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Philosophers 0...3</td>
</tr>
<tr>
<td><strong>forever do</strong></td>
</tr>
<tr>
<td><strong>think</strong></td>
</tr>
<tr>
<td><strong>take($f_i$)</strong></td>
</tr>
<tr>
<td><strong>take($f_{(i+1) \mod 5}$)</strong></td>
</tr>
<tr>
<td><strong>eat</strong></td>
</tr>
<tr>
<td><strong>release($f_i$)</strong></td>
</tr>
<tr>
<td><strong>release($f_{(i+1) \mod 5}$)</strong></td>
</tr>
</tbody>
</table>

We have to enforce a **global ordering** of locks.
What now?

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