MANCHESTER 1824

The University of Manchester

Chip Multiprocessors COMP35112

Lecture 7 - Hardware Support for Synchronisation

Graham Riley

Implementing Synchronisation

- Most shared memory parallel programming requires that we implement synchronisation to control how threads access shared resources
- Several forms in use but they are all closely related and most can be built from the others
- On shared memory multiprocessor systems, the implementation usually requires hardware support
- Will consider one of the simplest constructs – a binary semaphore (first encountered in COMP25111)
- Think of bus-based system with snoopy bus…

Example – Binary Semaphore

- A single shared ‘boolean’ variable **S** to ‘protect’ a shared resource:
  - S == 1 → resource is free
  - S == 0 → resource is in use
- Semaphore operations (both **atomic**)
  - **`wait(S)`** : wait until S!=0 then set S=0
  - **`signal(S)`** : set S=1

Use of Semaphore to Protect 'Critical Sections'

thread 1	thread 2
wait(S)	wait(S)
print	print
signal(S)	signal(S)

(initialise S = 1)

(critical section → operating on some shared resource)

‘Atomic’ Action Needed

How to implement **`wait(S)`**?

```
while (S == 0) ;
S=0;
```

```
loop: ldr  r1,r2
      cmp r1,#0
      beq  loop
      str #0, r2
```

(address of S in r2)

what if some other thread gets in here ← and changes the state of S in the middle of our code?

Atomic Action

- Way to avoid this problem is to ensure that **`wait()`** is 'indivisible'
- But this requires special instructions to be supported in hardware
- There exists a compromise, between complexity and performance
- Note that variable **S** may be cached and the desired 'indivisible' behaviour might require coherence operations in the cache

Test_and_Set Instruction

- Simple solution in older processors – e.g. Motorola 68000 – instruction level behaviour is **atomic**

  tas r2

  If memory location addressed by r2 is 0, set it to 1 and set the processor 'zero' flag – otherwise clear zero flag

  ```
  loop : tas r2
         bnz loop    // branch if [r2] != 0
  ```

- Note: this is the logical inverse of the standard software definition of **`wait(S)`**

Caching the Shared Variable **S**

- Operation of test_and_set (tas) is reasonably obvious if **S** is a single shared variable **in memory**
- Just read and then write (if necessary) the single variable – although it should be noted that such an atomic 'read-modify-write' operation must lock the access to memory and is hence likely to be expensive (i.e. slow)
- However, the variable **S** is naturally shared – this is the fundamental purpose of a semaphore
- Processors are therefore likely to end up with a copy in their cache

Test_and_Set and the Cache

- Assume shared variable is in cache
- If a tas succeeds (reads 0) it must then write a 1
- When it starts, it doesn't know whether it needs to do a write – in which case it will need to send an invalidate message to other cores
- Requires processor to 'lock' the snoopy bus for every multiprocessor tas operation – just in case – it cannot let any other core do a write
- But if it reads a '1' (busy), this locking of the bus is wasted

Test then Test_and_Set (1)

- Assume one thread has the lock (**S** is busy)
- Another wanting the semaphore will read this busy value and cache it
- It will then sit in a loop continually executing a tas until **S** becomes free
- All this time it will be wasting bus cycles
- There is a simple re-formulation of the ‘wait’ operation that can avoid much of this …

Test then Test_and_Set (2)

- In pseudo-code

```
do {
        while (test(S) == 1);
    } while (test_and_set(S))

loop:   ldr r1,r2
        cmp r1, #1
        beq loop
        tas r2
        bnz loop    // branch if r2 != 0
```

Test and Test_and_Set (3)

- The inner loop ‘spins’ while the **S** variable is seen to be busy
- Once **S** is seen to be free, a tas instruction is attempted
- Hopefully most times it will succeed – will only fail if another thread manages to get in between the cmp and tas instructions
- But waiting loop is only executing a normal ldr most of the time – all internal to core and its cache – so no bus cycles wasted

Other Synchronisation Primitives

- There are other machine level instructions which can be used, such as:
  - Fetch_and_Add – atomic instruction – returns the value of a memory location and increments it
  - Compare_and_Swap – again atomic – compare the value of a memory location with a value (in a register) and swap in another value (in a register) if they are equal
- All these instructions are ‘read-modify-write’ (RMW) with the need to lock the snoopy bus during their execution

Synchronisation without RMW

- All synchronisation operations have a read followed by a write
- To make these atomic, previous instructions do this as a RMW primitive
- But this is not really desirable:
  - Doesn't fit well with modern pipelined and/or split transaction busses, where operations can be overlapped and/or interleaved
  - Doesn't fit well with simple RISC pipelines, where RMW is really a CISC instruction requiring a read, a test and a write

Load_Linked – Store_Conditional

- LL/SC is a synchronisation mechanism used in modern RISC processors (ARM, PowerPC)
- It uses separate load_linked (ldl) and store_conditional (stc) instructions which, in terms of basic functionality, are very similar, but not identical, to ordinary loads and stores
- However, they have additional effects on processor state which allow them, as a pair, to act atomically
  - While avoiding holding the bus until completion

Load_Linked

ldl r1,r2

- Loads r1 with value addressed in memory by r2
- When it is executed, it:
  - sets a special ‘load linked flag’ on the core which executes it
  - records the address (in r2) in a ‘locked address register’ on the core which executes it
- That is, it keeps some state for the ldl

Store_Conditional

stc r1,r2

- Tries to store the value in r1 into the location addressed in memory by r2
  - But only succeeds if the ‘load linked flag’ is set
- The value of the ‘load linked flag’– i.e. whether-or-not the store was successful – is returned in r1
- The ‘load linked flag’ is cleared

The 'Load Linked Flag'

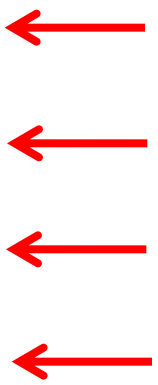
- The state of the 'load linked flag' on a core which has it set is changed **if** a write (from another core) occurs to the locked address and, hence, an invalidate message is sent (**or if** the current thread exits the running state)
- Detected by comparison with 'locked address register' – processor must monitor (snoop) the memory address bus to detect this
- This will occur because a write has occurred to the shared variable and, hence, another core has probably got the semaphore

Semaphore `wait()` Code

```
loop:  ldl r1,r2
       cmp #0,r1        // S == 0 (busy) ?
       beq loop         // if so, try again
       mov #0,r1        // must be free
       stc r1,r2        // S = 0 (busy) if load linked flag set
       cmp #1,r1        // was load linked flag set?
       bne loop         // if not, try again
       …                // otherwise in critical section
                        // a normal st of 1 will signal "free"
```

Why Does it Work?

```
loop:  ldl r1,r2
       cmp #0,r1
       beq loop
       mov #0,r1
       stc r1,r2
       cmp #1,r1
       bne loop
```



if any other core manages to write to [r2] here (or thread is descheduled), the stc will fail and loop will repeat

otherwise everything between ldl and stc must have executed as if 'atomically' so the core has the "lock"

Power of LL/SC

- In an instruction like tas, the load and store can be guaranteed to be atomic by RMW behaviour
- When using LL/SC we know that anything **between** the ldl and the stc has executed atomically (with respect to the synchronisation variable)
- This can be more powerful than tas for certain forms of usage
  - E.g. it is relatively easy and efficient to implement fetch_and_add etc. based on LL/SC, reducing the number of special instructions that need to be supported

Spin Lock

- Note that **`wait(S)`** implemented as discussed may require waiting for a lock by sitting in a loop (sometimes called **busy waiting** or **spinning**)
- More efficient use of resources may result (particularly in multiprocessors) if processor can switch to do useful work elsewhere
- Various more sophisticated forms of locking address this – however, the basic hardware support is the same

Next Lecture

- This lecture has introduced machine instructions that can implement locks and other synchronisation constructs
- The next lecture investigates whether we can use such instructions, **instead of locks**, to implement so-called ‘lock-free data structures’