COMP35112 Chip Multiprocessors

Hardware Support for Synchronisation

Pierre Olivier

Implementing Synchronisation

- Shared-memory programming requires synchronisation mechanisms to protect shared data

Implementing Synchronisation

- Shared-memory programming requires synchronisation mechanisms to protect shared data
- Mechanisms can take several forms
  - But all are closely related and most can be built from the others

Implementing Synchronisation

- Shared-memory programming requires synchronisation mechanisms to protect shared data
- Mechanisms can take several forms
    - But all are closely related and most can be built from the others
- **Their implementation usually requires hardware support**

Implementing Synchronisation

- Shared-memory programming requires synchronisation mechanisms to protect shared data
- Mechanisms can take several forms
    - But all are closely related and most can be built from the others
- **Their implementation usually requires hardware support**
- We'll have a look at one of the simplest constructs
    - A lock called **binary semaphore**, in a processor with a snoopy cache
        - Can be held by at most 1 thread
        - Waiting threads use busy-waiting

Example: Binary Semaphore

- It's a single shared "boolean" variable `S` which value is used to protect a shared resource
    - `S == 0` ➡ resource is free
    - `S == 1` ➡ resource is in use

Example: Binary Semaphore

- It's a single shared "boolean" variable **`S`** which value is used to protect a shared resource
  - `S == 0` ➡ resource is free
  - `S == 1` ➡ resource is in use
- Semaphore operations (should be **atomic**)
  - **`wait(S)`**: wait until `S != 1` then set `S = 1` (i.e. take the lock)
  - **`signal(S)`**: set `S = 0` (i.e. release the lock)

Semaphore Usage to Protect Critical Sections

- **Critical sections** are the code sections where shared resources are manipulated

Thread 1	Thread 2
`wait(S)`	`wait(S)`
`update shared data`	
`signal(S)`	
	`update shared data`
	`signal(S)`

- `S` should be initialised as `0`

Atomicity Needed

- How to implement `wait(S)`?

Atomicity Needed

- How to implement `wait(S)`?

```c
// naive implementation in C:
while(S == 1);
S = 1;
```

Atomicity Needed

- How to implement `wait(S)`?

```c
// naive implementation in C:
while(S == 1);
S = 1;
```

```
// address of `S` in `%r2`
loop: ldr %r1, %r2
      cmp %r1, $1
      beq loop
      str $1, %r2
```

Atomicity Needed

- How to implement `wait(S)`?

```
// naive implementation in C:
while(S == 1);
S = 1;
```

```
// address of `S` in `%r2`
loop: ldr %r1, %r2
      cmp %r1, $1
      beq loop
      str $1, %r2
```

- What if another thread changes the value of `S`?

Thread 1	Thread 2
ldr %r1, %r2	
cmp %r1, $1	ldr %r1, %r2
beq loop	cmp %r1, $1
str $1, %r2	beq loop
	str $1, %r2

Atomicity Needed

- How to implement `wait(S)`?

```
// naive implementation in C:
while(S == 1);
S = 1;
```

```
// address of `S` in `%r2`
loop: ldr %r1, %r2
      cmp %r1, $1
      beq loop
      str $1, %r2
```

- What if another thread changes the value of `S`?

Thread 1	Thread 2
ldr %r1, %r2	
cmp %r1, $1	ldr %r1, %r2
beq loop	cmp %r1, $1
str $1, %r2	beq loop
	str $1, %r2

Both threads got the lock!

The lock itself is a shared data structure...

Atomic Instructions

- We need to ensure that the execution of `wait()` is "indivisible"
  - I.e. it should be **atomic**: once a thread starts to execute `wait()` it should first finish it before any other thread can start it

Atomic Instructions

- We need to ensure that the execution of `wait()` is "indivisible"
  - I.e. it should be **atomic**: once a thread starts to execute `wait()` it should first finish it before any other thread can start it
- Requires special instructions to be supported in hardware: **atomic instructions**
  - Special CPU instructions that realise a few operations atomically
  - Operations are generally a memory load, a comparison, and possibly a memory write

Atomic Instructions

- We need to ensure that the execution of `wait()` is "indivisible"
  - I.e. it should be **atomic**: once a thread starts to execute `wait()` it should first finish it before any other thread can start it
- Requires special instructions to be supported in hardware: **atomic instructions**
  - Special CPU instructions that realise a few operations atomically
  - Operations are generally a memory load, a comparison, and possibly a memory write
- Implementing synchronisation primitives like `wait()` with these instructions involves a compromise between complexity and performance

Atomic Instructions

- We need to ensure that the execution of `wait()` is "indivisible"
  - I.e. it should be **atomic**: once a thread starts to execute `wait()` it should first finish it before any other thread can start it
- Requires special instructions to be supported in hardware: **atomic instructions**
  - Special CPU instructions that realise a few operations atomically
  - Operations are generally a memory load, a comparison, and possibly a memory write
- Implementing synchronisation primitives like `wait()` with these instructions involves a compromise between complexity and performance
- Also note that variable `S` when accessed will be cached, possibly in several core caches, and the desired atomic behaviour might require coherence operations in the cache

Atomic Test-And-Set Instruction

- Simple solution in older CPUs, e.g. Motorola 68K

```
tas %r2
```

- If memory location addressed by `%r2` contains `0`, switch its content to `1` and set the CPU "zero" flag, otherwise clear zero flag.

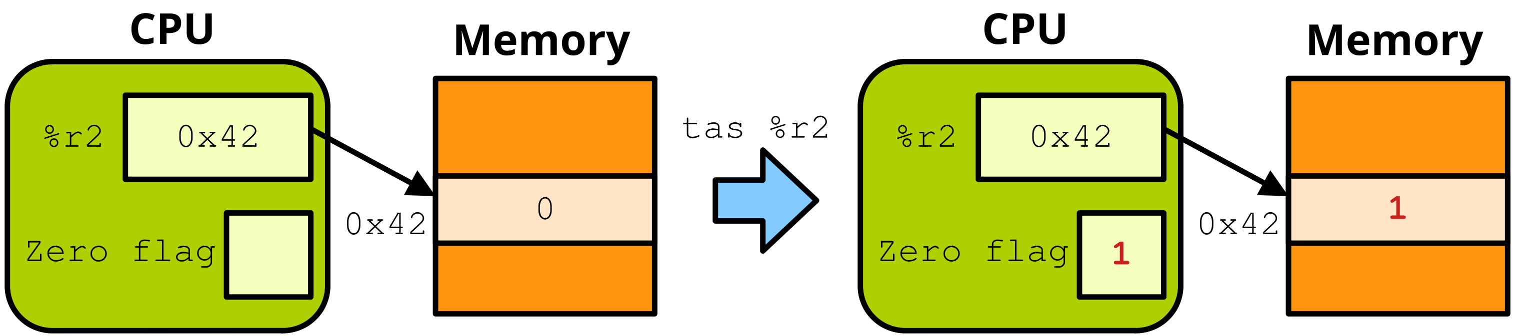
Atomic Test-And-Set Instruction

- Simple solution in older CPUs, e.g. Motorola 68K

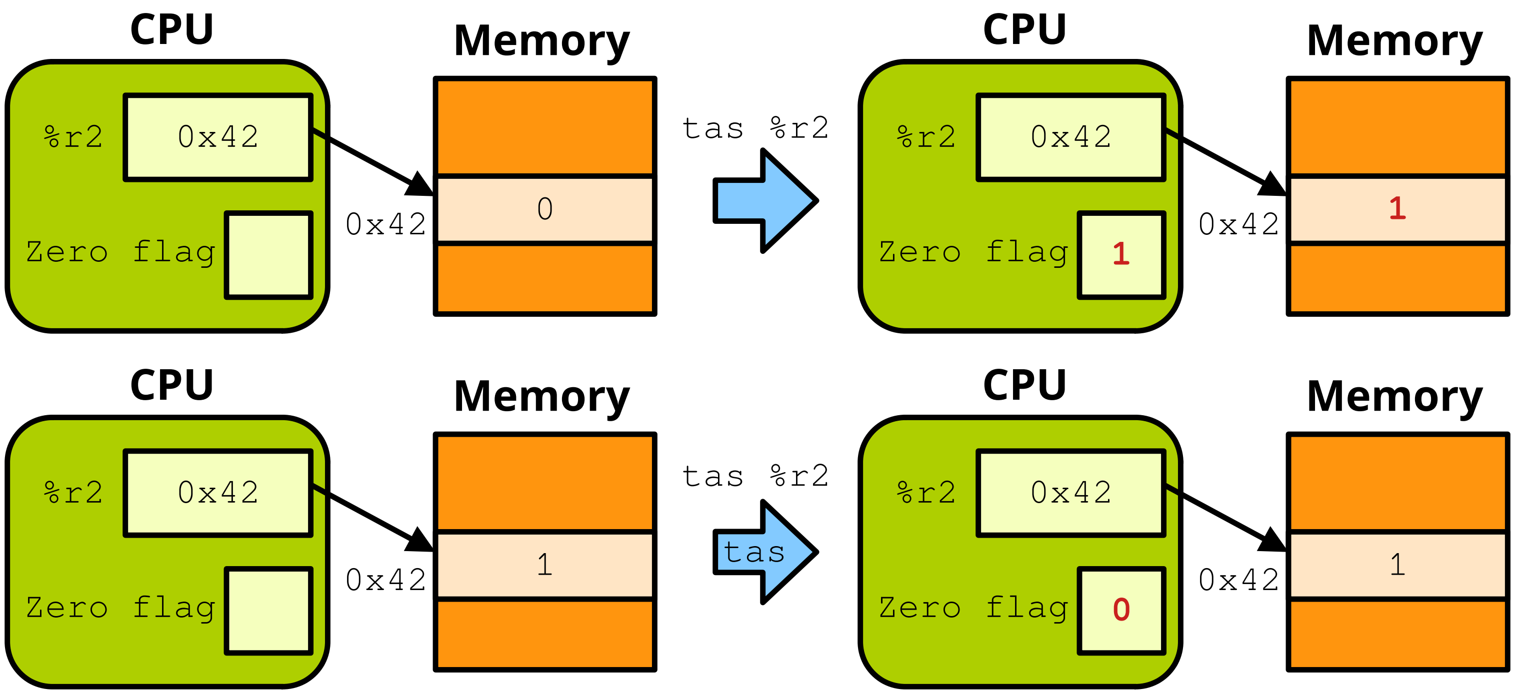
```
tas %r2
```

- If memory location addressed by `%r2` contains `0`, switch its content to `1` and set the CPU "zero" flag, otherwise clear zero flag.
- **Instruction-level behaviour is atomic**
  - Cannot be interrupted
  - No other core can modify what is pointed by `%r2` in memory while the `tas` runs

Atomic Test-And-Set Instruction



Atomic Test-And-Set Instruction



Our Semaphore with `tas`

- Remember that for our semaphore:
  - Lock is free when `S == 0`
  - Lock is taken when `S == 1`
- How to implement `wait()` and `signal` with test-and-set?

Our Semaphore with `tas`

- Remember that for our semaphore:
  - Lock is free when `S == 0`
  - Lock is taken when `S == 1`
- How to implement `wait()` and `signal` with test-and-set?

Wait operation (taking the lock):

```
// Address of S in %r2
// Loops (i.e. wait) while [%r2] != 0
loop: tas %r2
      bnz loop // branch if zero flag not set
```

Signal operation (releasing the lock):

```
// We assume that basic store operations
// are atomic
// Address of S in %r2
str $0, %r2
```

What About the Cache?

- Semaphore operation with test-and-set is reasonably obvious if S is a single shared variable **in memory**
- Just `tas` for `wait()` and `str` for `signal()` on that single variable

What About the Cache?

- Semaphore operation with test-and-set is reasonably obvious if S is a single shared variable **in memory**
- Just `tas` for `wait()` and `str` for `signal()` on that single variable
- `tas` is an atomic **read-modify-write** (RMW) instruction and as such it is expensive:
    - May involve 2 memory accesses (R & W)
    - Locks the access to memory from other processors to ensure atomicity

What About the Cache?

- Semaphore operation with test-and-set is reasonably obvious if S is a single shared variable **in memory**
- Just `tas` for `wait()` and `str` for `signal()` on that single variable
- `tas` is an atomic **read-modify-write** (RMW) instruction and as such it is expensive:
  - May involve 2 memory accesses (R & W)
  - Locks the access to memory from other processors to ensure atomicity
- By definition `S` is shared: this is the fundamental purpose of a semaphore
  - **Processors are therefore likely to end up with a copy of `S` in their cache**

Test-and-Set and the Cache

- Assume shared variable `S` in the cache
- Only when a `tas` succeeds (reads 0) it must then write a 1

Test-and-Set and the Cache

- Assume shared variable `S` in the cache
- Only when a `tas` succeeds (reads 0) it must then write a 1
  - When `tas` starts, don't know if a write will be needed or not...
  - ... if it is, need to send an invalidate message to other cores

Test-and-Set and the Cache

- Assume shared variable `S` in the cache
- Only when a `tas` succeeds (reads 0) it must then write a 1
  - When `tas` starts, don't know if a write will be needed or not...
  - ... if it is, need to send an invalidate message to other cores
  - **So the processor must 'lock' the snoopy bus for every multiprocessor `tas` operation**
    - Cannot let any other core do a write

Test-and-Set and the Cache

- Assume shared variable `S` in the cache
- Only when a `tas` succeeds (reads 0) it must then write a 1
  - When `tas` starts, don't know if a write will be needed or not...
  - ... if it is, need to send an invalidate message to other cores
  - **So the processor must 'lock' the snoopy bus for every multiprocessor `tas` operation**
    - Cannot let any other core do a write
    - But if it ends up reading a '1' (lock not available), this locking of the bus was wasted because the `tas` was read-only...

Test-and-Set and the Cache

- 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

Test-and-Set and the Cache

- 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**
    - Slowing down cache coherence traffic from other cores

Test-and-Set and the Cache

- 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**
    - Slowing down cache coherence traffic from other cores
- Can address that issue with a simple re-formulation of the wait operation: **test-and-test-and-set**
    - Tries to minimise the amount of costly test-and-set

Test-and-test-and-set

- How to implement `wait()` with test-and-test-and-set?

- In pseudo-code:

```
do {
  while(test(S) == 1);    // traditional ldr
} while (test-and-set(S));// tas

//
```

Test-and-test-and-set

- How to implement `wait()` with test-and-test-and-set?

- In pseudo-code:

```
do {
  while(test(S) == 1);    // traditional ldr
} while (test-and-set(S));// tas

//
```

- In assembly:

```
loop: ldr %r1, %r2   /* address of S in %r2 */
      cmp %r1, $1
      beq loop
      tas %r2
      bnz loop       /* branch if %r2 != 0 */
```

Test-and-test-and-set

- How to implement `wait()` with test-and-test-and-set?

- In pseudo-code:

```
do {
  while(test(S) == 1);    // traditional ldr
} while (test-and-set(S));// tas

//
```

- In assembly:

```
loop: ldr %r1, %r2   /* address of S in %r2 */
      cmp %r1, $1
      beq loop
      tas %r2
      bnz loop       /* branch if %r2 != 0 */
```

- Key idea:
  - Most of the time we busy wait with a standard `ldr`
  - **Only once S is seen to be free, a (costly) `tas` is made**

Other Synchronisation Primitives

- Other machine level atomic instructions:

Other Synchronisation Primitives

- Other machine level atomic instructions:
    - **fetch-and-add**: returns the value of a memory location and increments it

```
// in pseudo-code
fetch_and_add(addr, incr) {
    old_val = *addr;
    *addr += incr;
    return old_val;
}
```

Other Synchronisation Primitives

- Other machine level atomic instructions:
  - **fetch-and-add**: returns the value of a memory location and increments it
  - **compare-and-swap**: compares the value of a memory location with a value (in a register) and swap in another value (in a register) if they are equal

```
// in pseudo-code
compare_and_swap(addr, comp, new_val) {
    if(*addr != comp)
        return false;

    *addr = new_val;
    return true;
}
```

Other Synchronisation Primitives

- Other machine level atomic instructions:
  - **fetch-and-add**: returns the value of a memory location and increments it
  - **compare-and-swap**: compares 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 RMW with the need to lock the snoopy bus during their execution**

Other Synchronisation Primitives

- Other machine level atomic instructions:
  - **fetch-and-add**: returns the value of a memory location and increments it
  - **compare-and-swap**: compares 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 RMW with the need to lock the snoopy bus during their execution**
- Not really desirable with all CPU designs:
  - Doesn’t fit well with simple RISC pipelines, where RMW is really a CISC instruction requiring a memory load operation, a comparison, and possibly a store operation

Lock-Free Data Structures

- Atomic instructions can be used for other goals than implementing locks
- **Lock-Free data structures**: data structures that can be accessed concurrently without locks through the use of atomic instructions
  - Lists, stacks, etc.
- They are generally **hard to implement**:
  - Updating their state requires more than the single memory store operation done by RMW instructions
  - Hard to know when a member of the data structure can be freed on languages without garbage collectors (e.g. C/C++)
- Benefits: they can be **faster than lock-based data structures**

Lock-Free Data Structures

- Lock-free queue implementation examples:
  - In Java: https://github.com/olivierpierre/comp35112-devcontainer/tree/main/10-hardware-synchronisation/lock-free-queue-java
    - Not too hard because Java has a GC, still not entirely trivial
  - In C: https://github.com/olivierpierre/comp35112-devcontainer/tree/main/10-hardware-synchronisation/lock-free-queue-c
    - A bit convoluted/hacky
- More info: https://www.baeldung.com/lock-free-programming and "The Art of Multiprocessor Programming" chapters 10 and 11

Summary

- Synchronisation requires support from the hardware to ensure that critical code section are executed atomically
- Atomic read-modify-write instructions can be used but they are costly and hard to support on RISC CPUs
- Next lecture: how to address these issues by breaking an atomic RMW operation into two instructions working together: *load-linked* and *store-conditional*