Lecture: Coherence, Synchronization

- Topics: directory-based coherence, synchronization primitives (Sections 5.1-5.5)

Cache Coherence Protocols

- Directory-based: A single location (directory) keeps track of the sharing status of a block of memory

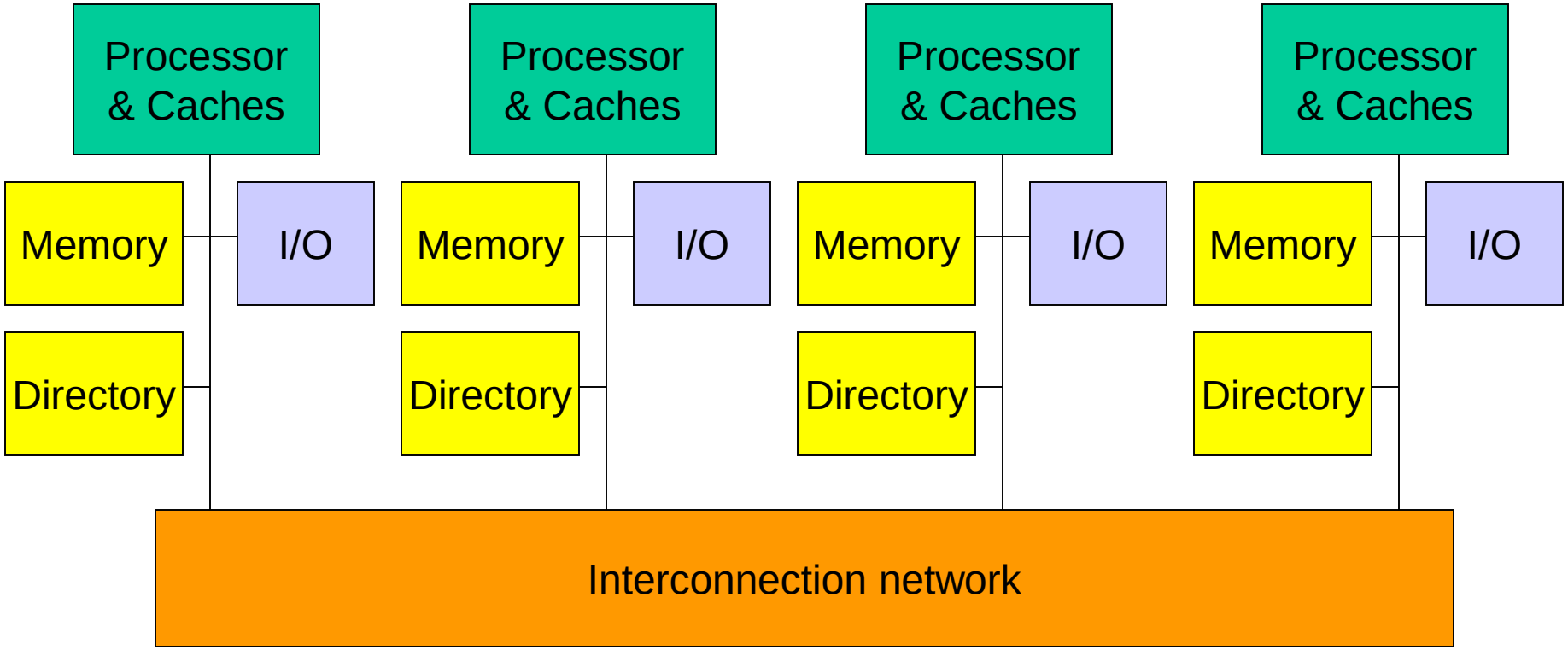
- Snooping: Every cache block is accompanied by the sharing status of that block – all cache controllers monitor the shared bus so they can update the sharing status of the block, if necessary

  - Write-invalidate: a processor gains exclusive access of a block before writing by invalidating all other copies
  - Write-update: when a processor writes, it updates other shared copies of that block

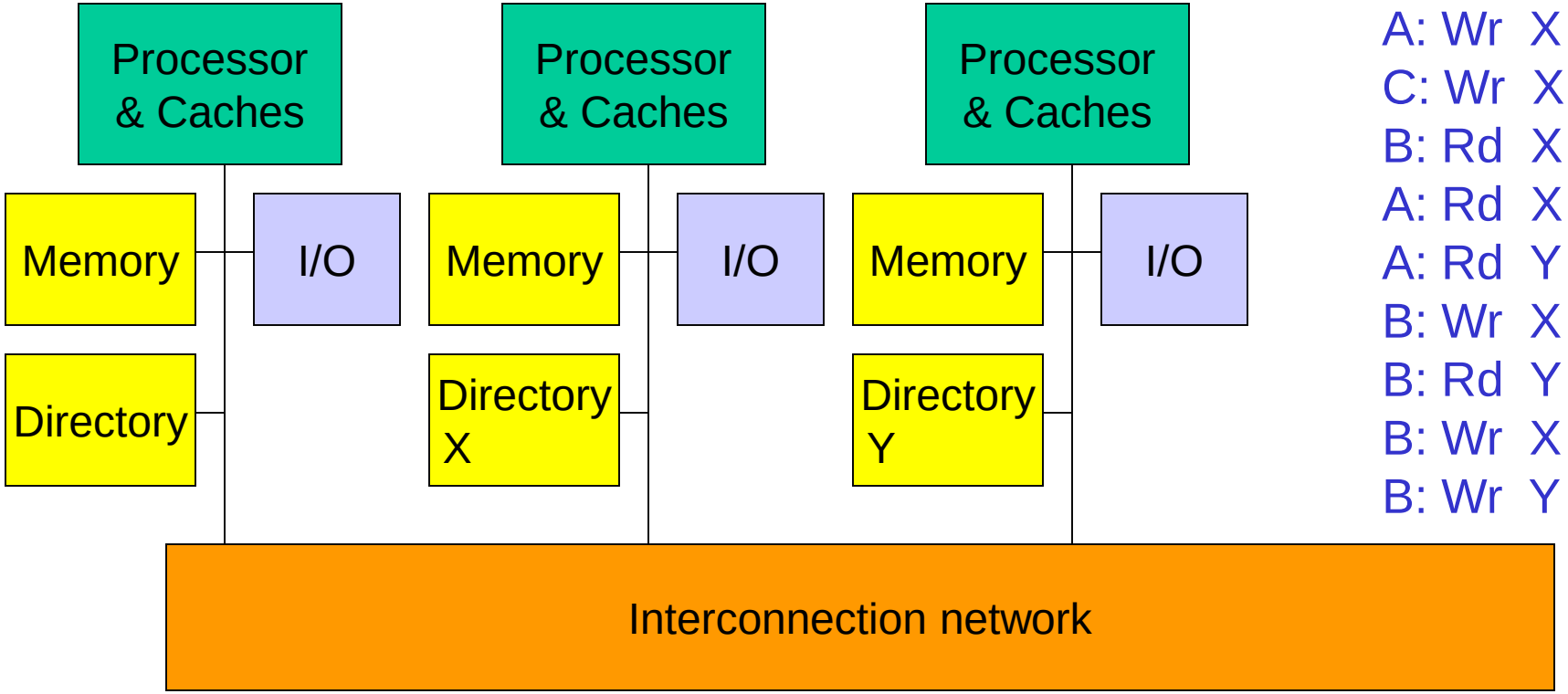
Directory-Based Cache Coherence

- The physical memory is distributed among all processors
- The directory is also distributed along with the corresponding memory
- The physical address is enough to determine the location of memory
- The (many) processing nodes are connected with a scalable interconnect (not a bus) – hence, messages are no longer broadcast, but routed from sender to receiver – since the processing nodes can no longer snoop, the directory keeps track of sharing state

Distributed Memory Multiprocessors



Directory-Based Example



A: Rd X
B: Rd X
C: Rd X
A: Wr X
A: Wr X
C: Wr X
B: Rd X
A: Rd X
A: Rd Y
B: Wr X
B: Rd Y
B: Wr X
B: Wr Y

Directory Example

	A	B	C	Dir	Comments
A: Rd X					
B: Rd X					
C: Rd X					
A: Wr X					
A: Wr X					
C: Wr X					
B: Rd X					
A: Rd X					
A: Rd Y					
B: Wr X					
B: Rd Y					
B: Wr X					
B: Wr Y					

Directory Example

	A	B	C	Dir	Comments
A: Rd X	S			S: A	Req to dir; data to A
B: Rd X	S	S		S: A, B	Req to dir; data to B
C: Rd X	S	S	S	S: A,B,C	Req to dir; data to C
A: Wr X	M	I	I	M: A	Req to dir;inv to B,C;dir recv ACKs;perms to A
A: Wr X	M	I	I	M: A	Cache hit
C: Wr X	I	I	M	M: C	Req to dir;fwd to A; sends data to dir; dir to C
B: Rd X	I	S	S	S: B, C	Req to dir;fwd to C;data to dir;dir to B; wrtbk
A: Rd X	S	S	S	S:A,B,C	Req to dir; data to A
A: Rd Y	S(Y)	S	S	X:S: A,B,C (Y:S:A)	Req to dir; data to A
B: Wr X	S(Y)	M	I	X:M:B	Req to dir; inv to A,C;dir recv ACK;perms to B
B: Rd Y	S(Y)	S(Y)	I	X: - Y:S:A,B	Req to dir; data to B; wrtbk of X
B: Wr X	S(Y)	M(X)	I	X:M:B Y:S:A,B	Req to dir; data to B
B: Wr Y	I	M(Y)	I	X: - Y:M:B	Req to dir;inv to A;dir recv ACK; perms and data to B;wrtbk of X

Cache Block States

- What are the different states a block of memory can have within the directory?

- Note that we need information for each cache so that invalidate messages can be sent

- The block state is also stored in the cache for efficiency

- The directory now serves as the arbitrator: if multiple write attempts happen simultaneously, the directory determines the ordering

Directory Actions

- If block is in uncached state:
  - Read miss: send data, make block shared
  - Write miss: send data, make block exclusive
- If block is in shared state:
  - Read miss: send data, add node to sharers list
  - Write miss: send data, invalidate sharers, make excl
- If block is in exclusive state:
  - Read miss: ask owner for data, write to memory, send data, make shared, add node to sharers list
  - Data write back: write to memory, make uncached
  - Write miss: ask owner for data, write to memory, send data, update identity of new owner, remain exclusive

Performance Improvements

- What determines performance on a multiprocessor:
  - What fraction of the program is parallelizable?
  - How does memory hierarchy performance change?

- New form of cache miss: coherence miss – such a miss would not have happened if another processor did not write to the same cache line

- False coherence miss: the second processor writes to a different word in the same cache line – this miss would not have happened if the line size equaled one word

Constructing Locks

- Applications have phases (consisting of many instructions) that must be executed atomically, without other parallel processes modifying the data
- A lock surrounding the data/code ensures that only one program can be in a critical section at a time
- The hardware must provide some basic primitives that allow us to construct locks with different properties
- Lock algorithms assume an underlying cache coherence mechanism – when a process updates a lock, other processes will eventually see the update

Synchronization

- The simplest hardware primitive that greatly facilitates synchronization implementations (locks, barriers, etc.) is an atomic read-modify-write

- Atomic exchange: swap contents of register and memory

- Special case of atomic exchange: test & set: transfer memory location into register and write 1 into memory

- lock:

```
lock:   t&s   register, location
        bnz   register, lock
        CS
        st    location, #0
```

Caching Locks

- Spin lock: to acquire a lock, a process may enter an infinite loop that keeps attempting a read-modify till it succeeds

- If the lock is in memory, there is heavy bus traffic → other processes make little forward progress

- Locks can be cached:
  - cache coherence ensures that a lock update is seen by other processors
  - the process that acquires the lock in exclusive state gets to update the lock first
  - spin on a local copy – the external bus sees little traffic

Coherence Traffic for a Lock

- If every process spins on an exchange, every exchange instruction will attempt a write → many invalidates and the locked value keeps changing ownership

- Hence, each process keeps reading the lock value – a read does not generate coherence traffic and every process spins on its locally cached copy

- When the lock owner releases the lock by writing a 0, other copies are invalidated, each spinning process generates a read miss, acquires a new copy, sees the 0, attempts an exchange (requires acquiring the block in exclusive state so the write can happen), first process to acquire the block in exclusive state acquires the lock, others keep spinning

Test-and-Test-and-Set

- lock:

```
lock:   test   register, location
        bnz    register, lock
        t&s    register, location
        bnz    register, lock
        CS
        st     location, #0
```

Load-Linked and Store Conditional

- LL-SC is an implementation of atomic read-modify-write with very high flexibility
- LL: read a value and update a table indicating you have read this address, then perform any amount of computation
- SC: attempt to store a result into the same memory location, the store will succeed only if the table indicates that no other process attempted a store since the local LL (success only if the operation was “effectively” atomic)
- SC implementations do not generate bus traffic if the SC fails – hence, more efficient than test&test&set

Spin Lock with Low Coherence Traffic

```
lockit:  LL      R2, 0(R1)    ; load linked, generates no coherence traffic
         BNEZ    R2, lockit   ; not available, keep spinning
         DADDUI R2, R0, #1 ; put value 1 in R2
         SC      R2, 0(R1)    ; store-conditional succeeds if no one
                              ; updated the lock since the last LL
         BEQZ    R2, lockit   ; confirm that SC succeeded, else keep trying
```

- If there are i processes waiting for the lock, how many bus transactions happen?

Spin Lock with Low Coherence Traffic

```
lockit:  LL      R2, 0(R1)    ; load linked, generates no coherence traffic
         BNEZ    R2, lockit   ; not available, keep spinning
         DADDUI R2, R0, #1 ; put value 1 in R2
         SC      R2, 0(R1)    ; store-conditional succeeds if no one
                              ; updated the lock since the last LL
         BEQZ    R2, lockit   ; confirm that SC succeeded, else keep trying
```

- If there are i processes waiting for the lock, how many bus transactions happen?

  1 write by the releaser + i read-miss requests + i responses + 1 write by acquirer + 0 (i-1 failed SCs) + i-1 read-miss requests + i-1 responses

  (The i/i-1 read misses can be reduced to 1)

Further Reducing Bandwidth Needs

- Ticket lock: every arriving process atomically picks up a ticket and increments the ticket counter (with an LL-SC), the process then keeps checking the now-serving variable to see if its turn has arrived, after finishing its turn it increments the now-serving variable

- Array-Based lock: instead of using a “now-serving” variable, use a “now-serving” array and each process waits on a different variable – fair, low latency, low bandwidth, high scalability, but higher storage

- Queueing locks: the directory controller keeps track of the order in which requests arrived – when the lock is available, it is passed to the next in line (only one process sees the invalidate and update)

Lock Vs. Optimistic Concurrency

```
lockit:   LL       R2, 0(R1)
          BNEZ     R2, lockit
          DADDUI R2, R0, #1
          SC       R2, 0(R1)
          BEQZ     R2, lockit
            Critical Section
          ST       0(R1), #0
```

LL-SC is being used to figure out if we were able to acquire the lock without anyone interfering – we then enter the critical section

```
tryagain: LL       R2, 0(R1)
          DADDUI R2, R2, R3
          SC       R2, 0(R1)
          BEQZ     R2, tryagain
```

If the critical section only involves one memory location, the critical section can be captured within the LL-SC – instead of spinning on the lock acquire, you may now be spinning trying to atomically execute the CS

Barriers

- Barriers are synchronization primitives that ensure that some processes do not outrun others – if a process reaches a barrier, it has to wait until every process reaches the barrier

- When a process reaches a barrier, it acquires a lock and increments a counter that tracks the number of processes that have reached the barrier – it then spins on a value that gets set by the last arriving process

- Must also make sure that every process leaves the spinning state before one of the processes reaches the next barrier

Barrier Implementation

```
LOCK(bar.lock);
if (bar.counter == 0)
  bar.flag = 0;
mycount = bar.counter++;
UNLOCK(bar.lock);
if (mycount == p) {
  bar.counter = 0;
  bar.flag = 1;
}
else
  while (bar.flag == 0)  { };
```

Sense-Reversing Barrier Implementation

```
local_sense = !(local_sense);
LOCK(bar.lock);
mycount = bar.counter++;
UNLOCK(bar.lock);
if (mycount == p) {
  bar.counter = 0;
  bar.flag = local_sense;
}
else {
  while (bar.flag != local_sense)  { };
}
```

Title

- Bullet