COSC4201

Distributed Shared Memory Architecture

Prof. Mokhtar Aboelaze

York University

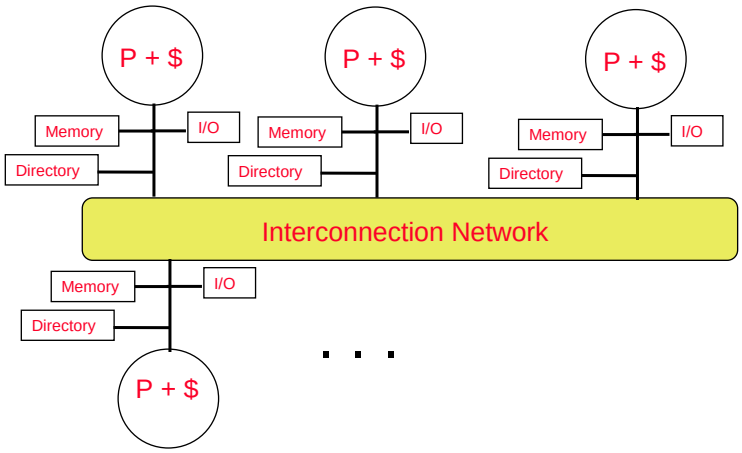
DSM

- Memory is distributed among the processors.
- An interconnection network is used to communicate
- Earlier attempts ignored cache coherence. Shared data are marked as uncachable.
- Still, shared data could be cached by S/W, but consistency should be maintained by the software.
- A software based approach must be conservative, every block that is *might* be shared, is considered shared.

DSM

- Also, for a small number of processors, a single bus is O.K. For large number, bus B.W. is not enough.
- For a snooping protocol, the assumption is every transaction is broadcast on the bus for every processor to hear.
- That generate a lot of traffic, and is not easily implemented in a non-bus architecture.

Directory-Based Cache Coherence Protocols



Directory Based Cache Coherence Protocols

° A block can be in one of three states

- **Shared**: One or more processor have the block cached. The value in the memory is up to date
- **Uncached**: No processor has a copy of the block
- **Exclusive** : Exactly one processor has a copy of the block, and it has written the block. The memory copy is out of date. The processor is called the owner of the block

Directory Based Cache Coherence Protocols

° We must track the state of every memory block, as well as the owner of any exclusive block, or the processors who have copies of a shared block

° A bit vector can be used for that.

° Write to a non exclusive block always result in a write miss.

° The interconnect is no more a single point of arbitration.

° Local node is the node where the request originates, the home node is the node where the memory location and the directory entry of an address reside, remote node is a node that have a copy of the block.

Directory Based Cache Coherence Protocols

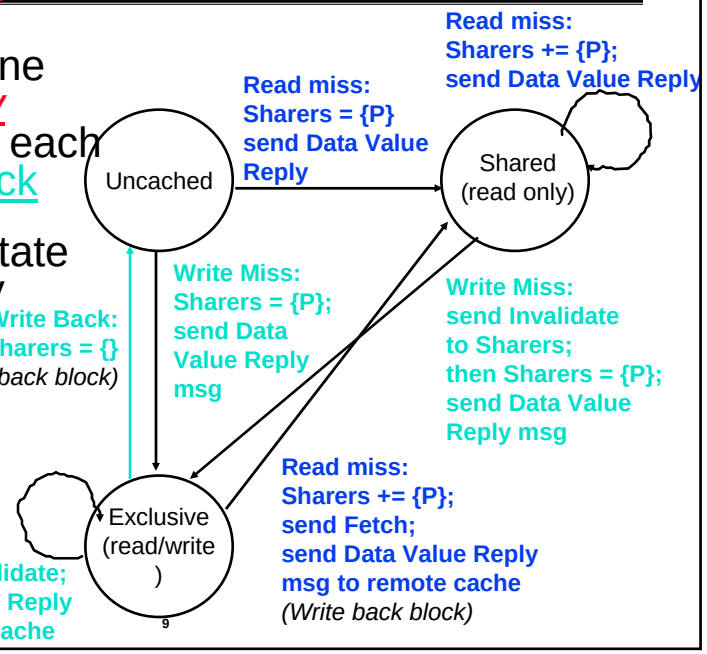
- A simple type of memory consistency is assumed.
- We assume that the messages are received and acted upon at the same order they are sent.
- That is a very difficult thing to do in practice (some times it is not possible to do at all).
- A single address space, the high order bits of the address can be used to determine the node number, the rest of the bits determine the offset in that memory.

Directory Protocol Messages

Message type	*Source*	*Destination*	*Msg Content*
Read miss	**Local cache**	**Home directory**	P, A
• *Processor P reads data at address A; make P a read share and request data*			
Write miss	**Local cache**	**Home directory**	P, A
• *Processor P has a write miss at address A; make P the exclusive owner and request data*			
Invalidate	Home directory	Remote caches	A
• *Invalidate a shared copy at address A*			
Fetch	**Home directory**	**Remote cache**	A
• *Fetch the block at address A and send it to its home directory; change the state of A in the remote cache to shared*			
Fetch/Invalidate	**Home directory**	**Remote cache**	A
• *Fetch the block at address A and send it to its home directory; invalidate the block in the cache*			
Data value reply	**Home directory**	**Local cache**	Data
• *Return a data value from the home memory (read miss response)*			
Data write back	**Remote cache**	**Home directory**	A, Data
• *Write back a data value for address A (invalidate response)*			

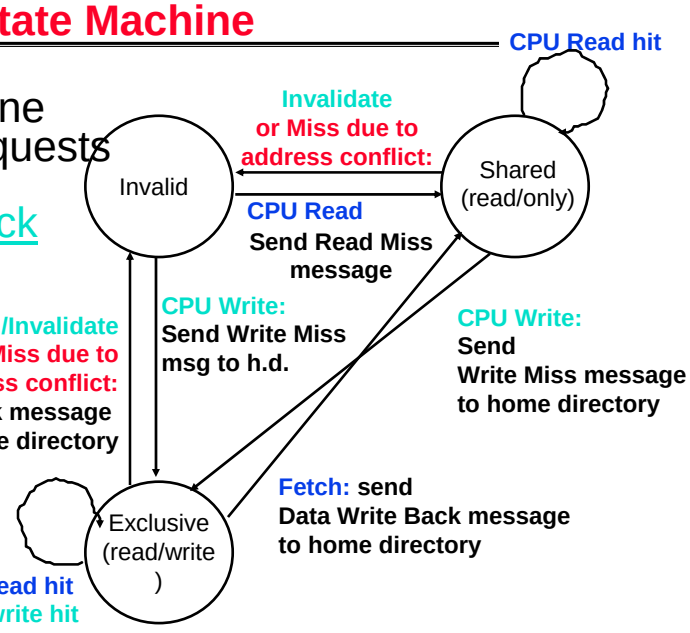
Directory State machine

- State machine for *Directory* requests for each memory block
- Uncached state if in memory



Cache State Machine

- State machine for *CPU* requests for each memory block
- Invalid state if in memory



Example

	Processor 1			Processor 2			Interconnect				Directory			Memory
	P1			P2			Bus				Directory			Memory
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	State	{Procs}	Value
P1: Write 10 to A1														
P1: Read A1														
P2: Read A1														
P2: Write 20 to A1														
P2: Write 40 to A2														

A1 and A2 map to the same cache block

Example

	Processor 1			Processor 2			Interconnect				Directory			Memory
	P1			P2			Bus				Directory			Memory
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	State	{Procs}	Value
P1: Write 10 to A1							WrMs	P1	A1		A1	Ex	{P1}	
	Excl.	A1	10				DaRp	P1	A1	0				
P1: Read A1														
P2: Read A1														
P2: Write 20 to A1														
P2: Write 40 to A2														

A1 and A2 map to the same cache block

Example

Processor 1 Processor 2 Interconnect Directory Memory

	P1			P2			Bus				Directory			Memory
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	State	{Procs}	Value
P1: Write 10 to A1							WrMs	P1	A1		A1	Ex	{P1}	
	Excl.	A1	10				DaRp	P1	A1	0				
P1: Read A1	Excl.	A1	10											
P2: Read A1														
P2: Write 20 to A1														
P2: Write 40 to A2														

A1 and A2 map to the same cache block

Example

Processor 1 Processor 2 Interconnect Directory Memory

	P1			P2			Bus				Directory			Memory
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	State	{Procs}	Value
P1: Write 10 to A1							WrMs	P1	A1		A1	Ex	{P1}	
	Excl.	A1	10				DaRp	P1	A1	0				
P1: Read A1	Excl.	A1	10											
P2: Read A1				Shar.	A1		RdMs	P2	A1					
	Shar.	A1	10				Ftch	P1	A1	10			A1	10
				Shar.	A1	10	DaRp	P2	A1	10	A1	Shar.	P1,P2}	10
P2: Write 20 to A1														10
														10
P2: Write 40 to A2														10

Write Back

A1 and A2 map to the same cache block

Example

	Processor 1			Processor 2			Interconnect				Directory			Memory
	P1			P2			Bus				Directory			Memory
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	State	{Procs}	Value
P1: Write 10 to A1							WrMs	P1	A1		A1	Ex	{P1}	
	Excl.	A1	10				DaRp	P1	A1	0				
P1: Read A1	Excl.	A1	10											
P2: Read A1				Shar.	A1		RdMs	P2	A1					
	Shar.	A1	10				Ftch	P1	A1	10			A1	10
				Shar.	A1	10	DaRp	P2	A1	10	A1	Shar.	P1,P2}	10
P2: Write 20 to A1				Excl.	A1	20	WrMs	P2	A1					10
	Inv.						Inval.	P1	A1		A1	Excl.	{P2}	10
P2: Write 40 to A2														10

A1 and A2 map to the same cache block

Example

	Processor 1			Processor 2			Interconnect				Directory			Memory
	P1			P2			Bus				Directory			Memory
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	State	{Procs}	Value
P1: Write 10 to A1							WrMs	P1	A1		A1	Ex	{P1}	
	Excl.	A1	10				DaRp	P1	A1	0				
P1: Read A1	Excl.	A1	10											
P2: Read A1				Shar.	A1		RdMs	P2	A1					
	Shar.	A1	10				Ftch	P1	A1	10			A1	10
				Shar.	A1	10	DaRp	P2	A1	10	A1	Shar.	P1,P2}	10
P2: Write 20 to A1				Excl.	A1	20	WrMs	P2	A1					10
	Inv.						Inval.	P1	A1		A1	Excl.	{P2}	10
P2: Write 40 to A2							WrMs	P2	A2		A2	Excl.	{P2}	0
							WrBk	P2	A1	20	A1	Unca.	{}	20
				Excl.	A2	40	DaRp	P2	A2	0	A2	Excl.	{P2}	0

A1 and A2 map to the same cache block

Synchronization

- Typically built with user-level routines that rely on hardware-supplied synchronization instructions.
- Hardware should be capable of supporting un-interruptible instruction or instruction sequence that can ***atomically*** retrieve and change a value.
- In a large scale multiprocessor (high contention) synchronization could be a bottleneck.
- Some hardware supported synchronization primitives can reduce contention and latency.

Basic Hardware Primitives

- The key ability to implement synchronization is a set of hardware primitives that can atomically read and modify a memory location.
- Generally users do not use these primitives, but rather system programmers use them to implement synchronization routine to be used by the user.
- A typical operation is an ***atomic exchange*** that can exchange a value in a register with a value in the memory, how can you use this to implement a lock?

Basic Hardware Primitives

- A pair of instruction could be used to implement synchronization.
- These two instructions are used in sequence (not aomically)
- Known as *load linked* or *load locked*, and *store conditional*.

Basic Hardware Primitives

- The load linked is LL R2,0(R1), reads the 0(R1) and stores it in R2.
- The hardware keep track of the address specified in LL and store it in a *link register*
- If an interrupt happened, or a write to the location specified in the link register, the link register is invalidated (cleared).
- Sore conditional checks to see if its address matches that in the link register, if yes, store and return 1, else do not store and return 0.

Basic Hardware Primitives

```
try:    mov     R3,R4           ; mov exchange value
        ll      R2,0(R1)        ; load linked
        sc      R3,0(R1)        ; store conditional
        beqz    R3,try          ; branch store fails (R3 = 0)
        mov     R4,R2           ; put load value in R4
```

Implementing fetch and increment using LL and SC

```
try:    ll      R2,0(R1)        ; load linked
        addi    R2,R2,#1        ; increment (OK if reg–reg)
        sc      R2,0(R1)        ; store conditional
        beqz    R2,try  ; branch store fails (R2 = 0)
```