QUIZ 6 SOLUTIONS

Problem M4.4: Implementing Directories

Problem M4.4.A

Overhead for a 4-processor system: 4 bits / 32 bytes = 4 / (32 * 8) = 1/64

Overhead for a 64-processor system: 64 bits / 32 bytes = 64 / (32 * 8) = 1/4

Problem M4.4.B

Sequence 1	**bit-vector scheme # of invalidate-requests**	**single-sharer scheme # of invalidate-requests**
Processor #0 reads **B**	0	0
Processor #1 reads **B**	**0**	**1**
Processor #0 reads **B**	**0**	**1**

For the bit-vector scheme: No invalidate-requests are sent.

For the single-sharer scheme:
1 invalidate-request is sent to P0 when P1 reads B.
1 invalidate-request is sent to P1 when P0 reads B the second time.

Sequence 2	**bit-vector scheme # of invalidate-requests**	**single-sharer scheme # of invalidate-requests**
Processor #0 reads **B**	0	0
Processor #1 reads **B**	**0**	**1**
Processor #2 writes **B**	**2**	**1**

For the bit-vector scheme:
1 invalidate-request is sent to each shared processor (P0 and P1) when P2 writes B.
-> 2 invalidate-requests are sent.

For the single-sharer scheme:
1 invalidate-request is sent to P0 when P1 reads B.
1 invalidate-request is sent to the only sharer (P1) when P2 writes B.

Problem M4.4.C

Sequence 1	global-bit scheme # of invalidate-requests
Processor #0 reads **B**	0
Processor #1 reads **B**	**0**
Processor #0 reads **B**	**0**

For the global-bit scheme: No invalidate-requests are sent.

Sequence 2	global-bit scheme # of invalidate-requests
Processor #0 reads **B**	0
Processor #1 reads **B**	**0**
Processor #2 writes **B**	**64**

For the global-bit scheme:
1 invalidate-request is sent to each of the 64 processors because the global bit is set when P2 writes B. -> 64 invalidate-requests are sent.

Note: If the protocol is optimized, no invalidate-request would be sent to P2 and the number of invalidate-requests would be 63 instead of 64.

Problem M4.4.D

In Tr(all), all is the set containing all the processors.

No.	Current State	Message Received	Next State	Action
1	R(dir) & (dir = ε)	ShReq	R({k})	ShRep->k
2	R(dir) & (dir = ε)	ExReq	W(k)	ExRep->k
3	R(dir) & (dir ≠ ε)	ShReq	**R(all)**	**ShRep->k**
4	R(all)	ShReq	**R(all)**	**ShRep->k**
5	R(dir) & (dir ≠ ε)	ExReq	**Tr(dir)**	**InvReq->dir** *(dir has only one entry.)*
6	R(all)	ExReq	**Tr(all)**	**InvReq->all**
7	W(id)	ShReq	Tw(id)	WbReq->id
8	Tr(dir) & (id ∈ dir)	InvRep	Tr(dir - {id})	nothing
9	Tr(dir) & (dir = {k})	InvRep	W(*j*)	ExRep->j
10	Tw(id)	FlushRep	**R(ε)**	**Data->memory**

Table M4.4-1: Partial List of Home Directory State Transitions

Problem M4.8: Snoopy Cache Coherent Shared Memory [? Hours]

Problem M4.8.A

Fill out the state transition table for the new COS state:

initial state	other cached	ops	actions by this cache	final state
COS	yes	none	none	COS
		CPU read	**none**	**COS**
		CPU write	**CI**	**OE**
		replace	**none**	**I**
		CR	**CCI**	**COS**
		CRI	**CCI**	**I**
		CI	**none**	**I**
		WR	**Impossible**	
		Or:	**none**	**COS**
		CWI	**none**	**I**

Note that WR is not necessary during replace because the line is clean.
Also, an incoming WR operations is Impossible because other caches can only have the block in the CS state, but (none, COS) was also accepted as a correct answer.

Problem M4.8.B

cache transaction	source for data	state for data block B			
		cache 1	cache 2	cache 3	cache 4
0. *initial state*	—	I	I	I	I
1. cache 1 reads data block B	**memory**	**CE**	I	I	I
2. cache 2 reads data block B	**CCI**	**COS**	**CS**	I	I
3. cache 3 reads data block B	**CCI**	**COS**	**CS**	**CS**	I
4. cache 1 replaces block B	**-**	**I**	**CS**	**CS**	**I**
5.cache 4 reads data block B	**memory**	**I**	**CS**	**CS**	**CS**

Problem M4.8.C

When the CPU does a write, it can change a cache block from CE to OE with no bus operation, but to transition from COS to OE it must first broadcast a CI on the bus to invalidate any shared (CS) copies of the block.

Problem M4.11: Relaxed Memory Models [? Hours]

We will study the interaction between two processes on different processors on such a system:

P1	P2
`P1.1: LW R2, 0(R8)`	`P2.1: LW R4, 0(R9)`
`P1.2: SW R2, 0(R9)`	`P2.2: SW R5, 0(R8)`
`P1.3: LW R3, 0(R8)`	`P2.3: SW R4, 0(R8)`

Problem M4.11.A

memory	contents
M[R8]	7
M[R9]	6

[✓ Yes] **No**

P1.1 P2.1 P1.2 P1.3 P2.2 P2.3

Problem M4.11.B

memory	Contents
M[R8]	6
M[R9]	7

Yes **[✓ No]**

The result would require that the memory contents don't change. Since each thread reads a data value and writes it to another address, this simply impossible here.

Problem M4.11.C

Is it possible for M[R8] to hold 0?

Yes **[✓ No]**

The only way that M[R8] could end up with 0 is if P2.3 is completed before P2.1 and P2.2. This violates Weak Ordering, so it is not possible.

Now consider the same program, but with two **`MEMBAR`** instructions.

P1	P2
P1.1: LW R2, 0(R8)	P2.1: LW R4, 0(R9)
P1.2: SW R2, 0(R9)	MEMBAR_{RW}
MEMBAR_{WR}	P2.2: SW R5, 0(R8)
P1.3: LW R3, 0(R8)	P2.3: SW R4, 0(R8)

We want to compare execution of the two programs on our system.

Problem M4.11.D

If both M[R8] and M[R9] contain 6, is it possible for R3 to hold 8?

Without **MEMBAR** instructions? [✓ **Yes**] **No**

With **MEMBAR** instructions? [✓ **Yes**] **No**

Following sequence works with and without MEMBAR instructions:
P1.1 -> P1.2 -> P2.1 -> P2.2 -> P1.3 -> P2.3

Problem M4.11.E

If both M[R8] and M[R9] contain 7, is it possible for R3 to hold 6?

Without **MEMBAR** instructions? **Yes** [✓ **No**]

With **MEMBAR** instructions? **Yes** [✓ **No**]

If M[R8] and M[R9] are to end up with 7, we have to execute P2.3 before we execute P1.1 Since P1.3 has to come after P1.1 (Weak Ordering), R3, has to end up with 7 not 6.

Problem M4.11.F

Is it possible for both M[R8] and M[R9] to hold 8?

Without **`MEMBAR`** instructions? [✓ **Yes**] **No**

P2.2 P1.1 P1.2 P2.1 P2.3 P1.3

With **`MEMBAR`** instructions? **Yes** [✓ **No**]

The sequence above violates the MEMBAR in P2—P2.2 executes before P2.1. That is the only way to get 8 into both memory locations, thus the result is impossible with MEMBARs insterted.