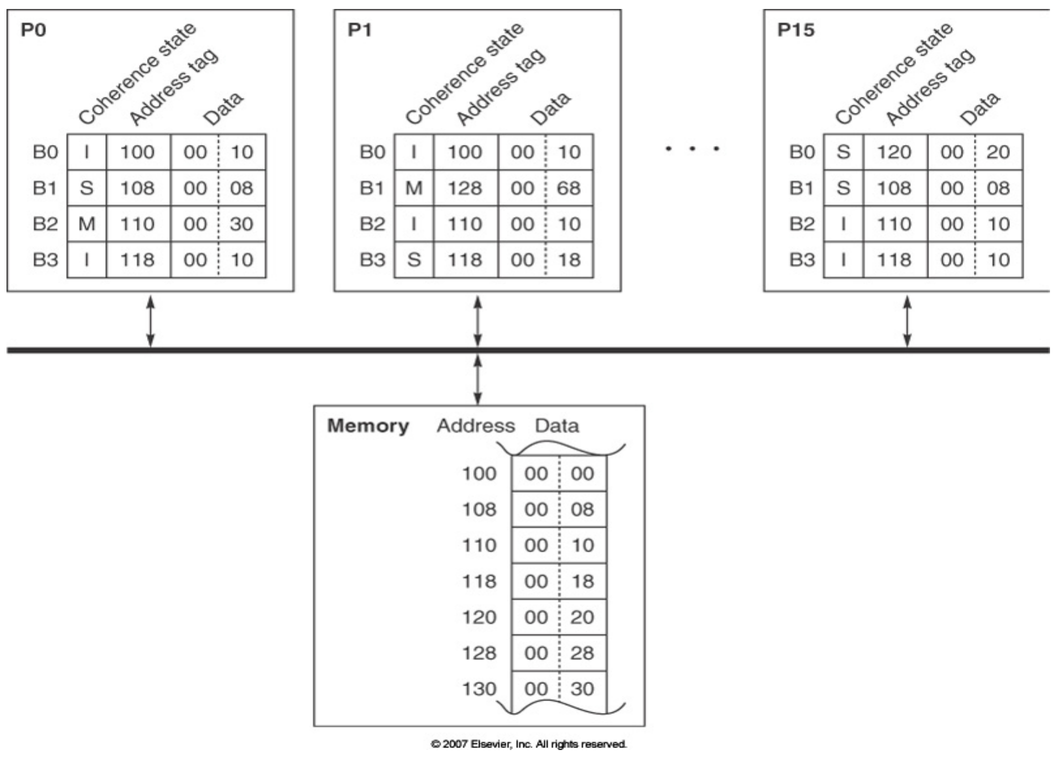
CS433 Homework 6

Assigned on 11/28/2017
Due in class on 12/12/2017

Instructions:
1. Please write your name and NetID clearly on the first page.
2. Refer to the course fact sheet for policies on collaboration.
3. Due **IN CLASS** on 12/12/2017.

Problem 1 [15 points]

This problem considers the simple MSI, bus-based snooping protocol for cache coherence discussed in class. There are several processor/cache nodes connected to the bus, along with main memory. Each processor has a private L1 cache that is direct mapped and contains 4 blocks of two words each. The initial state of main memory and each cache block is shown in the figure below. To simplify the figure, the cache address tag contains the full address and each word shows only two hex characters (the rest of the data word is all zeroes, not shown). The lower addressed word of the block is on the right; i.e., for block B0 of cache P0, the data for address x100 is on the right, and the data for x104 is on the left. The coherence states are denoted M, S, and I for Modified, Shared, and Invalid**,** respectively. Addresses and data are represented in hexadecimal.



P0

	Coherence state	Address tag	Data	
B0	I	100	00	10
B1	S	108	00	08
B2	M	110	00	30
B3	I	118	00	10

P1

	Coherence state	Address tag	Data	
B0	I	100	00	10
B1	M	128	00	68
B2	I	110	00	10
B3	S	118	00	18

. . .

P15

	Coherence state	Address tag	Data	
B0	S	120	00	20
B1	S	108	00	08
B2	I	110	00	10
B3	I	118	00	10

Memory

Address	Data	
100	00	00
108	00	08
110	00	10
118	00	18
120	00	20
128	00	28
130	00	30

List the cache block states that change for each of the actions below. **Treat each as an independent action applied to the initial state shown, do not accumulate changes from one part to the next**. Simply list the blocks that change state in any cache and memory for each action. List your answer in the form **P0.B0: (I, 100, 00, 10)** to mean processor cache P0's Block B0 is Invalid, the tag holds 100, the data words are 00 and 10; for memory use **M:100 (00,00)**. The action *write addr←data* means write data word *data* to address *addr*. When a block changes to Invalid, assume only the state bit changes, so your answer should include the tag and data field's actual (current) values.

(1) P15: read 118
P15.B3: (S, 118, 00, 18) (1 point)

(2) P15: write 100 ←48
P15.B0: (M, 100, 00, 48) (1 point)

(3) P15: write 118 ←80
P15.B3: (M, 118, 00, 80) (1 point)
P1.B3: (I, 118, 00, 18) (1 point)

(4) P15: write 108 ←80
P15.B1: (M, 108, 00, 80) (1 point)
P0.B1: (I, 108, 00, 08) (1 point)

(5) P15: read 110
P0.B2: (S, 110, 00, 30) (1 point)
P15.B2: (S, 110, 00, 30) (1 point)
M: 110 (00, 30) (1 point)

(6) P15: read 128
P1.B1: (S, 128, 00, 68) (1 point)
P15.B1: (S, 128, 00, 68) (1 point)
M: 128 (00, 68) (1 point)

(7) P15: write 110 ←40
P0.B2: (I, 110, 00, 30) (1 point)
P15.B2: (M, 110, 00, 40) (1 point)
M: 110 (00, 30) (1 point)

Problem 2 [21 points]

Based on the MSI protocol, the MESI protocol (also known as Illinois protocol) adds a fourth state, Exclusive. For a cache line to be held in Exclusive state, the following conditions must hold:

- This is the only processor with a copy of the cache block
  (i.e. in all other caches, this line is Invalid)
- The cache line in this cache is clean (has not been written)

Assume that after a cache performs a transaction on a bus, there is a mechanism for the cache to know whether other caches have a copy of the requested block or not at that time. This enables the cache to determine whether to transition to exclusive state.

Also assume that on a request for a line on a bus, memory always attempts to service the line unless a cache specifically aborts that attempt (because the cache is in a state that requires it to service the request).

Suppose a two-processor system uses the MESI protocol for cache coherence. Suppose in part of a program, the two processors issue the following stream of memory operations to a single memory address in the order and interleaving in the table below. There are no other accesses in between. Assume that the memory address contains value 0 initially. Fill out the state of the caches after each operation in the table below, and what bus request the processor broadcasts when it performs the operation, if any. Also fill out the data value in the caches and at memory at the end of each operation. *Write X* below means a write with value *X* to the considered address.

Operations		**Bus Request**	**P1 Cache**		**P2 Cache**		**Memory**
P1	**P2**		**State**	**Data**	**State**	**Data**	
		-	I	17	I	23	0
Read		Read Miss	E	0	I	23	0
Read		-	E	0	I	23	0
Write 3		-	M	3	I	23	0
	Read	Read Miss	S	3	S	3	3
	Read	-	S	3	S	3	3
	Write 4	Invalidate	I	3	M	4	3
	Write 5	-	I	3	M	5	3

Problem 3 [6 points]

compare-and-swap(R1,R2,L) is an atomic synchronization primitive which atomically compares the value in memory location L with R1, and if and only if they are equal, exchanges the values in R2 and L. *compare-and-swap(R1,R2,L)* can be used to efficiently emulate many other primitives.

Part A [2 points]

Implement an atomic *test-and-set* on memory address L in assembly using *compare-and-swap(R1,R2,L)* as the only atomic primitive. Let L = 1 when the lock is taken and L = 0 when it is free, and these will be the only values present at L. You may use any registers you like.
(HINT: If the location is in a certain state, you do not need to do anything.)

```
R1 <- 0
R2 <- 1
compare-and-swap(R1,R2,L)
```

Part B [2 points]

Implement the *test-and-test-and-set* semantics on memory address *L* in assembly using *compare-and-swap* as the only atomic primitive. Let *L*=1 when the lock is taken, and *L*=0 when it is free. You can use any registers you like as well as ordinary loads and stores. Include any instructions needed to ensure that the operation eventually completes successfully, as if you are actually trying to acquire a lock.

```
        R2 <- 1
LOCK:   R1 <- LOAD(L)
        BNEZ R1, LOCK
        compare-and-swap(R1,R2,L)
        BNEZ R2, LOCK
```

Part C [2 points]

Use compare-and-swap to implement an atomic *fetch-and-increment(R1,L)* in assembly, which atomically copies the old value in L to R1 and then increments the value in L by 1. Again, you can use any registers you like well as ordinary loads and stores. Include any instructions needed to ensure that the operation eventually completes successfully; i.e., the increment must be guaranteed to occur atomically.

```
FINC: R1 <- LOAD(L)
      R2 <- R1 + 1
      compare-and-swap(R1,R2,L)
      BNEQ R1, R2, FINC
```

Problem 4 [10 points]

We want to calculate the reliability of an I/O subsystem with the following components and MTTF (mean time till failure):

- 4 disks, each 120 GB and rated at 500,000-hour MTTF
- 1 IDE controller, 10,000,000-hour MTTF
- 1 power supply, 800,000-hour MTTF
- 4 IDE cables, 2,000,000-hour MTTF

Assume component lifetimes are exponentially distributed and the failures of the different components are independent.

Part A [4 points]

What is the MTTF of the entire I/O subsystem?

(Failure rate) = 4 / 500,000 + 1 / 10,000,000 + 1 / 800,000 + 4 / 2,000,000
= 0.00001135 per hour
MTTF = 1 / (Failure rate) = 88,105 hours
Grading: 2 points for failure rate. 2 points for MTTF.

Part B [3 points]

We want more space and performance and decides to double the number of hard disks and put all the disks under RAID 0 (no redundancy). What is the MTTF for the new I/O subsystem? Note, we will naturally also be adding 4 IDE cables for the 4 new hard disks too.

(Failure rate) = 8 / 500,000 + 1 / 10,000,000 + 1 / 800,000 + 8 / 2,000,000
= 0.00002135 per hour
MTTF = 1 / (Failure rate) = 46,838 hours
Grading: 2 points for failure rate; 1 point for MTTF

Part C [3 points]

Now we are concerned with the reliability of system and decide to use RAID3 on 6 hard disks. Assuming a working hot swap drive is available at all times, and it takes 1 hour to reconstruct the data onto this new replacement drive. What is the mean time till data loss? Assume the error we are concerned with is when a drive fails and during the reconstruction, when the RAID system is vulnerable to data loss, another of the drives fails.

(MTTF with 6 disks) = 500,000 / 6 = 83,333 hours
(MTTF with 5 disks) = 500,000 / 5 = 100,000 hours
We will have to perform a hot swap once every 83,333 hours on average, during which, the probability of another failure is 1/100,000.
(Failure rate during hot swap) = 1 / 100,000
Thus, MTTF = 83,333 * (1 / (Failure rate during hot swap)) = 8,333,300,000 hours
Grading: 1 points for 6 disk failure rate; 1 points for 5 disk failure rate; 1 points for MTTF

Problem 5 [6 points]

Consider the following hard drive:

Seek Time	8 ms
Rotation Speed	7200 rpm
Transfer Rate	80 MB/s
Controller Overhead	0.1 ms
Sector Size	512 Bytes

We have a 1 MB binary file stored on a single track. The file contains sorted key/value pairs. Each pair consists of two integers. The first integer is the key and the second is its corresponding value. Integers are 4 bytes long. Assuming the integer key we are searching for is contained within the file, calculate the average time needed to find its corresponding value using sequential search. Assume the only significant factor in the search time is the time to transfer sectors from disk (once transferred to memory, the time needed for to process each record is negligible). Assume we can process the file as we read. On average, how long does it take to find a key/value pair on the hard disk? [6 points]

On average, we will have to sequentially read through half the file before we find the key/value pair. This means we will have to read 512 KB.

Total time
= seek time + rotation delay + transfer time + overhead
= 8 ms + 0.5 rotation / (7200 rotations / 60000 ms)
+ 0.5 MB / (80 MB / 1000 ms) + 0.1 ms
= 8 ms + 4.17 ms + 6.25 ms + 0.1 ms = 18.52 ms

Grading: 1 point for each part (e.g., seek time); 2 point for correct answer

Problem 6 (*for graduate students only*) [8 points]

Considering the disk storage system from problem 5, what would be the worst case time needed if we used binary search? Assume that the file is stored on a single track and file and track sizes are equal.

The file holds 1MB × 1024 × 1024/8 = 131072 key/value pairs. If we used the binary search, it would take on average, $\log_2 131072 = 17$ accesses. However, the last 64 key/value pairs are all within the same 512 Bytes sector. So the last $\log_2 64 = 6$ accesses are all within the same sector, so effectively only one disk read is necessary for these last 6 accesses. That means there are a total of 12 reads.

Since we will use binary search, in the first step we locate the middle sector (1/2 rotation). Depending on the target key that we are looking for, we eliminate half of the sectors. Next, we need to do ¼ rotation again to find the middle sector among the remaining sectors. At each step, we will do half rotation of the previous step eliminating half of the remaining sectors to bootstrap the sector containing the target key.

Therefore, total rotations = 0.5+0.25+0.125+…(0.5)^12 = very close to 1.

First, we locate the track at the beginning only once (seek). Next, we do rotations and read 12 sectors one by one.

Total time
= seek time + one full rotation + transfer time of 12 sectors + overhead
= 8 ms + 1 / (7200 / 60,000) + (12 * 512) / (80 * 2^10) + 1 ms
= 8 ms + 8.33 ms + 0.075 ms + 0.1 ms = 16.5 ms

Grading: 4 points for correct number of sectors to read; 2 points for correct rotation; 2 points for correct answer