CS 433: Computer Architecture – Fall 2022
Homework 7
Total Points: Undergraduates (29 points), Graduates (42 points)
Undergraduate students should only solve the first 4 problems.
Graduate students should solve all problems.
Due Date: December 1st, 2022 at 10:00 pm CT
(See course information slides for more details)

Directions:

- **All students must write and sign the following statement at the end of their homework submission. "I have read the honor code for this class in the course information handout and have done this homework in conformance with that code. I understand fully the penalty for violating the honor code policies for this class." No credit will be given for a submission that does not contain this signed statement.**
- **On top of the first page of your homework solution, please write your name and NETID, your partner's name and NETID, and whether you are an undergrad or grad student.**
- **Please show all work that you used to arrive at your answer. Answers without justification will not receive credit. Errors in numerical calculations will not be penalized. Cascading errors will usually be penalized only once.**
- **See course information slides for more details.**

Problem 1 [13 points]
This problem concerns MOESI, an invalidation based snooping cache coherence protocol, for bus-based shared-memory multiprocessors with a single level of cache per processor. The MOESI protocol has five states. A block starting at address *Addr* can be in one of the following states in cache C:

- *Modified:* The block is present only in cache C and the data in the cache is dirty or modified (i.e., it reflects a more recent version than the copy in memory).

- *Owned:* The block is present in cache C and may also be present in other caches. The memory may not have an up-to-date copy of this block. The block is said to be owned by cache C and C must service the requests of other caches to this block since memory may not have an up-to-date copy.

- *Exclusive:* The block is present in a single cache (C) but is clean (i.e., memory has an up-to-date copy of the block).

- *Shared:* The block is present in cache C and possibly present in other caches.

- *Invalid:* The block is not valid in cache C (space for the block may or may not be currently allocated in this cache).

If the cache has a block in Owned state, then it services any requests to that block from other processors. Assume that the memory does NOT update its copy even if the request is a read by some other cache and the Owner cache has put the block on the bus. Hence, the owner cache remains in Owned state and continues to service other requests, until the block is replaced from its cache. Also assume that the only way to reach the Owned state is from the Modified or Exclusive state, when some other cache issues a read request for that block.

If a cache C has a block in exclusive or modified state, then it is responsible for servicing any requests to that block from other processors. If the request is a read, then the cache C transitions to the Owned state, and memory does NOT update its copy.

On replacement of a block in Owned or Modified state, the block is sent to memory, and memory resumes responsibility for servicing subsequent requests to that block. Replacement of a block in Exclusive state is similar, except that the block need not be sent to memory (since memory already has a copy).

Assume that after a cache performs a transaction on the bus, there is a mechanism for it 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.

Part A [2 points]

Consider a Block B in Owned state in the cache of processor P. Can B be in a non-invalid state in any other processor's cache? If yes, then what are the possible (non-invalid) states in which B could be in any of the other caches? If no, then explain why not.

Part B [6 points]

This part concerns the response of the cache of processor *i* to bus transactions initiated by the cache of processor *j* for a block that starts at address B (referred to as block B below). Fill out the following rows for the state transition table for the cache of processor *i*, showing the next state for block B in the cache and any action taken by the cache. Each entry should be filled out as: *Next State/Action* (e.g., S/Send block to memory) where

Next State = **M, O, E, S, or I**
Action = **Send block to memory, Send block to cache, Send block to cache and memory, or No action**
Note: If an entry is not possible (i.e., the system cannot be in such a state), write "Not Possible".

Current state of block B in cache of processor *i*	Read of block B by cache of processor *j*	Invalidate of block B by cache of processor *j* (with no read request for the block)	Read + Invalidate of block B by cache of processor *j*
M			
O			

Part C [5 points]

Consider the following sequence of operations by two processors for a block that starts at address B. Determine the state of that block in the caches of both the processors after each operation in the sequence for the MOESI protocol. Both caches are initially empty and all lines are in the I state. The table below is provided to help organize your answer.

No	Operation	MOESI	
		P1	P2
1	P1 reads B		I
2	P1 writes B		I
3	P2 writes B		
4	P1 reads B		
5	P1 writes B		
6	P2 reads B		

Problem 2 [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.

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.

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 as 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.

Problem 3 [4 points]

For this problem, assume sequential consistency. Consider the following code fragments executed on two processors:

Initially P = Q = R = S = 0

P1	P2
P = 5	L = Q
Q = 12	M = P
R = 8	N = S
S = 6	O = R

Part A [2 points]

The processors execute instructions independent of each other. Thus, the order of execution of instructions cannot be determined *a priori* if no constraint is placed on the execution of the instructions. What synchronization is required to ensure that all of P1's instructions are to be executed before any of P2's instructions, as given above? Ensure your solution makes clear the constituent memory operations used for the synchronization; i.e., don't insert just a call to a function or library without the code for that function/library. Unnecessarily inefficient solutions will not get full credit.

Part B [2 points]

Suppose we don't care about the relative order of execution of the sets of instructions, but we want atomicity in execution; i.e., we want all instructions of one processor to complete before any instructions are executed in the other processor. What synchronization is required to ensure this?

Problem 4 [6 points]

This problem depends on your solution to part A of problem 3. You will get credit for this problem only if part A of problem 3 is correct.

Part A [3 points]

On a sequentially consistent system, what values can L, M, N, and O have at the end of execution for the code in your solution of problem 3, part A? You must explain your answer for full credit.

Part B [3 points]

On a system that is not known to be sequentially consistent, what possible values can variables L, M, N, and O have at the end of execution, for the code in your solution of problem 3 part A? You must explain your answer for full credit.

NOTE: ONLY GRADUATE STUDENTS SHOULD SOLVE THIS PROBLEM

Problem 5 [13 points]

In the discussion of the barrier semantics in the lecture, a common parallel programming pattern was described where all processors produce data during one phase of a computation, and that data is consumed by many (or all) processors during the next phase. The "barrier" is used as synchronization so that the consumers of the data in phase n+1 are guaranteed to see the data that was produced in phase n.

An example program with such a pattern is molecular dynamics simulation where positions and forces for the molecules (or particles) are allocated in shared memory.. The outermost loop iterates over discrete timesteps covering the period of time covered by the simulation. In each timestep, we need to do two things:

(1) For each particle (or molecule) in the simulated system, we compute the force on the particle exerted by every other particle in the system. The force between two particles is a function of the distance between the two particles (i.e., a function of the positions of the two particles, as computed during the previous time step - see below). The total force on a given particle is the sum of the forces generated by all particles within the cutoff distance of the given particle.

(2) After the force on a particle is computed, its new position is computed using the force computed above and its old position computed in the previous timestep. The new positions are then used as input for the next timestep.

Part A [5 points]

How would you parallelize the computation for each timestep? Specifically, indicate how the computation is divided among N processors, which shared-memory variables are read and written by a given processor in different phases of the program, and where does what synchronization appear in the program and for what purpose. You can describe your algorithm in words and do not have to give code.

Part B [8 points]

Consider a shared-memory multiprocessor system where each processor has a private L1 cache and all processors and memory are connected by a bus. Suppose the L1 caches are magically of infinite size. Assume an MSI cache coherence protocol. Consider the 20th timestep in the simulation. What will be the state of all the shared variables in a processor's cache at the end of each synchronization event in this timestep? Which accesses will result in bus traffic in this timestep?