CS 433 Final Exam: May 14, 2025

Professor Sarita Adve

Time: 3 Hours

Please print your Name and NetID and circle your course section below.

Name:	Instructor	
NetID:		
Section:	S3 (Undergraduate)	S4 (Graduate)

Instructions

1. **Please write your Name and NetID on the top right corner of all pages in this exam.**
2. No books, papers, notes, or any other typed or written materials are allowed. No calculators or other electronic materials are allowed.
3. Please do not turn in loose scrap paper. Limit your answers to the space provided if possible. If this is not possible, please write on the back of the same sheet. You may use the back of each sheet for scratch work.
4. *In all cases, show your work. No credit will be given if there is no indication of how the answer was derived. Partial credit will be given even if your final solution is incorrect, provided you show the intermediate steps in reaching the final solution.*
5. If you believe a problem is incorrectly or incompletely specified, make a reasonable assumption and solve the problem. The assumption should not result in a trivial solution. In all cases, clearly state any assumptions that you make in your answers.
6. This exam has **6 problems** and **18 pages** (including this one). All students should solve problems 1 through 6B. **Only graduate students should solve problem 6C.** Please budget your time appropriately. Good luck!

Problem		**1**	**2**	**3**	**4**	**5**	**6**	**Total**
Points	**S3**	15	11	12	10	4	14	66
	S4	15	11	12	10	4	22	74
Score								

Name:
NetID:

Problem 1 [15 points]

Consider the following code:

```
double A[256][256];
double x = 0.0;

for (int i = 0; i < 64; i++) {
     for (int j = 0; j < 64; j++) {
          x += A[2*i][2*j];
     }
}
```

Assume the following:

- Array A contains double words (i.e., each element is 2 words long).
- Only accesses to array locations generate loads to the data cache. The rest of the variables are allocated in registers.
- The array A is stored in row-major order.
- The program is running on a machine with an L1 data cache that has 32 words per cache line.
- Memory is word addressable.
- Assume the array A starts at a cache line boundary.
- Assume an infinite data cache that is initially empty.

Part A [2 points]

How many L1 data cache misses occur for the above code? You must explain how you derived this number to receive any credit.

Solution:

Each cache line holds 16 elements of the matrix. Each inner loop invocation therefore results in 8 cache misses (since it must bring in 128 contiguous elements). There are a total of 64 invocations of the inner loop. Therefore there are a total of 512 L1 misses.

Grading:

2 total points for correct answer of 512 L1 misses.
In case of incorrect answer, partial credit of 1 point for identifying the correct number of misses in each inner loop invocation (8) or, equivalently, 1 cache miss per 8 accesses, or a similar partial correct computation. No points for an answer without an explanation.

Part B [2 points]

Suppose our machine has a data prefetch instruction with the format prefetch(array[i][j]). This prefetches the entire cache line containing the word array[i][j] into the data cache. Assume it takes one cycle for the processor to execute this instruction and send it to the data cache. The processor can then execute subsequent instructions.

Consider inserting prefetch instructions for the inner loop of the above code; i.e., ignore the outer loop for this part. Explain why we may need to unroll the inner loop to insert prefetches. What is the minimum number of times you would need to unroll the loop for this purpose?

Solution:

If we attempt to insert prefetch instructions without unrolling the inner loop, we would be prefetching superfluously since each prefetch brings in an entire cache line. To take advantage of prefetching, we would like every iteration of the loop to access as much data as possible in the prefetched cache line. We are told that each cache line contains 32 words (i.e., 16 double words). However, the program uses a strided access pattern where only the even array entries are used, which translates to only half of a cache line being used.
Thus, we should unroll the loop 16/2 = **8 times**.

Grading:

1 point for the correct answer of 8 unrolls.
1 point for explanation.
No points for an answer without an explanation.

Part C [4 points]

Modify the original program by unrolling the inner loop the number of times identified in Part B and insert the minimum number of software prefetches to minimize execution time. The technique to insert prefetches is analogous to software pipelining. You do not need to worry about startup and cleanup code (i.e., epilog and prolog) and do not introduce any new loops. Assume that two iterations of your unrolled inner loop are sufficient to hide the full latency of an L1 cache miss (i.e., if you suffer a cache miss on a load in unrolled loop iteration i, the value for that load will be available in unrolled loop iteration i+2). Again, for this part, you may ignore the outer loop.

Note that if you obtained an incorrect answer for Part B that results in a trivial solution to this part, you will not get any credit for this part. So be careful about Part B. Again, the goal is to minimize the number of unnecessary prefetches (ignoring epilog/prolog issues and the outer loop).

Solution:

Using the answer from the previous part, the loop should be unrolled 8 times. Using a technique analogous to software pipelining, we should insert a prefetch instruction at the start of the loop body to prefetch the cache line being used two (new) iterations later. The reason for prefetching two iterations ahead is that we need two iterations to cover the memory latency. The correct solution is:

```
for (int i = 0; i < 64; i++) {
     for (int j = 0; j < 64; j += 8) {
          prefetch(A[2*i][2*(j+16)]);
          x += A[2*i][2*(j+0)];
          x += A[2*i][2*(j+1)];
          x += A[2*i][2*(j+2)];
          x += A[2*i][2*(j+3)];
          x += A[2*i][2*(j+4)];
          x += A[2*i][2*(j+5)];
          x += A[2*i][2*(j+6)];
          x += A[2*i][2*(j+7)];
     }
}
```

Grading:

1 point for properly inserting the prefetch instruction.
1 point for calculating the correct array offset for the prefetch instruction.
2 points for unrolling the loop properly (correct indices for arrays and loop increment).
Take into account the answer to Part B for array offset and loop unrolling.
If an incorrect answer to Part B results in a trivial solution here e.g., no unrolling, no points are given.

Part D [3 points]

Consider your solution to Part C. How many of the data cache misses from the original code now have their latency fully hidden? Consider both the inner and outer loops for this part. You must explain how you derived this number to receive any credit. If an incorrect solution to Part C trivializes this problem or makes it too difficult, no credit will be given.

Solution:

The first two iterations of the new inner loop will still result in cache misses because that data is not being prefetched. There are a total of 64 invocations of the inner loop. Therefore there are a total of 128 L1 misses using prefetches. Since there were originally 512 L1 misses, 384 of these L1 misses have their latency fully hidden due to the prefetches.

Grading:

3 points for the correct answer with a reasonable explanation.
Cascading errors from previous parts are not penalized (unless they result in a trivial solution, e.g., all/no misses hidden).
Partial credit of 1.5 points for incorrect answers with a reasonable explanation.
No points for an answer without an explanation.

Part E [4 points]

Consider the data cache misses for which the latencies are not fully hidden in Part C. Suggest one enhancement to your code in Part C that can hide some of the exposed latency. Consider both the inner and outer loops for this part. You may answer with a clear description in words or with code that contains the enhancement accompanied with some explanatory text. Again, solutions to previous parts that trivialize this part will not get any credit.

Solution:

The following code eliminates all but the first two L1 misses:

```
for (int i = 0; i < 64; i++) {
     prefetch(A[2*(i+1)][0]);
     prefetch(A[2*(i+1)][16]);

     for (int j = 0; j < 64; j += 8) {
          prefetch(A[2*i][2*(j+16)]);
          x += A[2*i][2*(j+0)];
          x += A[2*i][2*(j+1)];
          x += A[2*i][2*(j+2)];
          x += A[2*i][2*(j+3)];
          x += A[2*i][2*(j+4)];
          x += A[2*i][2*(j+5)];
          x += A[2*i][2*(j+6)];
          x += A[2*i][2*(j+7)];
     }
}
```

Since the data cache is infinite, this simply prefetches the data needed by the first two iterations of the inner loop during the previous iteration of the outer loop. Therefore, the only two remaining L1 misses take place during the first iteration of the outer loop for the first two iterations of the inner loop.

Grading:

1 point for each correct prefetch instruction in the outer loop (2 total).
2 points for a correct explanation.
Alternate solutions that hide some of the exposed latency will be accepted. Cascading errors from previous parts (that don't result in a trivial solution here) will not be penalized.

Name:
NetID:

Problem 2 [11 points]

Consider a computer with a memory system with 16 bits for physical address, 32 bits for virtual address, page size of 4KB, 16 bit words, and word (16 bit) addressable memory. The computer system contains a 32KB data cache that is virtually indexed and physically tagged. The data cache is 8-way set associative and the cache line size is 16 words.

Part A [2 points]:
Specify what bit ranges of the address (virtual or physical) comprise the virtual page number and the physical page frame number. How many physical and virtual pages does this system have?

Solution:
Since 4KB is 2**11 words (for 16 bit words), 11 bits are needed as the offset within a page. So bits 11:31 of the virtual address are the virtual page number, and bits 11:15 of a physical address are the physical page number. There are 2^21 virtual pages, and 2^5 physical pages.

Grading:
0.5 point for each of the following: the bit ranges of the page number in the virtual address, the bit ranges of the page number in the physical address, the number of physical pages, and the number of virtual pages.

Part B [3 points]:
Assume each Page Table Entry (PTE) has 5 state bits (dirty bit, protection bits, etc.) in addition to the address translation. How many bits does each PTE consume? How much memory, in bytes, does the page table consume to map all the virtual pages? Assume only one level of page translation (as discussed in class) and assume each page table entry is word aligned.

Solution:
A PTE has 5 bits of the physical page number in addition to the 5 state bits. This makes a total of 10 bits which fits in one 16 bit word. Each mapping therefore takes one word. There are 2^21 mappings. Therefore, the page tables take up 4MB.

Grading:
1 point for the size in bits of a PTE. 1 point for the total number of PTEs. 0.5 point for rounding up the PTE size to a multiple of word size and 0.5 point for expressing the final answer as the product of the number and size of PTEs.

Part C [3 points]

Specify what bit ranges within the virtual or physical address (whichever is appropriate) comprise the cache index and cache tag bits. State explicitly whether the bits used are from the physical or virtual address.

Solution:

There are 32KB / 2 bytes per word / 16 words per line / 8 lines per set = 2^7 sets. The index needs 7 bits. The cache is virtually indexed, so the bits are taken from the virtual address. The offset within a cache line is 4 bits, so bits 4:10 are used for the index. However, the page size is 2^11 words, so these bits will be the same in the physical address as well. The tag is the remaining bits of the physical address, bits 11:15.

Grading:

1.5 points for each answer. 0.5 point for correct number of bits, 0.5 point for correct range of bits, and 0.5 point for specifying virtual or physical address.

Part D [3 points]

If the architect were to decrease the cache associativity to 4-way, what would the cache index bits be then? Would this create a problem for the Virtually Indexed/Physically Tagged caching scheme? If so, describe the problem and a hardware-only way of solving it that does not require waiting for address translation before indexing the cache.

Solution:

If the cache were only 4-way associative, there would be 256 sets. Then the cache index would be 8 bits long, and use bits 4:11 of an address. If these bits are taken from the virtual address, bit 11 is part of the virtual page number. If two arbitrary virtual pages are allowed to map to the same physical page, then these two virtual addresses could have different index bits. One solution is that while indexing, the cache should assume bit 11 could be 0 or 1 and look for the data in the two corresponding sets in parallel (i.e., in 8 total ways) Assuming the translation completes in the time that these ways are accessed, the correct way can be determined once the data is accessed.

Grading:

0.5 point for giving the range of index bits. 1 point for noticing and explaining how this makes aliasing problematic. 1.5 points for a correct hardware only solution (software-only solutions like prohibiting aliasing are not accepted, nor doing address translation before indexing). Cascading errors from previous parts resulting in a trivial solution (i.e., no aliasing) will be given partial credit of 1.5 points.

Name:
NetID:

Problem 3 [12 points]

This problem concerns MESIF, an invalidation-based snooping cache coherence protocol for bus-based shared-memory multiprocessors with a single level of cache per processor. The MESIF protocol has five states. A block can be in one of the following states in cache C:

- **M** *Modified*: The block is present in a single cache C. The block is dirty or modified; i.e., the data in the block reflects a more recent version than the copy in memory.
- **E** *Exclusive*: The block is present in a single cache C. The block is clean; i.e., memory has an up-to-date copy of the block.
- **S** *Shared*: The block is present in cache C and possibly present in other caches. The block is clean.
- **I** *Invalid*: The block is not valid in cache C. Space for the block may or may not be currently allocated in this cache.
- **F** *Forward*: The block is present in cache C and possibly present in other caches. The block is clean. Additionally, cache C must service the requests of other caches to this block; memory will not respond to requests for this block. Only one cache can have the block in Forward state.

If cache C has a block A in the Modified, Exclusive, or Forward state, then cache C responds to any requests for block A from other processors. If the request is a read, then the state of block A in cache C transitions to the Shared state, while the new copy of block A in the requester's cache assumes the Forward state. If cache C had block A in the Modified state, then memory updates its copy as well.

On replacement of a block in the Modified, Exclusive, or Forward state, memory resumes responsibility for servicing subsequent requests for that block.

For this problem, assume a system with three processors – P1, P2, and P3 – each with a single level cache that stores only a single block and starts empty.

Complete the following table for the MSI protocol studied in class and for the MESIF protocol described above. The first column gives memory accesses (Read or Write) initiated by P1, P2, or P3 to block A. For each protocol, you are to fill the entry for the states of block A in the caches of P1 and P2 after the access in the first column completes. You are also to fill the entry for the actions on the bus initiated by the processors or Memory (Mem) required to complete the access in column 1.

The possible states are: M, E, S, I, and F.

The possible bus actions are X / Y, where:

- X is the initiator of the action and can be: *P1, P2, P3, Mem, or None*
- Y is the action initiated by X and can be: *Get Block A, Send Block A, Invalidate, or None*

Name:
NetID:

For each entry, there can be multiple actions, possibly by multiple initiators.

Assume that the accesses occur in the order below and that no other memory accesses / traffic occur. The first row is filled out for you.

	MSI			MESIF		
Access on block A	**P1 State**	**P2 State**	**Bus Actions**	**P1 State**	**P2 State**	**Bus Actions**
P1 Read	S	I	P1 / Get Block A Mem / Send Block A	E	I	P1 / Get Block A Mem / Send Block A
P2 Read	S	S	P2 / Get Block A Mem / Send Block A	S	F	P2 / Get Block A P1 / Send Block A
P3 Read	S	S	P3 / Get Block A Mem / Send Block A	S	S	P3 / Get Block A P2 / Send Block A
P2 Write	I	M	P2 / Invalidate	I	M	P2 / Invalidate
P2 Write	I	M	None	I	M	None
P1 Write	M	I	P1 / Get Block A, Invalidate P2 / Send Block A	M	I	P1 / Get Block A, Invalidate P2 / Send Block A
P2 Read	S	S	P2 / Get Block A P1 / Send Block A	S	F	P2 / Get Block A P1 / Send Block A

Grading:
¼ point for each correct state entry.
½ point for each correct bus action entry, for all actions together.
No partial credit.
We aim to not penalize cascading errors, unless the cascading errors fundamentally impact the intended learning from the problem.

Problem 4 [10 points]

You are to implement a queue using an array in a multiprocessor system. The elements of the array can be accessed in parallel by multiple processors. You are to write two functions:

- ***enqueue***, which will add an element to the tail of the queue, and
- ***dequeue***, which will remove an element from the head of the queue.

Consider the following function definitions:

```
int head; /* index for the head of the queue */
int tail; /* index for the tail of the queue */
int local-index; /* current index for enqueuing or dequeuing,
each processor has its own copy */

enqueue(item) {

     queue[local-index] = item;

}

dequeue() {

     item = queue[local-index];

     return item
}
```

Assume the queue always has at least one element; i.e., a *dequeue* is never called on an empty queue. Also assume that the queue never gets full; i.e., the array is infinitely long and you don't have to worry about calling an *enqueue* on a full queue.

Name:
NetID:

Part A [6 points]:

Implement the *enqueue* and *dequeue* functions using an atomic *test&set* instruction to achieve synchronization. Don't worry about using test&test&set, but otherwise, write the most efficient code possible. Use C-like pseudocode to implement the functions. Assume local-index is local to each processor and is not part of shared memory; i.e., each processor has its own copy of this variable (e.g., in its own register). Assume the system implements sequential consistency.

Solution:

```
int head; /* index for the head of the queue */
int tail; /* index for the tail of the queue */
int local-index; /* current index for enqueuing or dequeuing,
each processor has its own copy */

enqueue(item) {
     while (test&set(qlock)); /* Spin until lock is obtained */
     local-index = tail; /* index for tail of queue */
     tail++;
     qlock = 0; /* unlock */
     queue[local-index] = item;
}

dequeue() {
     while (test&set(dqlock)); /* Spin until lock is obtained */
     local-index = head;
     head++;
     dqlock = 0; /* unlock */
     item = queue[local-index];
     return item;
}
```

Grading:

2.5 points for a correct queue function. 2.5 points for a correct dequeue function.
0.5 additional point for realizing that different locks must be used for these functions.
0.5 additional point for updating the queue outside the critical section in both the queue and dequeue functions.
Solutions that use an alternate interface for test&set compared to the one in the lecture, will not be penalized, as long as they are specified and reasonable.
6 total points.

Part B [4 points]:
Repeat Part A using the atomic *fetch&increment* instruction instead of the test&set for synchronization.

Solution:

```
int head; /* index for the head of the queue */
int tail; /* index for the tail of the queue */
int local-index; /* current index for enqueuing or dequeuing,
each processor has its own copy */

enqueue(item) {
     local-index = fetch&increment(tail);
     queue[local-index] = item;
     return;
}

dequeue() {
     local-index = fetch&increment(head);
     item = queue[local-index];
     return item;
}
```

Grading:
2 points for each function.
A solution that implements test&set using fetch&increment gets graded out of 3 points (1.5 points for each function).
Solutions that use an alternate interface for fetch&increment compared to the one in the lecture, will not be penalized, as long as they are specified and reasonable.

Name:
NetID:

Problem 5 [4 points]

This problem concerns the mini-project presentations in class. Circle the most appropriate choices for each question below. Some questions have multiple correct choices. Points will be given for a question only if all appropriate choices for that question are circled and no incorrect choice is circled.

Part A [1 point]

Which of the following branch predictors are used in AMD Zen 2?

a) Simple neural network (Perceptron)
b) Static, always-taken
c) G-share predictor
d) TAGE predictor

Part B [1 point]

Which of the following is true about Intel Lunar Lake?

a) It is built out of multiple smaller chiplets rather than one large monolithic chip
b) Its performance benefits come from Simultaneous Multithreading (Hyperthreading)
c) It includes a Neural Processing Unit dedicated for AI workloads
d) Its CPU includes 1000s of lightweight cores to achieve massive parallelism

Part C [1 point]

Which of the following is true about IBM POWER 10?

a) It moves toward a RISC architecture with 50 times the numbers of cores as POWER 9
b) It is the first IBM product with 3D stacking of 16 compute and 58 memory layers
c) It can be connected with 16 other POWER 10 chips to form a fully coherent system
d) Its CPU includes 1000s of lightweight cores to achieve massive parallelism

Part D [1 point]

Which of the following accelerators were present in the NVIDIA Ada Lovelace architecture?

a) Genomic sequencer
b) Tensor core
c) Ray tracing accelerator
d) Merge sort accelerator

Name:
NetID:

ALL STUDENTS SHOULD SOLVE PARTS A TO C.
ONLY GRADUATE STUDENTS SHOULD SOLVE PART D AND E.

Problem 6 [14 points for undergraduates, 22 points for graduates]

For this problem, consider a system that implements sequential consistency. You may assume that uniprocessor control and data dependencies are always respected. The following code fragment is executed on three processors:

```
Initially X = Y = 0

P1          P2          P3
X = 1       B = X       C = Y
A = X       Y = 2       X = 3
```

Part A [8 points]

Consider an execution of the program where the **final value of C = 2** and fill the table below. For each row:

- In the third column, indicate (by writing *Yes* or *No*) whether the corresponding combination of the final value of A and B (and C = 2 as specified earlier) is possible.
- In the fourth column, justify your (Yes/No) answer in the third column with a reason. You may answer in words or justify with an example execution order of the program.

Final value of A	Final value of B	Combination of A and B possible?	Reason for Yes/No answer in Column 3
0	0		
1	1		
1	3		
3	1		

Name:
NetID:

Final value of A	Final value of B	Combination of A and B possible?	Reason for Yes/No answer in Column 3
0	0	No	Due to the uniprocessor dependency between X = 1 and A = X on P1, A = X can only return the value of X = 1 or a later write to X.
1	1	Yes	If all instructions of P1 execute before P2 and all instructions of P2 execute before P3, A = B =1.
1	3	No	X = 3 can not be observed before C = Y. Therefore, if C = 2, then Y = 2 on P2 executed earlier in global order. Therefore, B = X should have executed earlier. As a result, there is no possible execution of the program where B = 3.
3	1	Yes	In the following order of instructions: P1: X = 1 → P2: B = X → P2: Y = 2 → P3: C = Y → P3: X = 3 → P1: A = X, then A = 3 and B = 1.

Grading:
2 points for each row of the table. 1 point for a correct answer in the third column and 1 point for a reasonable justification in the fourth column.
No points for a row without a reasonable justification.

Name:
NetID:

Part B [2 points]

What synchronization is required to ensure that all of P1's instructions are to be executed before any of P2's instructions, and all of P2's instructions are to be executed before any of P3's instructions? Modify the given program by inserting new (synchronization) instructions.

Do not reuse any of the variables in the original data accesses of the program to perform synchronization. Unnecessarily inefficient solutions will not get full credit.

Solution:

```
Initially X = Y = M = N = 0
```

P1	P2	P3
	`while(M != 1);`	`while(N != 1);`
`X = 1`	`B = X`	`C = Y`
`A = X`	`Y = 2`	`X = 3`
`M = 1`	`N = 1`	

Grading:

1 point for ensuring correct order between P1 and P2 with synchronization flags.
1 point for ensuring correct order between P2 and P3 with synchronization flags.

Name:
NetID:

Part C [4 points]

Consider a system that implements a relaxed consistency model, where any program-ordered pair of accesses to different variables may appear reordered to an external observer, except as restricted by a `fence` instruction. A fence instruction ensures all previous (by program order) reads and writes are complete to any observer in the system before subsequent (by program order) accesses can begin. You may assume that uniprocessor control and data dependencies are still respected.

Starting with the solution from Part B, modify the program to **introduce fence instructions** such that the values of A, B, and C at the end of an execution of the modified program on the system that implements the relaxed consistency model described above are the same as Part B (i.e., a system that implements sequential consistency). Ensure that fence instructions are used judiciously (only where necessary). Solutions with unnecessary fence instructions will not get full credit. Full credit for this part requires a reasonable answer to Part B.

Solution:

```
Initially X = Y = M = N = 0
```

P1	P2	P3
	`while(M != 1);`	`while(N != 1);`
`X = 1`	`fence`	`fence`
`A = X`	`B = X`	`C = Y`
`fence`	`Y = 2`	`X = 3`
`M = 1`	`fence`	
	`N = 1`	

Grading:

1 point for each fence instruction placed at the correct place in the program. 4 total points.
-1 point if unnecessary fences are introduced (e.g., between X = 1 and A = X in P1)

ONLY GRADUATE STUDENTS SHOULD SOLVE PART D AND E.

Part D [4 points]

Describe how the use of a write buffer for Processor 1 could potentially violate sequential consistency for your solution in Part C, if the system ignored the fence instructions. Full credit will be given only if your solution in Part C is reasonable.

Now describe how you would consider the fence instructions in the design of the write buffer to avoid the above violation of sequential consistency.

Solution:

The writes to X and M from P1 may be buffered in the write buffer, and evicted in any order. For example, the write to M may be written out before the write to X – leading to P2 observing X as 0. This violates the guarantees of sequential consistency.

To avoid the above violation of sequential consistency, the write buffer must be flushed at the point a fence instruction is executed. All writes pending in the write buffer (before the fence in program order) must be visible to all processors before the fence completes.

Grading:

2 points for the correct scenario. There are alternate solutions possible, which will not be penalized unless the basis for the scenario is fundamentally incorrect (e.g., a scenario that depends on some reordering from a uniprocessor point of view).
2 points for the correct solution to avoid the violation of sequential consistency.

Part E [4 points]

Again, consider your solution in Part C. Processor 3 is designed with an out-of-order pipeline employing Tomsaulo's algorithm with speculative execution of loads. Describe how this could potentially violate sequential consistency for your solution in Part C, if the system ignored the fence instructions. Full credit will be given only if your solution in Part C is reasonable.

Again, describe how you would consider the fence instructions in the design of the above pipeline for Processor 3 to avoid the above violation of sequential consistency.

Solution:

P3 may speculatively read C = Y even before the while (N != 1) is not complete. This will lead to the case where C may be read as 0. This violates the guarantees of sequential consistency.

To avoid the above violation of sequential consistency, no loads (or memory operations in general) that are after the fence in program order may be issued until the fence is committed.

Grading:

2 points for the correct scenario. There are alternate solutions possible, which will not be penalized unless the basis for the scenario is fundamentally incorrect (e.g., a scenario that depends on some reordering from a uniprocessor point of view).
2 points for the correct solution to avoid the violation of sequential consistency.