
CS152/252A
Fall 2026

Memory Consistency, Cache Coherence, and Synchronization Primitives

Problem Set #4

Due: November 8 @ 11:59:59 PM PST

The problem sets are intended to help you learn the material, and we encourage you to collaborate with other students and to ask questions in discussion sections and office hours to understand the problems. However, each student must turn in their own solution to the problems.

The problem sets also provide essential background material for the exam and the midterms. The problem sets will be graded primarily on a completion basis, but with a few problems graded for correctness to reward earnest engagement. But if you do not work through the problem sets yourself you are unlikely to succeed on the exam or midterms!

We will distribute solutions to the problem set after the deadline to give you feedback.

Assignments must be submitted through Gradescope by **11:59:59pm PT** on the specified due date. *Box/ clearly mark all solutions that don't involve filling in a figure/table. Only boxed/clearly marked solutions and filled in figures/tables will be considered for grading.* See the course website for the policy on slip days (late submissions).

Name: _____

SID: _____

Collaborators (Name, SID):

Problem 1: Branch Prediction

Problem 1.A

Global versus Local History (4 points)

Consider the following program, executed repeatedly. Each `randbit` instruction generates an independent, uniformly random bit. There are exactly **two** filler branches between A and B, and the only conditional branches entering the predictor's global history are A, the two fillers, and B.

```

    randbit a0
    beq     a0, x0, A_DONE    # Branch A
A_DONE:
    randbit t0
    beq     t0, x0, F1_DONE  # Filler 1
F1_DONE:
    randbit t0
    beq     t0, x0, F2_DONE  # Filler 2
F2_DONE:
    beq     a0, x0, B_DONE    # Branch B
B_DONE:

```

(2 points) Find the minimum number **G** of global-history bits required to predict B with 100% long-run accuracy, assuming a sufficiently large prediction table.

(2 points) If the predictor instead uses only B's local branch-outcome history, what is its best possible long-run accuracy? Explain.

Problem 1.B**Predictor-Table Aliasing (4 points)**

Two branches execute equally often. X is never taken and Y is always taken:

```
PC_X:  bne x0, x0, X_TARGET    # PC = 0x100; never taken
PC_Y:  beq x0, x0, Y_TARGET    # PC = 0x110; always taken
```

A predictor uses a shared table containing **M** one-bit predictions. M must be a power of two. Entries are programmed optimally and held fixed during evaluation. The table index is:

```
index = (PC >> 2) & (M - 1);
```

(2 points) Find the minimum **M** that allows 100% accuracy on both branches under this indexing rule. Explain how the indices differ.

(2 points) What is the highest combined prediction accuracy possible using **M/2** entries under the same indexing rule?

Problem 2: Relaxed Memory Models

The following code implements a *seqlock*, which is a reader-writer lock that supports a single writer and multiple readers. The writer never has to wait to update the data protected by the lock, but readers may have to wait if the writer is busy. We use a seqlock to protect a variable that holds the current time. The lock is necessary because the variable is 64 bits and thus cannot be read or written atomically on a 32-bit system.

The seqlock is implemented using a sequence number, *seqno*, which is initially zero. The writer begins by incrementing *seqno*. It then writes the new time value, which is split into the 32-bit values *time_lo* and *time_hi*. Finally, it increments *seqno* again. Thus, if and only if *seqno* is odd, the writer is currently updating the counter.

The reader begins by waiting until *seqno* is even. It then reads *time_lo* and *time_hi*. Finally, it reads *seqno* again. If *seqno* didn't change from the first read, then the read was successful; otherwise, the read is retried.

This code is correct on a sequentially consistent system, but on a system with a fully relaxed memory model it may not be. Insert the minimum number of memory fences to make the code correct on a system with a relaxed memory model. To insert a fence, write the needed fence (*Membar_{LL}*, *Membar_{LS}*, *Membar_{SL}*, *Membar_{SS}*) in the blanks between the lines of code below. Not all blanks need to be used, please leave unused blanks empty.

Writer	Reader
LOAD R _{seqno} , (seqno)	Loop: LOAD R _{seqno_before} , (seqno)
ADD R _{seqno} , R _{seqno} , 1	IF(R _{seqno_before} & 1)
STORE (seqno), R _{seqno}	goto Loop
STORE (time_lo), R _{time_lo}	LOAD R _{time_lo} , (time_lo)
STORE (time_hi), R _{time_hi}	LOAD R _{time_hi} , (time_hi)
ADD R _{seqno} , R _{seqno} , 1	LOAD R _{seqno_after} , (seqno)
STORE (seqno), R _{seqno}	IF(R _{seqno_before} != R _{seqno_after})
	goto Loop

Problem 3: Directory-based Cache Coherence Update Protocols

Please refer to Handout #6 (on website) for this problem.

In Handout #6, we examine a cache-coherent distributed shared memory system. Ben wants to convert the directory-based invalidate cache coherence protocol from the handout into an update protocol. He proposes the following scheme.

Caches are write-through, not write allocate. When a processor wants to write to a memory location, it sends a WriteReq to the memory, along with the data word that it wants written. The memory processor updates the memory and sends an UpdateReq with the new data to each of the sites caching the block, unless that site is the processor performing the store, in which case it sends a WriteRep containing the new data.

If the processor performing the store is caching the block being written, it must wait for the reply from the home site to arrive before storing the new value into its cache. If the processor performing the store is not caching the block being written, it can proceed after issuing the

WriteReq.

Ben wants his protocol to perform well, and so he also proposes to implement silent drops. When a cache line needs to be evicted, it is silently evicted and the memory processor is not notified of this event.

Note that WriteReq and UpdateReq contain data at the word-granularity, and not at the block-granularity. Also note that in the proposed scheme, memory will always have the most up-to-date data and the state C-exclusive is no longer used.

As in the lecture, the interconnection network guarantees that message-passing is reliable, and free from deadlock, livelock, and starvation. Also as in the lecture, message-passing is FIFO, meaning; each home site keeps a FIFO queue of incoming requests and processes them in the order received.

Problem 3.A**Sequential Consistency**

Alyssa claims that Ben's protocol does not preserve sequential consistency because it allows two processors to observe stores in different orders. Describe a scenario in which this problem can occur.

Problem 3.B**State Transitions**

Noting that many commercial systems do not guarantee sequential consistency, Ben decides to implement his protocol anyway. Fill in the following state transition tables (Table P4-1 and Table P4-2) for the proposed scheme. (Note: the tables do not contain all the transitions for the protocol).

No.	Current State	Event Received	Next State	Action
1	C-nothing	Load		
2	C-nothing	Store		
3	C-nothing	UpdateReq		
4		Load	C-shared	
5	C-shared	Store		
6	C-shared	UpdateReq		
7	C-shared	(Silent drop)		
8	C-transient	ShRep		
9	C-transient	WriteRep		
10	C-transient	UpdateReq		

Table P4-1: Cache State Transitions

No.	Current State	Message Received	Next State	Action
1	R(dir) & id is not in dir	ShReq		
2	R(dir) & id is not in dir	WriteReq		
3	R(dir) & id is in dir	ShReq		
4	R(dir) & id is in dir	WriteReq		

Table P4-2: Directory State Transitions

Problem 3.C

UpdateReq

After running a system with this protocol for a long time, Ben finds that the network is flooded with UpdateReqs. Alyssa says this is a bug in his protocol. What is the problem and how can you fix it?

Problem 3.D

FIFO Assumption

FIFO message passing is a necessary assumption for the correctness of the protocol. If the network were non-FIFO, it becomes possible for a processor to never see the result of another processor's store. Describe a scenario in which this problem can occur.

Problem 4: Snoopy Cache Coherent Shared Memory

Please refer to Handout #7 (on website) for this problem.

In this problem, we investigate the operation of the snoopy cache coherence protocol in Handout #7. The following questions are to help you check your understanding of the coherence protocol. You do not need to answer these for credit.

- Explain the differences between **CR**, **CI**, and **CRI** in terms of their purpose, usage, and the actions that must be taken by memory and by the different caches involved.
- Explain why **WR** is not snooped on the bus.
- Explain the I/O coherence problem that **CWI** helps avoid.

Problem 4.A**Current-Value Locations**

In Table Q4-1, Q4-2, and Q4-3, column 1 indicates the initial state of a certain address X in a cache. Column 2 indicates whether address X is currently cached in any other cache. (The “cached” information is known to the cache controller only immediately following a bus transaction. Thus, the action taken by the cache controller must be independent of this signal, but state transition could depend on this knowledge.) Column 3 enumerates all the available operations on address X, either issued by the CPU (read, write), snooped on the bus (**CR**, **CRI**, **CI**, etc), or initiated by the cache itself (replacement). Some state-operation combinations are impossible; you should mark them as such. (See the first table for examples). In columns 6, 7, and 8 (corresponding to this cache, other caches and memory, respectively), **check all possible locations where up-to-date copies of this data block could exist after the operation in column 3 has taken place** and ignore column 4 and 5 for now. Table Q4-1 has been completed for you. Make sure the answers in this table make sense to you.

Problem 4.B**MBus State Transitions**

In this problem, **we ask you to fill out the state transitions in Column 4 and 5**. In column 5, fill in the resulting state after the operation in column 3 has taken place. In column 4, list the necessary MBus transactions that are issued by the cache as part of the transition. Remember, the protocol should be optimized such that data is supplied using **CCI** *whenever possible*, and only the cache that **owns** a line should issue **CCI**.

initial state	other	ops	actions by this	final	this	other	mem
	cached		cache	state	cache	caches	
Invalid	no	none	none	I			yes
		CPU read	CR	CE	yes		yes
		CPU write	CRI	OE	yes		
		replace	none		<i>impossible</i>		
		CR	none	I		yes	yes
		CRI	none	I		yes	
		CI	none		<i>impossible</i>		
		WR	none		<i>impossible</i>		
		CWI	none	I			yes
Invalid	yes	none		I		yes	yes
		CPU read	same	CS	yes	yes	yes
		CPU write		OE	yes		
		replace			<i>impossible</i>		
		CR	as	I		yes	yes
		CRI	above	I		yes	
		CI		I		yes	
		WR		I		yes	yes
		CWI		I			yes
initial state	other	ops	Actions by this	final	this	other	mem
	cached		cache	state	cache	caches	
cleanExclusive	no	none	none	CE			
		CPU read					
		CPU write					
		replace					
		CR					
		CRI					
		CI					
		WR					
		CWI					

Table Q4-1

1 Some Sun MBus implementations perform CCI from the cleanExclusive state, while others do not. We accept both answers.

initial state	other	ops	Actions by this	final	this	other	mem
	cached		cache	state	cache	caches	
ownedExclusive	no	none	none	OE			
		CPU read					
		CPU write					
		replace					
		CR		OS			
		CRI					
		CI					
		WR					
		CWI					
initial state	other	ops	actions by this	final	this	other	mem
	cached		cache	state	cache	caches	
cleanShared	no	none	none	CS			
		CPU read					
		CPU write					
		replace					
		CR					
		CRI					
		CI					
		WR					
		CWI					
cleanShared	yes	none					
		CPU read					
		CPU write	same				
		replace		I		yes	
		CR	as	CS			
		CRI	above	I		yes	
		CI		I		yes	
		WR		CS			
		CWI		I			yes

Table Q4-2

2 Some Sun MBus implementations perform CCI from the cleanShared state. However, in these implementations, requests are not broadcast on a bus, but are handled by a central system controller. The system controller arbitrates which cache with a cleanShared copy provides the data. Unless an explanation is provided, CCI is not a valid response from this state.

initial state	other	ops	actions by this	final	this	other	mem
	cached		cache	state	cache	caches	
ownedShared	no	none	none	OS			
		CPU read					
		CPU write					
		replace					
		CR					
		CRI					
		CI					
		WR					
		CWI					
ownedShared	yes	none					
		CPU read					
		CPU write	same				
		replace		I		yes	
		CR	as	OS			
		CRI	above	I		yes	
		CI		I		yes	
		WR		impossible			
		CWI		I			yes

Table Q4-3

Problem 5: Locking Performance (OPTIONAL)

While analyzing some code, you find that a big performance bottleneck involves many threads trying to acquire a single lock.

Conceptually, the code is as follows:

```
int mutex = 0;
while (true) {
    noncritical_code();
    lock(&mutex);
    critical_code();
    unlock(&mutex);
}
```

Assume for all questions that our processor is using a directory protocol, as described in Handout #6.

Test&Set Implementation

First, we will use the atomic instruction `test_and_set` to implement the `lock(mutex)` and `unlock(mutex)` functions.

In C, the instruction has the following function prototype:

```
int return_value = test_and_set(int* maddr);
```

Recall that `test_and_set` atomically reads the memory address `maddr` and writes a 1 to the location, returning the original value.

Using `test_and_set`, we arrive at the following first-draft implementation for the `lock()` and `unlock()` functions:

```
inline void lock(int* mutex_ptr)
{
    while (test_and_set(mutex_ptr) == 1);
}

inline void unlock(int* mutex_ptr)
{
    *mutex_ptr = 0;
}
```

Problem 5.A**Test&Set, The Initial Acquire**

Let us analyze the behavior of Test&Set while running 1,000 threads on 1,000 cores.

Consider the following scenario: At the start of the program, the lock is invalid in all caches. Then, every thread executes Test&Set once. The first thread wins the lock, while the other threads will find that the lock is taken. How many invalidation messages must be sent when all 1,000 threads execute Test&Set once?

Problem 5.B**Test&Set, Spinning**

While the first thread is in the critical section (the “winning thread”), the remaining threads continue to execute Test&Set, attempting to acquire the lock. Each waiting thread is able to execute Test&Set five times before the winning thread frees the lock. How many invalidation messages must be sent while the winning thread was executing the critical section?

Problem 5.C**Test&Set, Freeing the Lock**

How many invalidation messages must be sent when the winning thread frees the lock? Assume the critical section is very long, and all 999 other threads have been waiting to acquire the lock.

Test&Test&Set Implementation

Since our analysis from the previous parts show that a lot of invalidation messages must be sent while waiting for the lock to be freed, let us instead use a regular load alongside the atomic instruction test&set to implement the mutex lock.

```
void inline lock(int* mutex_ptr)
{
    while ((*mutex_ptr == 1) || test&set(mutex_ptr) == 1);
}

void inline unlock(int* mutex_ptr)
{
    *mutex_ptr = 0;
}
```

Note: the loop evaluation is short-circuited if the first part is true; thus, test&set is only executed if (*mutex_ptr) does not equal 1.

Problem 5.D**Test&Test&Set, The Initial Acquire**

Let us analyze the behavior of Test&Test&Set while running 1,000 threads on 1,000 cores.

Consider the following scenario: At the start of the program, the lock is invalid in all caches. Then every thread performs the first Test (reading *mutex_ptr*) once. After every thread has performed the first Test (which evaluates to *False*, because *mutex* == 0), each thread then executes the atomic Test&Set once. Naturally, only one thread wins the lock. How many invalidation messages must be sent in this scenario?

Problem 5.E**Test&Test&Set, Spinning**

While the first thread is in the critical section, the remaining threads continue to execute Test&Test&Set. Each waiting thread is able to execute Test&Test&Set five times before the winning thread frees the lock. How many invalidation messages must be sent while the winning thread was executing the critical section?

Problem 5.F

Test&Test&Set, Freeing the Lock

How many invalidation messages must be sent when the winning thread frees the lock for the Test&Test&Set implementation? Assume the critical section is very long, and all 999 other threads have been waiting to acquire the lock.