Problem Set 5 Solutions CS152 Spring 2019

Problem P5.1: Sequential Consistency

Problem P5.1.A

Can X hold value of 4 after all three threads have completed? Please explain briefly.

(Yes) / No

C1, B1-B6, A1-A4, C2-C4

Problem P5.1.B

Can X hold value of 5 after all three threads have completed?

Yes / (No)

All results must be even!

Problem P5.1.C

Can X hold value of 6 after all three threads have completed?

(Yes) / No

All of C, All of A, All of B

Problem P5.1.D

For this particular program, can a processor that reorders instructions but follows local dependencies produce an answer that cannot be produced under the SC model?

Yes / (No)

All stores/loads must be done in order because they're to the same address, so no new results are possible.

Problem P5.2: Synchronization Primitives

The mechanism here is as follows: LdR requests READ access to the address, StC requests WRITE access to the address. Many students suggested that LdR can request WRITE access to the address right away, which could lead to live lock.

Problem P5.2.A

Describe under what events the local reservation for an address is cleared.

If another processor requests Write access to the same cache line.

Problem P5.2.B

Is it possible to implement LdR/StC pair in such a way that the memory bus is not affected, i.e., unaware of the addition of these new instructions? Explain

Yes. WbReq and ShReq are sent normally. The "reservation" is local (probably in the snooper or in the cache, though that might take too much resources – there are very few reservations needed at the same time for any processor).

Problem P5.2.C

Give two reasons why the LdR/StC pair of instructions is preferable over atomic read-test-modify instructions such as the TEST&SET instruction.

1. Bus doesn't need to be aware of them.
2. Everything is local.
3. No ping-pong.
4. No extra hardware (tied to 1)

Problem P5.2.D

LdR/StC pair of instructions were conceived in the context of snoopy busses. Do these instructions make sense in our directory-based system in Handout #7? Do they still offer an advantage over atomic read-test-modify instructions in a directory-based system? Please explain.
No they don't offer an advantage. Busses have the advantage that they observe other transactions and thus could track other LdRs. But with coherence a store invalidates
All sharers anyway which is what the directory would do. So the result is the same.

Problem P5.3: 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 between the lines of code below.

Writer&	**Reader&**
!	!
LOAD R_{seqno}, (seqno)	Loop: LOAD R_{seqno_before}, (seqno)
ADD R_{seqno}, R_{seqno}, 1	IF(R_{seqno_before} & 1) goto Loop
	$Membar_{LL}$
STORE (seqno), R_{seqno}	LOAD R_{time_lo}, (time_lo)
$Membar_{SS}$	
STORE (time_lo), R_{time_lo}	LOAD R_{time_hi}, (time_hi)
	$Membar_{LL}$
STORE (time_hi), R_{time_hi}	LOAD R_{seqno_after}, (seqno)
$Membar_{SS}$	
ADD R_{seqno}, R_{seqno}, 1	IF(R_{seqno_before} != R_{seqno_after})
!	
!	goto Loop!
STORE (seqno), R_{seqno}!	

Problem P5.4: Locking Performance

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&set` to implement the *lock(mutex)* and *unlock(mutex)* functions.

In C, the instruction has the following function prototype:

```
int return_value = test&set(int* maddr);
```

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

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

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

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

Problem P5.4.A **Test&Set, The Initial Acquire**

Let us analyze the behavior of `Test&Set` while running 1,000 threads on a 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?

1,000 Test&Sets are performed in the above scenario.

Test&Set is an atomic read-write operation and requires exclusive access to the lock's address. Therefore, each Test&Set invalidates the previous core who performed Test&Set. However, the first core had no one to invalidate, because the lock was initially uncached. Therefore, 999 invalidation messages were sent.

Invalidations _999_

Problem P5.4.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?

999 cores are spinning, each executes T&S five times for a total of 4995 Test&Sets performed.

Each Test&Set invalidates the previous core who performed Test&Set. Therefore, 4995 invalidation messages were sent.

(This assumes that every thread is interleaved).

Invalidations ____4995___

Problem P5.4.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.

Freeing the lock involves writing to the lock's address which requires invalidating anybody else who has cached that address. Because all of the other cores are spinning on Test&Set, and only one core will have the lock address at a time, the winning lock will invalidate only the last core to perform a Test&Set.

Invalidations ____1___

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 the atomic instruction `test&set` to implement Test&Test&Set.

```
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 P5.4.D **Test&Set&Set, The Initial Acquire**

Let us analyze the behavior of `Test&Test&Set` while running 1,000 threads on a 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?

1,000 cores perform the first Test. That requires read permissions and invalidates nobody (since the lock is initially invalid). All 1,000 cores end up with a copy of the lock.

Then, all cores execute T&S. The *first* T&S will invalidate the other 999 cores' copy, for 999 invalidations.

The other 999 T&S's will invalidate the previous core to perform T&S, for 999 more invalidations. In total 999+999 invalidations occur.

Invalidations ___1998____

Problem P5.4.E **Test&Set&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?

Once the lock has been grabbed by the winning core, the other 999 threads will only see *mutex == 1*, and not execute the Test&Set. Therefore, executing the 4995 Test&Test&Sets while waiting for the lock to be freed only requires read permissions.

However, the very *first* Test&Test&Set will require downgrading the last core who performed a Test&Set operation to the *Shared* state, so it could be argued that 1 invalidation message was sent (technically, a *WriteBackRequest* message). So 0 invalidations occurred and 1 downgrade occurred. Either 0 or 1 would be acceptable answers.

Invalidations __0/1_____

Test&Set&Set, Freeing the Lock

Problem P5.4.F

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.

Freeing the lock will require invalidating the 999 shared copies held by the spinning threads.

Invalidations __999_____

Problem P5.5: Directory-based Cache Coherence Invalidate Protocols

In this problem we consider a cache-coherence protocol presented in Handout #6.

Problem P5.5.A — Protocol Understanding

Consider the following scenario:
Assume initially the home directory state is W(m), indicating that the block is modified at site m.

1. The home site receives a ExReq from a site (not m). The home site sends a FlushReq to site m. The ExReq is suspended and buffered at the home site. The home directory site becomes Tw(m).
2. Before the FlushReq arrives, the modified cache line is replaced at site m (due to a cache line conflict). Site m sends a FlushRep to the home site. The cache line state becomes C-nothing. The processor at site m issues a ExReq to the home site. The ExReq is suspended at site m. The cache line state becomes C-pending.
3. The FlushReq arrives at site m.

Problem P5.5.B — Non-FIFO Network

Assume initially that a particular block is not cached at any site. Consider the following scenario:

1. The home site receives a ShReq from site A.
2. The home site sends a ShRep to site A and changes the state to R({A})
3. The home site receives a ExReq from site B.
4. The home site sends a InvReq to site A and changes the state to Tr({A}).
5. The InvReq arrives at site A before the ShRep.
6. Site A dequeues the InvReq , it DOESN'T send a InvRep to the home site (because Site A is in C-pending state).
7. The ShRep arrives at site A and A changes state to C-shared.
8. The home site is still in Tr({A}).
9. Site B is still waiting for ExRep.

There is a deadlock in this situation.

Problem P5.5.C — Replace

When a cache A decides to invalidate a shared cache line and the directory is not informed, the home directory will continue to have A in its member list R(dir). Say cache C (not a current sharer) now wants to write to this cache line, an ExReq will be sent to the directory. The directory then sends an InvReq to cache A as well as any other nodes in the member list and the directory changes to the Tr(dir) state. It waits for InvRep from all these nodes before an ExRep to C can be sent. However, it will not receive a reply from A.

Problem P5.6: Directory-base Cache Coherence Update Protocols

Problem P5.6.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.

Consider blocks X and Y, whose home sites are A and B, respectively:

1. A receives a WriteReq for X, and B receives a WriteReq for Y
2. C and D are both caching X and Y. So both A and B send UpdateReq to C and D.
3. C receives first the UpdateReq for X, then the UpdateReq for Y
4. D receives first the UpdateReq for Y, then the UpdateReq for X

C and D can receive the UpdateReq in different orders because they are arriving from different home sites, and FIFO message passing only provides guarantees for messages with the same source and destination.

Problem P5.6.B — State Transitions

No.	Current State	Event Received	Next State	Dequeue Message?	Action
1	C-nothing	Load	C-transient	**No**	ShReq(id, Home, a)
2	C-nothing	Store	**C-transient**	**No**	**WriteReq(id, Home, a)**
3	C-nothing	UpdateReq	**C-nothing**	**Yes**	**None**
4	C-shared	Load	C-shared	**Yes**	reads cache
5	C-shared	Store	**C-transient**	**No**	**WriteReq(id, Home, a)**
6	C-shared	UpdateReq	**C-shared**	**Yes**	**data-> cache**
7	C-shared	(Silent Drop)	**C-nothing**	**N/A**	None
8	C-transient	ShRep	**C-shared**	**Yes**	data → cache
9	C-transient	WriteRep	**C-shared**	**Yes**	**data -> cache, Store retires**
10	C-transient	UpdateReq	**C-transient**	**Yes**	**data -> cache**

Table P5.6-1: Cache State Transitions

No.	Current State	Message Received	Next State	Dequeue Message?	Action
1	R(dir) & id ∉ dir	ShReq	R(dir + {id})	**Yes**	ShRep(Home, id, a)
2	R(dir) & id ∉ dir	WriteReq	**R(dir + {id})**	**Yes**	**data -> memory** **forall i** in **dir** **if(i==id) WriteRep-> id** **else UpdateReq-> i**
3	R(dir) & id ∈ dir	ShReq	**R(dir)**	**Yes**	ShRep(Home, id, a)
4	R(dir) & id ∈ dir	WriteReq	**R(dir)**	**Yes**	**data -> memory** **forall i** in **dir** **if(i==id) WriteRep-> id** **else UpdateReq-> i**

Table P5.6-2: Home Directory State Transitions

Problem P5.6.C **UpdateReq**

Because caches do not notify the home site when a line gets replaced, the set S for a memory block will only increase. Eventually, every site that has ever loaded a particular block will be in the set S for that block, resulting in numerous UpdateReq on the network, even though many of the recipients of the update have already replaced that cache line. The solution is to notify the home site when a cache line is replaced, and have the home site remove a site from S when such a notification is received.

Problem P5.6.D **FIFO Assumption**

Consider a site A:

1. Site A sends ShReq for block X to X's home site.
2. X's home site receives ShReq and issues a ShRep.
3. Site B sends a WriteReq for block X to X's home site.
4. X's home receives WriteReq, and issues an UpdateReq to A
5. The ShReq and UpdateReq are re-ordered in the network
6. The UpdateReq arrives at A. Since A is waiting for ShRep, it is in C-transient. It updates its cache with the UpdateReq data, but remains in C-transient.
7. The ShRep arrives at A. The cache writes the stale data into its cache, and reads the result.

Although A received the UpdateReq, it will never see the result of B's store unless the cache line gets replaced, and is re-loaded.

Problem P5.7: Snoopy Cache Coherent Shared Memory

Problem P5.7.A — Where in the Memory System is the Current Value

See Table P5.7-1, P5.7-2 and P5.7-3.

Problem P5.7.B — MBus Cache Block State Transition Table

See Table P5.7-1, P5.7-2 and P5.7-3.

Problem P5.7.C — Adding atomic memory operations to MBus

Imagine a dual processor machine with CPUs A and B. Explain the difficulty of CPU A performing fetch -and-increment(x) when the most recent copy of x is cleanExclusive in CPU B's cache. You may wish to illustrate the problem with a short sequence of events at processor A and B.

The problem is that CPU B can read the value in location x while CPU A is performing the fetch-and-increment operation—which violates the idea of fetch-and-increment being atomic. For example, consider the following sequence of events and corresponding state transitions and operations:

Event	**CPU A**	**CPU B**
1	Read(x); I->CS; send CR	
2		Snoop CR; CE->CS
3		Read(x)
4	Write(x); CS->OE; send CI	
5		Snoop CI; CS->I

Fill in the rest of the table below as before, indicating state, next state, where the block in question may reside, and the CPU A and MBus transactions that would need to occur atomically to implement a fetch-and-increment on processor A.

State	other cached	ops	actions by this cache	next state	this cache	other caches	mem
Invalid	yes	read	**CR**	**CS**	yes	yes	yes
cleanShared	yes	write	**CI**	**OE**	yes		

initial state	other cached	ops	actions by this cache	final state	this cache	other caches	mem
Invalid	no	none	none	**I**			Yes
		CPU read	**CR**	**CE**	Yes		Yes
		CPU write	**CRI**	**OE**	Yes		
		replace	none	*Impossible*			
		CR	none	**I**		yes	Yes
		CRI	none	**I**		yes	
		CI	none	*Impossible*			
		WR	none	*Impossible*			
		CWI	none	**I**			Yes
Invalid	yes	none	same as above	**I**		Yes	Yes
		CPU read		**CS**	Yes	Yes	yes
		CPU write		**OE**	Yes		
		replace		*Impossible*			
		CR		**I**		yes	Yes
		CRI		**I**		Yes	
		CI		**I**		Yes	
		WR		**I**		yes	Yes
		CWI		**I**			yes

initial state	other cached	ops	Actions by this cache	final state	this cache	other caches	mem
cleanExclusive	no	none	none	**CE**	Yes		Yes
		CPU read	none	CE	Yes		Yes
		CPU write	none	OE	yes		
		replace	none	I			Yes
		CR	none or CCI[1]	**CS**	yes	yes	yes
		CRI	none or CCI[1]	I		yes	
		CI	none	*Impossible*			
		WR	none	*Impossible*			
		CWI	none	I			yes

Table P5.7-1

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

initial state	other cached	ops	Actions by this cache	final state	this cache	other caches	mem
ownedExclusive	no	none	none	**OE**	Yes		
		CPU read	none	**OE**	Yes		
		CPU write	none	**OE**	Yes		
		replace	WR	**I**			Yes
		CR	CCI	**OS**	Yes	yes	
		CRI	CCI	**I**		Yes	
		CI	none	*Impossible*			
		WR	none	*Impossible*			
		CWI	none	**I**			Yes

initial state	other cached	ops	actions by this cache	final state	this cache	other caches	Mem
cleanShared	no	none	none	**CS**	Yes		Yes
		CPU read	none	**CS**	Yes		yes
		CPU write	CI	**OE**	Yes		
		replace	none	**I**			Yes
		CR	none[2]	**CS**	yes	Yes	yes
		CRI	none	**I**		Yes	
		CI	none	*Impossible*			
		WR	none	*Impossible*			
		CWI	none	**I**			Yes
cleanShared	yes	none	same as above	**CS**	Yes	yes	Yes
		CPU read		**CS**	Yes	Yes	yes
		CPU write		**OE**	Yes		
		replace		**I**		yes	Yes
		CR		**CS**	Yes	yes	yes
		CRI		**I**		Yes	
		CI		**I**		Yes	
		WR		**CS**	yes	yes	Yes
		CWI		**I**			yes

Table P5.7-2

[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 cached	ops	actions by this cache	final state	this cache	other caches	mem
ownedShared	no	none	none	**OS**	Yes		
		CPU read	none	**OS**	Yes		
		CPU write	CI	**OE**	Yes		
		replace	WR	**I**			Yes
		CR	CCI	**OS**	yes	Yes	
		CRI	CCI	**I**		yes	
		CI	none	*Impossible*			
		WR	none	*Impossible*			
		CWI	none	**I**			yes
ownedShared	yes	none	same as above	**OS**	yes	Yes	
		CPU read		**OS**	Yes	yes	
		CPU write		**OE**	Yes		
		replace		**I**		yes	Yes
		CR		**OS**	Yes	yes	
		CRI		**I**		Yes	
		CI		**I**		Yes	
		WR		*Impossible*			
		CWI		**I**			yes

Table P5.7-3