

EECS 570 Midterm Exam

Fall 2025

Name: _____ Uniqname: _____

Sign the honor code:

I have neither given nor received aid on this exam nor observed anyone else doing so.

Scores:

#	Question	Points
1	Short Answers	/ 15
2	Amdahl's Law	/ 5
3	Coherence Overhead	/ 10
4	Coherence Protocol	/ 20
5	Vector Processing	/ 10
6	Transactional Memory	/ 15
7	Memory Consistency	/20
<i>Total</i>		/ 95

NOTES:

- Calculators are allowed, but no PDAs, portables, cell phones, etc.
- Don't spend too much time on any one problem.
- You have 90 minutes for the exam.
- Be sure to show work and explain what you've done when asked to do so.
- The exam has 17 pages. Make sure you have all of them.

1) Short Answer [15 pts]

a) Consider sense-reversing barrier implementation. Explain in 1-2 sentences why we need the "sense" variable to alternate between true/false, rather than just using a simple counter. [2 pts]

The **sense** variable must alternate so threads can distinguish between consecutive barrier phases; otherwise, a thread finishing one barrier might see the same flag value and incorrectly proceed into the next phase, potentially causing **deadlock**.

b) In hardware transactional memory (HTM), which system updates architectural memory state before committing a transaction. Choose one. [1 point]

- i) eager conflict detection ii) lazy conflict detection
- iii) **eager versioning** iv) lazy versioning

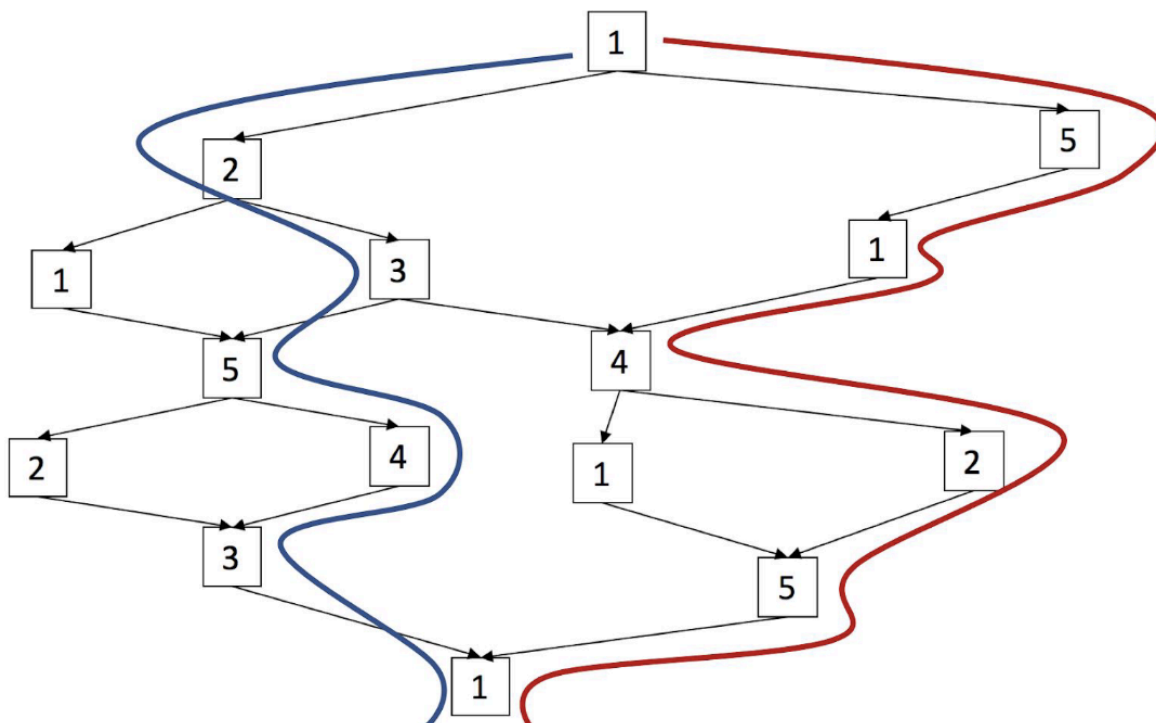
c) What locking discipline ensures deadlock freedom? [3 points]

The **locking discipline** that ensures deadlock freedom is the **global lock ordering** rule — all threads must acquire multiple locks in a consistent, predefined order (e.g., always acquire L1 before L2). This prevents circular wait conditions that cause deadlocks.

d) Select **all** statements that are **true** about a **ticket lock**. [5 pts]

- i) Ticket locks provide fairness (FIFO ordering).
- ii) Ticket locks are **non-blocking**.
- iii) Ticket locks need an **atomic operation** to acquire and release the lock.
- iv) Ticket locks incur **$O(P)$** interconnect traffic per lock acquire/release with **P** threads.
- v) Ticket locks can incur **$O(P^2)$** interconnect traffic under high contention when multiple threads spin on the same variable.

e) Calculate the work and depth of the following acyclic directed graph. Also, draw the critical path(s) in the system. [4 points]



Depth = 19

Work = 40

2) Amdahl's Law [5 pts]

You parallelized a data processing app and got a **6x speedup on 16 cores**. Profiling shows the remaining bottleneck is a **sequential code**, while the rest of the code **scales perfectly**.

a) Based on the measured 6x speedup on 16 cores, what percentage of the original execution time was sequential? Show your work. [2 points]

$$6 = 6f + 3f/8 = 1$$

$$5 = 5.625f$$

$$f = 0.889$$

So parallel portion $\rightarrow 1 - f = 11.1\%$

b) You have budget for one optimization:

- Option A: Optimize the sequential bottleneck to run 2x faster, but stay on 16 cores
- Option B: Upgrade to 32 cores (speed of sequential code remains unchanged)

Calculate the speedup for each option. Which should you choose and why? [3 points]

$$\text{Option A: } 1 / ((1/9)/2 + (8/9)/16) = 9x$$

$$\text{Option B: } 1 / ((1/9) + (8/9)/32) = 7.2x$$

3) Cache Coherence [10 pts]

Consider the following multithreaded C code that runs on a 4-core system with an MSI snooping cache coherence protocol. Each core has a private 32KB L1 cache with 64-byte cache lines. Assume the cache is write-back and uses a write-allocate policy.

```
#define NUM_THREADS 4
#define ITERATIONS 1000000

struct counter_t {
    long count;           // 8 bytes
    long account_info [5]; // 40 bytes
} counters[NUM_THREADS];

void worker(int id) {

    for (int i = 0; i < ITERATIONS; i++) {
        counters[id].count++;
    }

    return NULL;
}
```

a) Even though this program is **data-parallel**—each thread updates a different counter—it does **not scale well** with more threads.

Identify a coherence protocol related overhead. Indicate which shared variables cause the problem. [3 pts]

Even though the program is data-parallel, it suffers from **false sharing** in the `counters` array. Each `counter_t` is 48 bytes, so multiple counters share a cache line. As different threads update nearby counters, this causes **unnecessary coherence traffic**, hurting scalability.

b) Assume **Thread 0** (Core 0) and **Thread 1** (Core 1) run concurrently, each executing `counters[id].count++` at nearly the same time.

Thread 0 performs its increment **first**, followed immediately by Thread 1.

Initially, both cores have the cache line containing `counters` in the **Invalid (I)** state.

Trace the coherence traffic for one loop iteration and **fill in the table below**. [4 pts]

Event	Bus Message(s)	Coherence state (Core 0)	Coherence state (Core 1)
Initial State	-	I	I
Core 0: counters[0].count++	Message: _____ Sent by: _____ Message: _____ Sent by: _____		
Core 1: counters[1].count++	Message: _____ Sent by: _____ Message: _____ Sent by: _____		

c) Propose a simple change to the code to reduce the coherence overhead you identified in part (a). Your solution should ONLY modify the `counter_t` struct definition, and leave all other code unmodified. Briefly explain. [3 points]

```
struct counter_t {
    long count;           // 8 bytes
    long account_info[5]; // 40 bytes
    long padding[2];      // 16 bytes
} counters[NUM_THREADS];
```

Forces different `counter_t` elements to be on different cache lines.

4) Coherence Protocols [20 Points]

The **Xerox Dragon protocol** uses four states:

- **E (Exclusive)**: Block is only in this cache, clean.
- **M (Modified)**: Block is only in this cache, dirty.
- **Sc (Shared-clean)**: Block is shared by multiple caches; this cache is *not* the last writer.
- **Sm (Shared-modified)**: Block is shared; this cache is the *last writer* and must update memory on eviction.

Dragon is a **write-update** protocol—when a block is written, other caches' copies are **updated**, not invalidated.

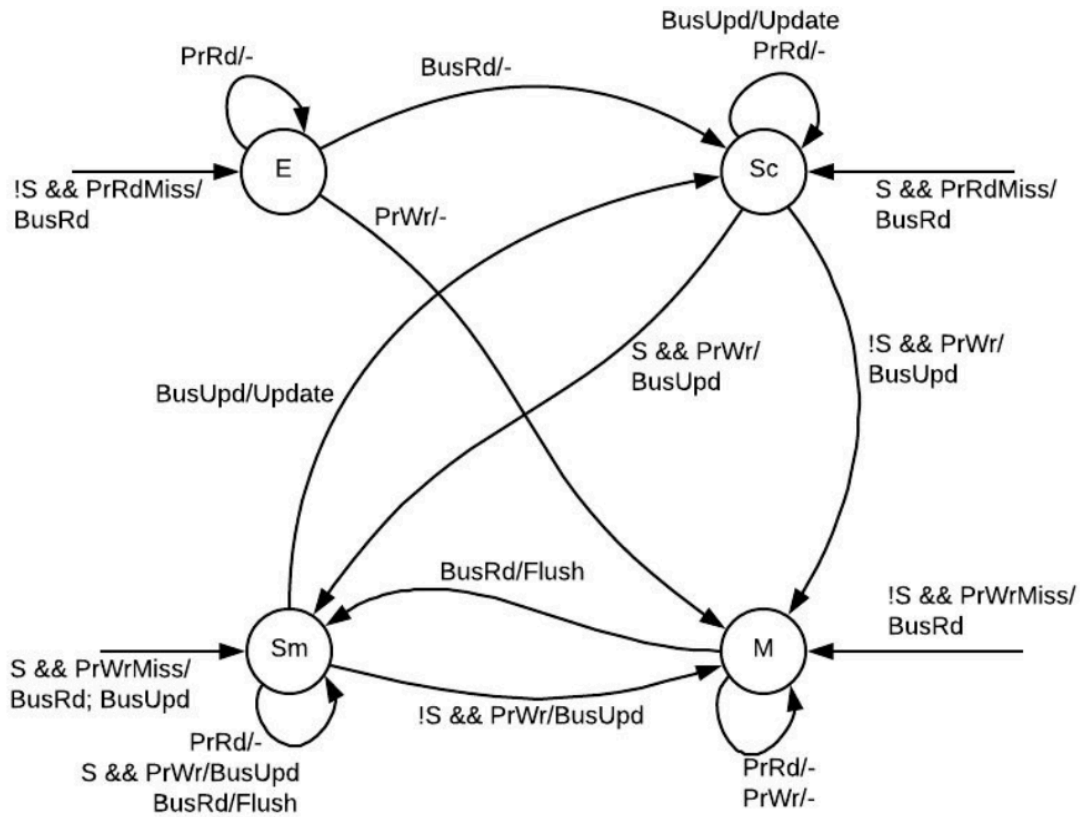
- a) Complete the following **state transition diagram** for the Dragon protocol using only the given action/reaction symbols. [15 pts]

- Three transitions are already labeled; fill in the remaining ones.
- You may add up to **2 extra edges**.
- Do **not** add states or transient states.
- Superfluous edges incur a **-1 point** penalty.

Use **S** to mark transitions when sharers exist, and **!S** when no sharers exist. Here, **S** refers to a **hardware “shared” line** connected to all processors, which indicates whether a block being broadcast on the bus is cached by more than one processor.

Event	Description
PrRd	Processor Read
PrWr	Processor Write
PrRdMiss	Processor Read Miss
PrWrMiss	Processor Write Miss
BusRd	Bus Read
BusUpd	Bus Update
BusReply	Bus Reply
Update	Update own cache block
Flush	Place the block on the bus and write to memory

The solution is as follows:



We will accept solutions where a BusReply is used instead of a flush.

b) The Dragon protocol allows the cache blocks in the Sc state to be replaced silently without any bus activity. What optimization to the coherence protocol can be made if a broadcast was made to let other caches know that a Sc block is being replaced? **[5 points]**

Other caches could test the shared line and move to the E state if there are no other sharers.

5) Vector Processing [10 pts]

Consider the following scalar code:

```
int indices[16];
int values[16], result[16];

for (int i = 0; i < 16; i++) {
    int idx = indices[i];
    result[idx] = result[idx] + values[i] * 2;
}
```

Consider the following vector instruction set. Each element is an `int` (32-bit integers).

Instruction	What it does	Example
VLD	Load 4 consecutive elements from memory.	VLD V1, [A+i] → loads A[i..i+3]
VST	Store 4 consecutive elements to memory.	VST V1, [A+i] → stores V1 → A[i..i+3]
VGATHER	Load 4 elements using an index list (indirect access).	VGATHER V2, [A], V1 → loads A[V1[0..3]]
VSCATTER	Store 4 elements using an index list.	VSCATTER V2, [A], V1 → stores V2[i] → A[V1[i]]
VMUL	Multiply each of the 4 elements by a number.	VMUL V3, V1, #2 → doubles each element
VADD	Add two 4-element vectors element-by-element.	VADD V4, V1, V2 → V4[i] = V1[i] + V2[i]

Vector registers: V0–V5

a) For the following access, specify whether the access pattern is contiguous or irregular. Also, specify the vector instruction you would use. [2 pts]

Access	Access Pattern	Vector Instruction
<code>indices[i]</code> (read)	Contiguous	VLD
<code>values[i]</code> (read)	Contiguous	VLD
<code>result[idx]</code> (read)	Irregular	VGATHER
<code>result[idx]</code> (write)	Irregular	VSCATTER

b) Show the vectorized loop body for processing 4 elements per iteration. [3 pts]

```
for (int i = 0; i < 16; i += 4) {
    // 1) Load 4 consecutive indices[i..i+3]
    __VLD_____ V0, [indices + i]

    // 2) Gather 4 old result values using indices
    __VGATHER_____ V1, [result], V0

    // 3) Load 4 consecutive values[i..i+3]
    __VLD_____ V2, [values + i]

    // 4) Multiply each by 2
    __VMUL_____ V3, V2, #2

    // 5) Add old result + doubled values
    __VADD_____ V4, V1, V3

    // 6) Scatter back to result using indices
    __VSCATTER_____ V4, [result], V0
}
```

c) Calculate the speedup of the vectorized version compared to the scalar version for the 16-element array. Show your work. [5 pts]

Use these simple costs:

- **Vector and scalar load/store:** 10 cycles
- **Vector and scalar arithmetic (mul/add):** 4 cycles
- **Scatter and gather:** 40 cycles
- **Loop overhead:** 3 cycles per iteration (scalar and vector)

Scalar: $(3 \text{ loads} + 1 \text{ mul} + 1 \text{ add} + 1 \text{ store} + \text{loop overhead}) * 16 \text{ iterations} = 51 * 16 = 816$

Vectorized: $(2 \text{ VLDs} + 1 \text{ VGATHER} + 1 \text{ VMUL} + 1 \text{ VADD} + 1 \text{ VSCATTER} + \text{loop overhead}) * 4 \text{ iterations} = 111 * 4 = 444$

Speedup = $816 / 444 = 1.838x$

6) Transactional Memory [15 pts]

Initial state: $X = Y = 0$

Transaction T1	Transaction T2
I1: Read X I2: Write Y = 1 I3: Read Y	I4: Read Y I5: Write X = 1 I6: Read X

- a) Is there a conflict between T1 and T2? If so, identify all pairs of conflicting instructions. [2 pts]

(I1, I5): T1 reads X, T2 writes X → Read-Write conflict on X

(I2, I4): T1 writes Y, T2 reads Y → Write-Read conflict on Y

- b) Transactions guarantee serializability. That is, all transactions can be ordered such that they appear to execute one after another in a single core (total order).

Assume T1 and T2 are executed concurrently on two cores starting at the same cycle 0. Assume that each core can execute one memory access in each cycle.

Assume a transactional memory system that uses eager versioning and lazy conflict detection.

Create a serializable order for instructions in T1 and T2 that takes the least number of cycles. (Hints: An execution of T1 and T2 can conflict, but still be serializable.

There exists a solution that takes fewer than 6 cycles). [5 pts]

I4 reads I2's write value for Y. This 5-cycle execution is serializable as it appears as if T1 executed before T2.

You can create a schedule with 4 cycles, if you reorder I1 and I2 within T1.

Cycles	Transaction T1	Transaction T2
0	I1: Read X	
1	I2: Write Y = 1	
2	I3: Read Y	I4: Read Y
3		I5: Write X = 1
4		I6: Read X

- c) If two concurrent transactions execute conflicting memory accesses, a conventional system would abort one of them. While this is sufficient constraint, it is not necessary to ensure serializability of transactions.

Say your runtime system can keep track of the data dependencies (RAW, WAR, WAW) between two or more concurrent transactions. Determine a less restrictive constraint that the system could check between those dependencies to determine that the execution of transactions is NOT serializable. It should be less restrictive than checking for any conflict. Illustrate with an example that is serializable but not conflict-free. [5 pts]

Create an execution graph where each node represents an executed transaction. Add a directed edge between two transactions to represent any data dependencies observed between them. An execution is serializable iff there are no **cycles in the graph**.

Example (serializable but not conflict-free):

T1: W(A) ; W(B)

T2: R(A) ; R(B)

Say T1 and T2 executed concurrently, but the reads in T2 returned T1's write values.

Conflicts: $T1.W(A) \rightarrow T2.R(A)$ and $T1.W(B) \rightarrow T2.R(B)$ give edges **T1 → T2** (twice), no edge back; graph is acyclic $\Rightarrow$ serializable (equivalent to T1 then T2) despite conflicts.

- d) Give a simple code example with two transactions that can deadlock. [3 pts]

```
int flag1 = 0, flag2 = 0;
```

Transaction T1	Transaction T2
begin_tx	begin_tx
flag1 = 1	flag2 = 1
while(!flag2) ;	while(!flag1) ;
end_tx	end_tx

7) Consistency Models [20 pts]

- a) Which of the following memory ordering constraints does in-window speculation help mitigate the overhead of? Circle all that apply. [4 pts]

Load -> Load	Load -> Store	
Store -> Store	Store -> Load	
fence -> Load	fence -> Store	(fence is full fence)
Load -> fence	Store -> fence	

- b) Is it possible to guarantee sequentially consistent (SC) execution on an x86 processor? If so, how? [2 pts]

Yes. The compiler inserts a fence before and after every memory access.

- c) Select True or False.

+1 for correct. -1 for incorrect answer. No penalty for blank. [5 pts]

- i) An SC execution guarantees data-race-freedom

True / False

- ii) SC guarantees that there is only one possible order of execution for memory accesses in a program

True / False

- iii) Data-race-free program guarantees that there is only one possible order of execution for memory accesses in that program

True / False

- iv) Function inlining optimization in the compiler cannot violate SC. True / False

- v) Modern concurrent languages such as C++/Java guarantee x86 TSO. True / False

- d) Which of the memory models allows the following program behavior? Select all that apply. [2 pts]

```
int A = B = 0;
```

Thread T1	Thread T2
<pre>A = 1; print (B);</pre>	<pre>B = 1; print (A);</pre>

Output: 0, 0

- a) Sequential Consistency
- b) Total Store Order
- c) Relaxed Consistency
- e) Consider the following program. Assume function `incr()` is invoked concurrently in two threads.

```
int balance = 0;
lock l = UNLOCK;
int flag = 0;

int incr(int x)
{
    int tmp = 0;

    lock(l)
        tmp = balance;
    unlock(l)

    lock(l)
        tmp = tmp + x;
        balance = tmp;
    unlock(l)

    flag = 1;
}
```

i) Identify data-races in this code, if any. Circle the pair of memory accesses that constitute a data-race. Also, indicate whether it would result in an incorrect execution under SC. [3 pts]

There is a data-race between writes "flag =1" when incr() is executed concurrently in two or more threads. This is still a valid execution under SC.

ii) Is there a bug in this code if the programmer wants to ensure that updates to balance are correct. If so, identify the bug. Would using transactions instead of locks avoid the bug? [4 pts]

Locks in the program don't guarantee atomicity between read and update to balance.

Substituting lock with tx_begin and unlock with tx_end would NOT resolve this bug.

Read and write to balance should be part of the same critical section. You can use transaction or locks to guarantee that critical section.

[Work sheet]