

Name: _____ Uniqname: _____

EECS 570 Winter 2022 Midterm Exam - **SOLUTIONS**

Total points: 100

Total time: 80 minutes

Closed book, closed notes.

Calculators are allowed, but no PDAs, Portables, Cell phones, etc.

For calculation questions, show your work and clearly indicate your final answers.

Use pen or a dark pencil.

Do not spend too much time on any one question.

State any reasonable assumptions that you need to make.

There are 11 pages in this exam. Please make sure that you have all pages.

Please try to write all your answers in the assigned spaces.

If you really need extra space, you may use the backs of pages (but please **clearly** indicate that you are doing so).

Please sign the Honor Code:

I have neither given nor received unauthorized aid on this examination, nor have I concealed any violations of the Honor Code.

Q1	/10
Q2	/15
Q3	/20
Q4	/20
Q5	/15
Q6	/20
Total	/100

1 Amdahl's Law (10 points)

Consider a program where 20% of the computation must run serially, while the remaining 80% of the computation is parallelisable.

- (a) (5 points) What is the speedup gained by running this program on a processor with 8 cores?

$$S = \frac{1}{(1 - f) + \frac{f}{p}} = \frac{1}{0.2 + \frac{0.8}{8}} = 3.33x.$$

- (b) (5 points) If we assume that the parallel portion of the program is “embarrassingly parallel” (i.e., its performance scales linearly in the number of cores for an arbitrary number of cores), what is the maximum possible speedup we can gain by parallelising this program?

$$S = \frac{1}{(1 - f) + \frac{f}{\infty}} = \frac{1}{1 - f} = \frac{1}{0.2} = 5x.$$

2 Shared Memory and Message Passing (15 points)

- (a) (8 points) Is it possible to implement a message passing system on a shared memory architecture? Why or why not? Briefly explain your answer.

Answer: Yes, it is possible. One can use shared memory to implement mailboxes and message queues for each processor, and construct APIs for sending and receiving messages that processors can use. The implementation of these APIs can route and copy messages to and from the mailboxes and queues of various processors in accordance with message passing semantics. In addition, one could modify the OS to forbid the sharing of memory between processes, and enforce that all message passing programs must use one process per core (i.e., no threads). This would make the programming model quite similar to that of a regular message passing system.

- (b) (7 points) Is it possible to implement a shared memory abstraction on top of a message passing architecture? Why or why not? (Assume that processes/threads in the message passing system do not share any address space.) Briefly explain your answer.

Answer: Yes, it is possible. Message passing can be used to implement the equivalent of a coherence protocol (perhaps on a per-page basis) among the various processors. APIs can be added to load and store data from “shared” memory addresses. The implementation of these APIs can implement the equivalent of the coherence protocol for those addresses. This will give the semantics of shared memory for such accesses.

3 GPUs, Costs, and a Dash of Datacenters (20 points)

Answer the following questions. **Be sure to briefly explain your answers.**

- (a) (7 points) Why can branches in GPU code be problematic?

Answer: GPUs execute the same instructions on multiple data elements as a set of threads, in lockstep. If there is a branch in the code of the threads, and some threads take the branch and others do not, both paths of the branch are then executed sequentially. Specifically, threads that do not take a branch path basically stall while waiting for threads that do take that path to finish it. Since the threads that execute one path of the branch must wait for those that execute the other path of the branch, overall performance is reduced.

- (b) (6 points) Under what circumstances will an n-processor parallel system be more **cost-effective** than using n uniprocessor systems?

Answer: The parallel system will be more cost-effective when its speedup over the uniprocessor system is greater than the “costup” with respect to the uniprocessor system, where costup is defined as

$$\text{costup} = \frac{\text{cost}(\text{n-processor parallel system})}{\text{cost}(\text{uniprocessor system})}$$

Name: _____

Uniqname: _____

- (c) (7 points) Is there a single optimal architecture for datacenter hardware, or are different architectures optimal for different performance (QoS) requirements and different request distributions? (“Request distributions” here refers to the fraction of requests that are “big”/“small” requests, etc.)

Answer: No, there is no single optimal architecture for datacenter hardware. Different architectures will be optimal for different scenarios. For instance, if QoS requirements are very high but request frequency is not that high and individual requests are largely “big”, then a few big cores may perform better than many small cores. On the other hand, if all requests are “small” and QoS requirements are high and requests are extremely frequent, an architecture of many small cores will likely be better than a few big cores.

4 Cache Coherence (20 points)

- (a) (10 points) Consider a MESI snooping coherence protocol where each core has its own private cache. Assume that the memory controller responds to read requests by either providing the line to the requestor with exclusive permissions (E state) or without exclusive permissions (i.e., S state), depending on its knowledge of the current state of the system. In other words, it is the responsibility of the memory controller to decide whether the requestor gets the line in E state or S state. The protocol does not have clean eviction notifications for lines in S.

Is it possible to have an execution where at some point, no private caches have the line for a given address x , a private cache then requests the line, and the line is provided to it in S state (as opposed to E)? If yes, clearly describe how this could happen in an execution. If no, clearly explain why such a scenario is impossible.

Answer: Yes, the scenario is possible. Consider the following execution:

- i. Core 0 makes a read request for address x and gets it in E. The memory controller goes to the EorM state.
- ii. Core 1 makes a read request for x , core 0 forwards the data to it, and both cores and the memory controller go to S.
- iii. Core 0 and core 1 evict their lines for x .
- iv. Core 2 makes a read request for x and gets the line in S, because the memory controller was still in the S state.

- (b) (5 points) Give one difference between typical snooping coherence protocols and typical directory coherence protocols.

Answer: There are multiple valid answers to this question. One example answer follows.

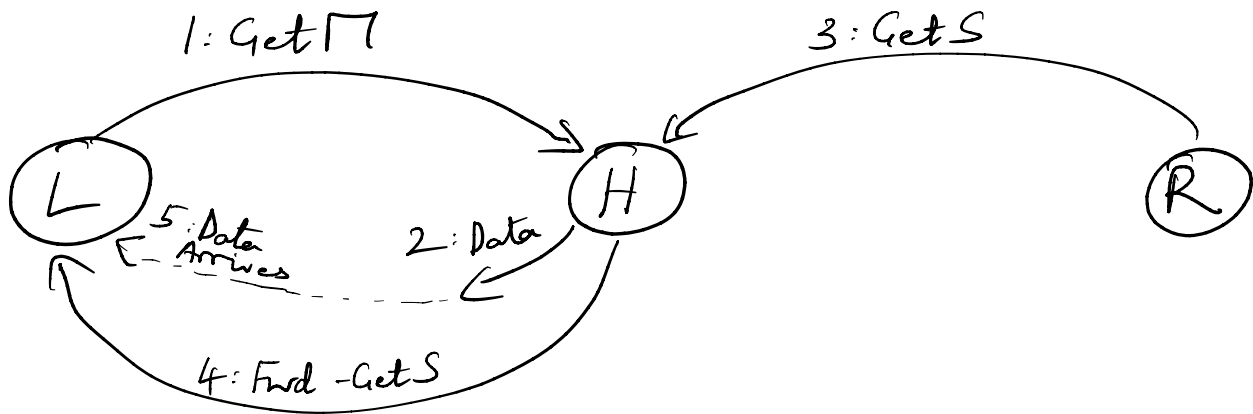
Snooping coherence protocols do not store list(s) of sharers at centralised locations, while directory coherence protocols store lists of sharers for each cached line at the directory/directories.

Name: _____

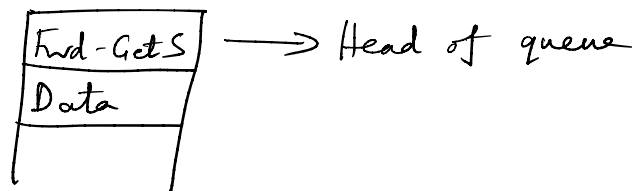
Uniqname: _____

- (c) (5 points) Give an example of a coherence protocol deadlock that could occur in a naively designed coherence protocol due to network and/or buffering issues. The protocol you use in your example must be an MSI protocol.

Answer: There are multiple answers to this question. An example answer is the following:
Consider the following execution where events occur in numerical order. Also assume that each core has only one input queue.



If events occur in the order mentioned above, then L's input queue would look like this:



L would stall on the Fwd-GetS until the data arrived, but the Data is blocked by the Fwd-GetS and will never arrive if L keeps stalling on the Fwd-GetS. Meanwhile core R would be waiting forever for the Data in order to finish its load. The result is deadlock.

5 Transactional Memory (15 points)

Consider the following lock-based program where each thread runs the code below. Assume that the function `threadId()` returns the thread number of the thread that calls it. So it will return 0 for thread 0, 1 for thread 1, so on and so forth. `a1` and `a2` are arrays of integers. `a1Len` contains the length of array `a1`, and `a2Len` contains the length of array `a2`. The function `process(..)` both reads and writes each of its arguments. The threads enforce mutual exclusion using lock `l1`. **Please read the code carefully!**

```
01: lock(l1);
02: for (i = threadId(); i < a1Len - 1; i += 8) {
03:     process(a1[i], a1[i+1]);
04: }
05:
06: for (i = threadId(); i < a2Len; i += 8) {
07:     a2[i] = a1[i];
08: }
09: unlock(l1);
```

- (a) (7 points) Convert this lock-based code to code that uses only transactions. Your transaction-based code should only be able to generate exactly the same possible results that could have been generated by the original lock-based code. In other words, no new results should become possible, and no previously allowed results should become forbidden.

```
01: begin_transaction();
02: for (i = threadId(); i < a1Len - 1; i += 8) {
03:     process(a1[i], a1[i+1]);
04: }
05:
06: for (i = threadId(); i < a2Len; i += 8) {
07:     a2[i] = a1[i];
08: }
09: end_transaction();
```


Name: _____

Username: _____

- (b) (8 points) Assume that your transactional code runs on a hardware transactional memory system where each core has its own private cache which has 64-byte cache lines. Assume that integers are 64 bits (8 bytes) in size, and that arrays `a1` and `a2` can fit in a given core's private cache at the same time. The hardware transactional memory uses the coherence protocol (as is typical) to detect conflicts between transactions. In such a system, will the transactional memory code be faster, roughly as fast as, or a lot slower than the lock-based code? Explain your answer.

Answer: The transactional code will run roughly as fast as the lock-based code. Even though there exist transactions that access disjoint sets of array elements, conflict detection is done at a cache block granularity (since conflicts are detected by the coherence protocol). As a result, each transaction will end up conflicting with virtually all other transactions, and all the transactions will need to be serialized—which is exactly what happens in the lock-based version (where all threads share a single lock). The performance of the transactional version may even be slightly worse than the lock-based version due to latency added by transactions beginning execution and then having to roll back.

6 Synchronization (20 points)

(20 points) You are tasked with implementing a highly concurrent queue data structure that can be accessed in parallel by multiple threads **without locks**. The queue stores its entries as a singly-linked list with a head and tail pointer. Entries are added (enqueued) to the tail of the list and removed (dequeued) from the head of the list. The structure of the linked list nodes and the list's enqueue function are as follows. (In the code below, `fetch_and_store(A, B)` stores the value of B at the memory location A and returns the old value at memory location A, executing these operations atomically. So for instance, `fetch_and_store(tail, I)` makes `tail` point to I and returns the old value of `tail`.)

```
01: struct node {
02:     node *next;
03:     void *data;
04: }
05:
06: node *head = NULL;
07: node *tail = NULL;
08:
09: void enqueue(node *I) {
10:     if (I == NULL)
11:         return;
12:
13:     I->next = NULL;
14:     pred = fetch_and_store(tail, I);
15:     if (pred == NULL) {
16:         head = I;
17:     } else {
18:         pred->next = I;
19:     }
20: }
```

Implement the list's dequeue function **on the next page**. The dequeue function should remove the head of the list and return a pointer to it, updating the list's head and tail pointers appropriately. If the queue is currently empty, the dequeue function should spin until the queue becomes non-empty. **You may assume that at most one instance of the dequeue function is ever running at any time (similar to how at most one thread can ever be releasing an MCS lock at any time)**. However, multiple enqueueers may be running the enqueue function at the same time, and multiple enqueueers may overlap execution with an instance of your dequeue function. The code must run correctly in all such cases.

Remember, **you may not use locks in your dequeue function**. However, you can use a compare-and-swap (CAS) instruction, whose semantics are as follows:

`CAS(A, B, C)` compares the value of A with the value of B. If they match, it stores C at address A and returns true. Otherwise it returns false. So for example, `CAS(tail, head, NULL)` checks if `tail` and `head` point to the same address, and if they do, it sets `tail` to NULL and returns true. If `tail` and `head` point to different addresses, it returns false. The CAS instruction executes atomically.

Name: _____

Uniqname: _____

```
01: node *dequeue() {
02:     while (!head) {}
03:
04:     cur = head;
05:
06:     if (head->next == NULL) {
07:         if (CAS(tail, head, NULL)) {
08:             CAS(head, cur, NULL);
09:             return cur;
10:         } else {
11:             while (head->next == NULL) {}
12:         }
13:     }
14:
15:     head = head->next;
16:     return cur;
17: }
```