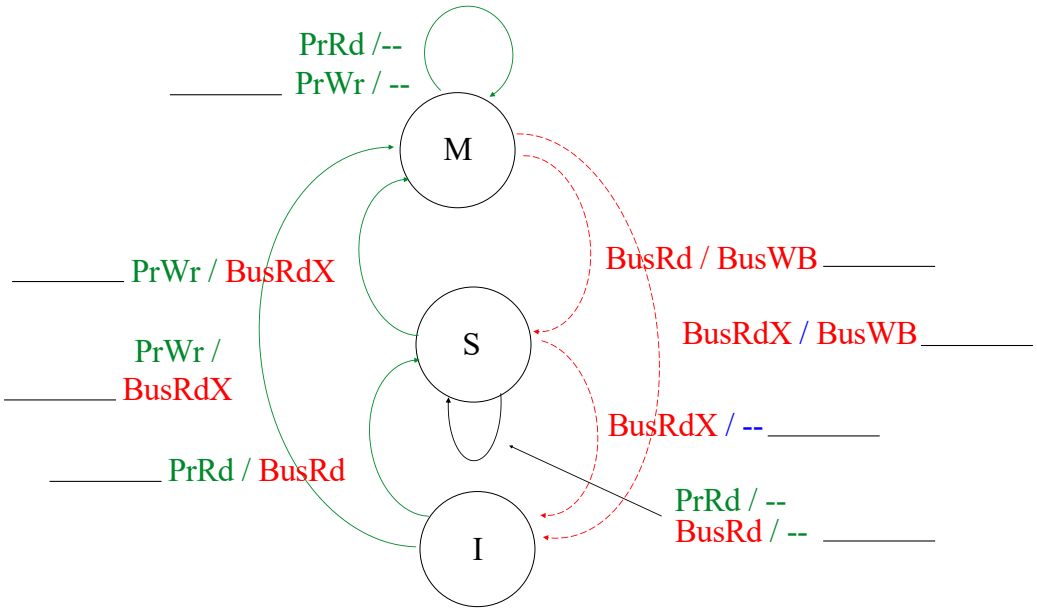
**Stanford CS149: Parallel Computing**
**Written Assignment 4**

**MSI Coherence Protocol Warmup**

## Problem 1: (Graded for Correctness - 25 pts)

Below is a state diagram for the MSI protocol.




Consider the follow sequence of operations by processors 1 and 2. Assume that each processor has a local cache carrying out the MSI protocol. Each cache can store two cache lines of data (there's room for both X and Y). For simplicity, please assume that the variables X and Y take an entire cache line.

Please fill out the table below, which indicates the state of the lines X and Y in the local caches of P0 and P1 after each operation. We've given the first four rows for you as examples.

| Operation | P0 X state | P1 X state | P0 Y state | P1 Y state |
|---|---|---|---|---|
| P0 LOAD X | S [MISS] | I | I | I |
| P0 LOAD Y | S | I | S [MISS] | I |
| P1 LOAD Y | S | I | S | S [MISS] |
| | | | | |
| P1 LOAD Y | | | | |
| P1 STORE Y | | | | |
| P0 STORE Y | | | | |
| P0 LOAD X | | | | |
| P1 STORE X | | | | |
| P0 LOAD X | | | | |

**Load Linked / Store Conditional and Cache Coherence**

## Problem 2: (Graded for Correctness - 25 pts)

A common set of instructions that enable atomic execution is load linked-store conditional (LL-SC). The idea is that when a processor loads from an address using a `load_linked` (LL) operation, the corresponding `store_conditional` (SC) to that address will succeed **only if no other writes to that address from any processor have intervened**. If the SC succeeds the LL and SC have been executed atomically.

Note that unlike `test_and_set` or `compare_and_swap`, which are single atomic operations, load linked and store conditional are two different operations... **each is atomic on its own, but the processor may execute other instructions in between a LL and a later SC.** Pseudocode for these instructions is given below.

```
int load_linked(int* addr) {
  return *addr;
}

// atomically perform this sequence
bool store_conditional(int* addr, int new_val) {
  if ( \* data in addr has not been written to by any processor *\
       \* since the last load_linked on addr                    *\ ) {
    *addr = new_val;
    return true;
  } else {
    return false;
  }
}
```

A read-write lock has the property that multiple threads may hold the read lock, but **only one thread may hold the write lock. If a thread is holding the write lock, no other thread may hold a read lock.** Here is a simple implementation for read-write locks using the LL and SC primitives (it should remind you of the way invalidation-based cache coherence works).

```
struct read_write_lock {
  int is_write_locked;
  int num_readers;
};

void write_lock(read_write_lock *l) {

  // Need to make sure there are no other writers.
  // This also prevents new readers
  while (true) {
      if (load_linked(&l->is_write_locked) == 0 && store_conditional(&l->is_write_locked, 1))
      break;
  }

  // wait until all the readers are clear
  while (l->num_readers > 0);

  // now we have the writer lock and can proceed
}

void write_unlock(read_write_lock *l) {
  l->is_write_locked = 0;
}
```

Here is the implementation of a read-lock:

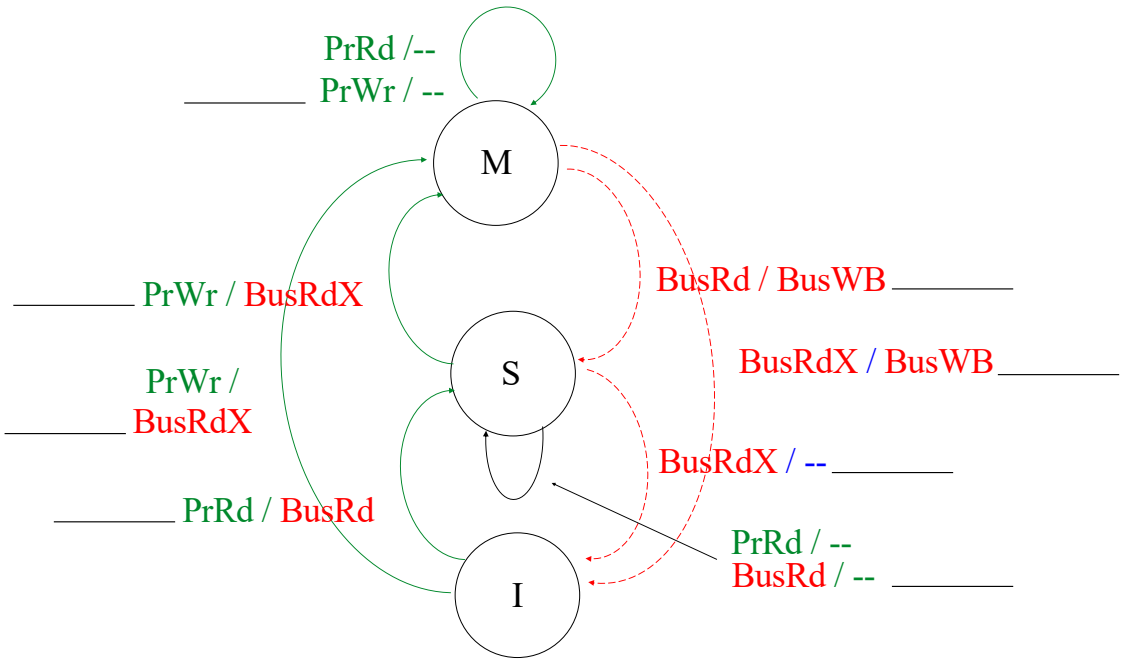
```
void read_lock(read_write_lock *l) {

  // need to make sure there is not a current writer
  while (true) {
    if (load_linked(&l->is_write_locked) == 0 && store_conditional(&l->is_write_locked, 1))
      break;
  }

  // atomic increment count
  while (true) {
    int val = load_linked(&l->num_readers);
    if (store_conditional(&l->num_readers, val+1))
      break;
  }

  l->is_write_locked = 0;
}

void read_unlock(read_write_lock *l) {

  // atomic decrement count
  while (true) {
    int val = load_linked(&l->num_readers);
    if (store_conditional(&l->num_readers, val-1))
      break;
  }
}
```

A. (5 pts) What is the key difference between the way that read-write locks are implemented above and the way that an invalidation-based cache coherence protocol like MSI works? Specifically think about the approach used to ensure readers and writers don't have access to the same data at the same time. Give two reasons why a read-write lock style implementation of cache coherence would be difficult?

B. (20 pts) The table below represents the memory accesses from three processors executing the `read_unlock` function for a lock variable. Fill in the rest of the table. Assume that at each step every processor executes an LL, an SC, or nothing. Assume that the bus transaction represents the set of "compatible" bus transactions from all processors. If bus transactions conflict, the processor with the lowest number will win and places its transaction on the bus.

| P0 | P1 | P2 | Bus transactions | Cache States | Data comes from |
|---|---|---|---|---|---|
| LL | LL | LL | P0:RD, P1:RD, P2:RD | P0:S, P1:S, P2:S | Memory |
| SC | SC | SC | | | |
| | LL | LL | | | |
| | SC | SC | | | |
| | | LL | | | |
| | | SC | | | |

The MSI coherence protocol is given below as a student aid:



## Problem 3: (Graded on Effort Only - 25 pts)

A. (10 pts) In the lecture on implementing locks/fine-grained synchronization we talked about a basic lock implementation like this:

```
void lock(int* lock) {
   while (CAS(lock, 0, 1) == 1) {}
}

void unlock(int* lock) {
   *lock = 0;
}

// As a reminder CAS() performs this logic atomically
int CAS(int* addr, int compare, int val) {
   int old = *addr;
   *addr = (old == compare) ? val : old;
   return old;
}
```

Consider a situation where many threads, each running on a different core in a system that **implements cache coherence using the MSI protocol**, are attempting to acquire the lock. Please describe why it can be the case that the processor that is executing the thread that **is holding the lock IS NOT the processor whose cache holds the cache line containing the variable `lock` in the M state.** Please keep in mind that a `CAS()` is always a write operation from the perspective of cache coherence.

B. (15 pts) Consider the following code, where a lock, implemented using compare and swap (CAS) is used to make the operation of incrementing the variable x atomic.

```
void lock(int* l) {
  while (CAS(l, 0, 1) == 1);
}

void unlock(int*) {
  *l = 0;
}

int x;  // shared counter variable
int l;  // lock variable

// per-thread code
lock(&l);
x = x + 1;
unlock(&l);
```

Imagine this code running on a system that relaxes both WRITE AFTER WRITE and READ AFTER WRITE memory orderings. Consider the case where x is initialized to 0, and both thread 1 and thread 2 attempt to atomically increment x using the code above. Assume thread 1 acquires the lock first. Why is it possible for thread 2 to observe that x=0 when it later acquires the lock and enters the critical section. (You may assume that each invocation of CAS is treated as a write by the coherence protocol.)

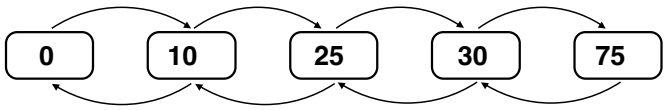
**Two threads + a doubly Linked List = trouble**

## Problem 4: (Graded on Effort Only - 25 pts)

Consider a **SORTED doubly-linked list** that supports the following operations.

- `insert_head`, which inserts a node by traversing the list starting from the head.
- `insert_tail`, which inserts a node by traversing the list **backwards starting from the tail**.

Insertions are the ONLY OPERATIONS on the data structure, and there are NO DELETES. Furthermore, you can assume that only two threads will ever be operating on the data structure at the same time, thread 1 is calling `insert_head`, and thread 2 calling `insert_tail`. (NOTE: these simplifications are important!)




Code for `insert_head` is given below. You can assume the code for `insert_tail` is similar. **NOTE THERE ARE NO LOCKS!**

```
struct Node {
  int value;
  Node* next, *prev;
};

void insert_head(Node* head, int value) {
  Node* n = new Node;
  n->value = value;

  Node* cur = head;

  // ignore insert at front of list case
  while ( /* position in list not found */ ) {

    if (n->value > cur->value &&
        n->value <= cur->next->value) {

      // link the new node forward/backward
      n->next = cur->next;
      n->prev = cur;

      cur->next->prev = n;  // link next back to n
      cur->next = n;        // link cur forward to n
      return;
    } else {
        cur = cur->next;    // step forward
    }
  }
}
```

**Questions are on the next page...**

A. (6 pts) Consider the case where thread 1 calls `insert_head(8)` and thread 2 concurrently calls `insert_tail(27)`. **Does the code generate correct output for all instruction interleavings?** Explain why. If not, draw one possible incorrect data structure that results and briefly describe the interleaving that causes this result.

B. (6 pts) Consider the case where thread 1 calls `insert_head(27)` and thread 2 calls `insert_tail(26)`. **Does the code produce correct output for all instruction interleavings?** Explain why. If not, draw one possible incorrect data structure that results and briefly describe the interleaving that causes this result.

C. (6 pts) Consider a solution where threads use a 1-2-1 hand-over-hand locking strategy similar to the one discussed in class. (The thread grabs the lock for the next node it is traversing to in the list, so it holds two locks at once, and then releases the lock for the node it is leaving behind.) Assume that when inserting into the list, threads must hold a lock for the node before and after the future newly inserted node. Like in part C, consider the case where thread 1 calls `insert_head(27)` and thread 2 calls `insert_tail(26)`. What problem can happen now that the threads grab locks? Please describe the state of thread 1 and thread 2 that causes the problem (e.g., what locks are the threads holding?).

D. (6 pts) Imagine that you had a function `trylock(Lock* l)` that locks the lock `l` if the lock is free, **but immediately returns false if the lock is not free.** (It does not block until the lock is acquired like a normal call to `lock(l)`.) Give an explanation of an implementation of `insert_head` that uses `trylock` to solve the problem you identified in the previous problem. To get full credit we require that your implementation must:

- Hold two locks: one for the node prior to and after new node `n`, when inserting `n`.
- Cannot perform repeated iteration through the list (e.g, it cannot restart from the beginning of the list if it fails to get the required locks)
- YOU DO NOT NEED TO WORRY ABOUT LIVELOCK.

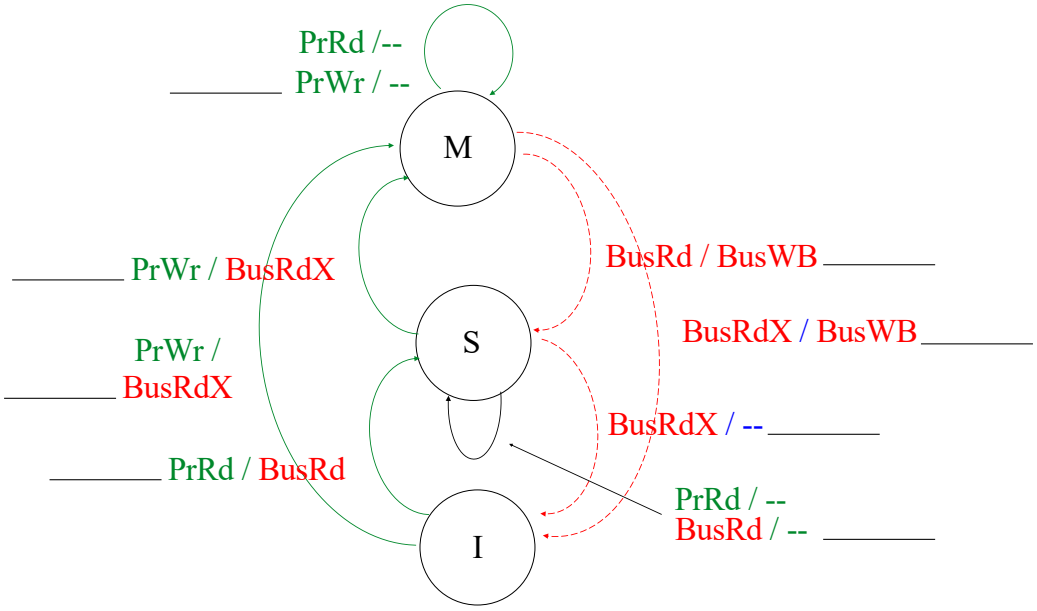
**Hint 1: keep in mind there are only inserts on the data structure, so a pointer cannot "disappear" out from underneath a thread. Hint 2: recall that if a thread has no locks on any nodes it has no guarantee what changes have been made to the data structure since it last checked. Nodes might have been added after it!**

Your answer can be in pseudo code or words, just be clear about specifically what it does, and address all important cases.

**MSI Coherence Protocol (Another Example)**

## PRACTICE PROBLEM 1:

Below is a state diagram for the MSI protocol.




Consider the follow sequence of operations by processors 1 and 2. Assume that each processor has a local cache carrying out the MSI protocol. Each cache can store two cache lines of data (there's room for both X and Y). For simplicity, please assume that the variables X and Y take an entire cache line.

Please fill out the table below, which indicates the state of the lines X and Y in the local caches of P0 and P1 after each operation. We've given the first four rows for you as examples.

| Operation | P0 X state | P1 X state | P0 Y state | P1 Y state |
|---|---|---|---|---|
| P0 LOAD X | S [MISS] | I | I | I |
| P0 LOAD X | S [HIT] | I | I | I |
| P1 STORE Y | S | I | I | M [MISS] |
| P1 STORE X | I | M [MISS] | I | M |
| P1 LOAD Y | | | | |
| P1 STORE Y | | | | |
| P1 LOAD Y | | | | |
| P0 STORE X | | | | |
| P0 STORE Y | | | | |

**Angry Students**

## PRACTICE PROBLEM 2:

Your friend is developing a game that features a horde of angry students chasing after professors for making long exams. Simulating students is expensive, so your friend decides to parallelize the computation using one thread to compute and update the student's positions, and another thread to simulate the student's angriness. The state of the game's N students is stored in the global array `students` in the code below).

```
struct Student {
   float position;   // assume position is 1D for simplicity
   float angriness;
};

Student students[N];

////////////////////////////////

void update_positions() {
   for (int i=0; i<N; i++) {
      students[i].position = compute_new_position(i);
   }
}

void update_angriness() {
   for (int i=0; i<N; i++) {
      students[i].angriness = compute_new_angriness(i);
   }
}

////////////////////////////////

// ... initialize students here

std::thread t0, t1;
t0 = std::thread(update_positions);
t1 = std::thread(update_angriness);
t0.join();
t1.join();
```

Questions are on the next page...

A. Since there is no synchronization between thread 0 and thread 1, your friend expects near a perfect 2× speedup when running on two-core processor that implements invalidation-based cache coherence **using 64-byte cache lines**. She is shocked when she doesn't obtain it. Why is this the case? (For this problem assume that there is sufficient bandwidth to keep two cores busy – "the code is bandwidth bound" is not an answer we are looking for.) HINT: consider how data is laid out in memory in a C struct.

B. Modify the program to correct the performance problem. You are allowed to modify the code and data structures as you wish, **but you are not allowed to change what computations are performed by each thread and your solution should not substantially increase the amount of memory used by the program.** You only need to describe your solution in pseudocode (compilable code is not required).

**Counting Fan Votes**

## PRACTICE PROBLEM 3:

Bad Bunny and Olivia Rodrigo team up to write a program that tallies fan votes for the best song on Bad Bunny's 2023 album "Nadie Sabe Lo Que Va a Pasar Mañana" (Nobody Knows What Is Going to Happen Tomorrow). Bad Bunny implements the following parallel code for generating per-song counts from an input array `votes`, where `votes[i]` indicates one vote for the votes[i]'th song on the album. (`votes[i] = 0` indicates a vote for the first song on the album). For simplicity, he limits his code to only tally votes for the first `NUM_SONGS` songs on the album.

Bad Bunny's code targets a small laptop with only two cores, each with a private 64 KB cache. **This machine features 64-byte cache lines (recall sizeof(int)=4 bytes) and uses the MSI invalidation-based cache coherence protocol.** Bad Bunny's implementation, which is carefully designed to use barriers to avoid the use of fine-grained locks, is given below:

```
int votes[N];                      // assume votes is initialized and N is a very large
int counts[NUM_SONGS];             // per-song counts
int partial_counts[2][NUM_SONGS];  // assume all counts are initialized to 0
                                   // assume partial_counts is 64-byte aligned.
                                   // (partial_counts[0][0] is at the start of a line)

////////////////////////// Code executed by thread 0 //////////////////////////
for (int i=0; i<N/2; i++)
  if (votes[i] < NUM_SONGS) // don't count vote if it's not in range
    partial_counts[0][votes[i]]++;

barrier();  // wait for both threads to reach this point

for (int i=0; i<NUM_SONGS; i++)
  counts[i] = partial_counts[0][i] + partial_counts[1][i];

////////////////////////// Code executed by thread 1 //////////////////////////
for (int i=N/2; i<N; i++)
  if (votes[i] < NUM_SONGS)  // don't count vote if it's not in range
    partial_counts[1][votes[i]]++;

barrier();  // wait for both threads to reach this point
```

**Questions begin on the next page...**

A. (4 pts) Bad Bunny runs his code on an input of 10 million fan votes (`N`=10,000,000) to compute tallies for the albums first eight songs. (`NUM_SONGS`=8). He is shocked when his program obtains far less than a linear speedup, and glumly looks at Olivia and says "Shoot, I got myself in trouble trying to avoid locks. I think I need to completely completely restructure the code to eliminate load imbalance." Olivia takes a look and says *"That's a bad idea, right? I think the simplest solution for improving performance is to just tally results for the first sixteen songs on the album instead (`NUM_SONGS=16`). Then you should expect near linear speedup."* Whose approach will lead to better performance on this computer? Why?

Circle one: Bad Bunny Olivia

Explain:

B. (8 pts) A few weeks later, Prof. Kayvon runs into Bad Bunny and says *"Hey Mr. Bunny, Olivia told me you have some code to count fan votes, and that you two fixed it up so it scaled linearly on two cores. I'd like to run my own little fan vote on my Twitter to see which of my CS149 lecture slides is the most popular. Can you send me that code?"* Prof. Kayvon runs a Twitter poll, but due to low popularity only collects 5,000 fan votes (`N=5,000`) for his set of 2000 CS149 slides (he renames `NUM_SONGS` to `NUM_SLIDES` in Bad Bunny's code). When Prof. Kayvon runs the code with `NUM_SLIDES=2000`, the speedup obtained by the code is nowhere near 2. Improve the existing code to scale near linearly with the larger number of slides. (Please provide pseudocode as part of your answer – **it need not be compilable C code, but it should clearly indicate where barriers are and what operations are performed by thread 0 and thread 1.**)

```
float votes[N];                      // assume input is initialized and N is a very large
int   counts[NUM_SLIDES];            // output counts
int   partial_counts[2][NUM_SLIDES]; // assume partial_counts are initialized to 0
                                     // assume partial_counts is 64-byte aligned.

////////////////////////// Code executed by thread 0 //////////////////////////
```

```
////////////////////////// Code executed by thread 1 //////////////////////////
```

## PRACTICE PROBLEM 4:

Consider the following program.

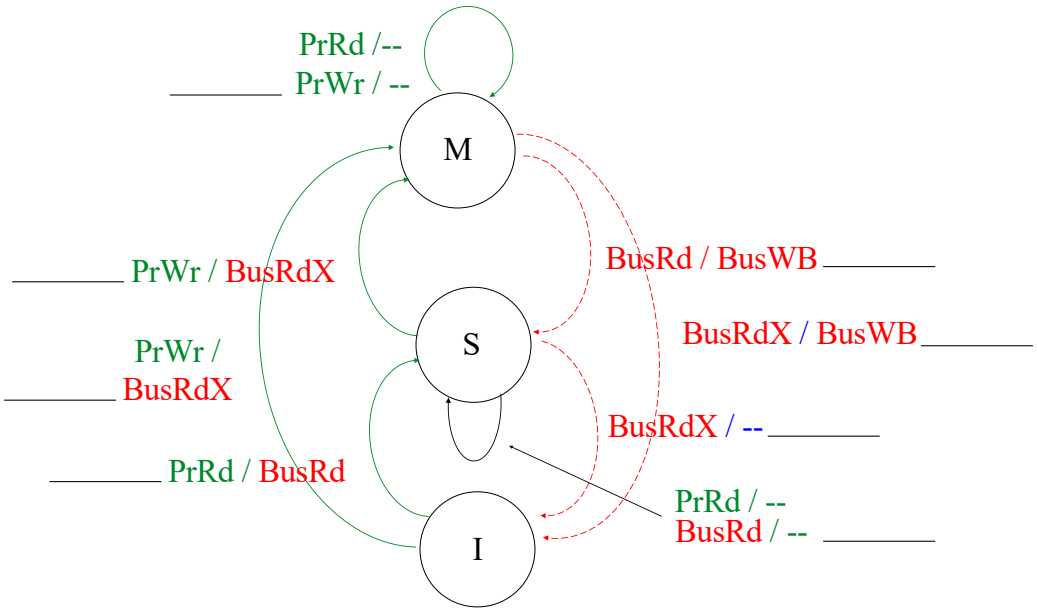
```
void worker(int* counter) {  // each thread runs this function

  barrier();

  // <--- invalidate caches: assume all lines in all caches are
  //      invalidated here after leaving barrier --->

  for (int i=0; i<NUM_ITERS; i++)
    (*counter)++;   // work here (load + incr + store)
}

void test(int num_threads) {
  std::thread threads[MAX_THREADS];
  int         counter[MAX_THREADS];  // Assume this is aligned on a cache line boundary

  for (int i=0; i<num_threads; i++)
    threads[i] = std::thread(worker, &counter[i]); // spawn thread
  for (int i=0; i<num_threads; i++)
    threads[i].join();  // wait for thread to terminate
}
```

Consider calling `test()` with `num_threads`=2.

Assuming that code is run on a dual-core processor where:

- Each core has its own private cache
- Caches use the MSI protocol to implement coherence. For reference the MSI diagram is given on the next page.

**PROBLEM CONTINUES ON NEXT PAGE...**




In your answer to the following questions:

- You should only analyze references to the `counter` array
- Assume all arithmetic is free, you only need to think about memory operations in this problem.
- **Assume that the bus access protocol is such that one core executes the whole C statement involving both the memory read and then the memory write (see the comment "// work here") before allowing bus transactions from the other core.**
- **Assume that between iterations of the `for` loop by one core, the other core is able to execute one iteration of its for loop as well.**
- Each processor action (PrRd, PrWr shown in green) takes 1 cycle. (e.g., A cache hit on a PrWr takes 1 cycle)
- Each bus transaction (BusRd, BusRdX, BusWB shown in red) takes 10 cycles. (Specifically: a PrWr that generates BuxRdX takes a total of 1+10=11 cycles. A PrWr that generates a BusRdX that triggers a remote cache BusWB (write back) requires 1+10+10=21 cycles.)
- No other operations manipulate the state of caches
- Only consider operations that occur after the line <invalidate caches>

**PROBLEM CONTINUES ON NEXT PAGE...**

A. (13 pts) How many cycles does the memory system take to execute the worker for-loop on one of the threads with the following cache organization. On the figure, please list how many times the various coherence protocol transitions occur.
   - 16 byte cache size
   - Fully associative cache (any line can go anywhere, no conflict misses)
   - 4 byte cache line size

B. (12 pts) How many cycles does the memory system take to execute the worker for-loop on one of the threads with the following cache organization. On the figure, please list how many times the various coherence protocol transitions occur.
   - 16 byte cache size
   - Fully associative cache (any line can go anywhere, no conflict misses)
   - 8 byte cache line size

## PRACTICE PROBLEM 5:

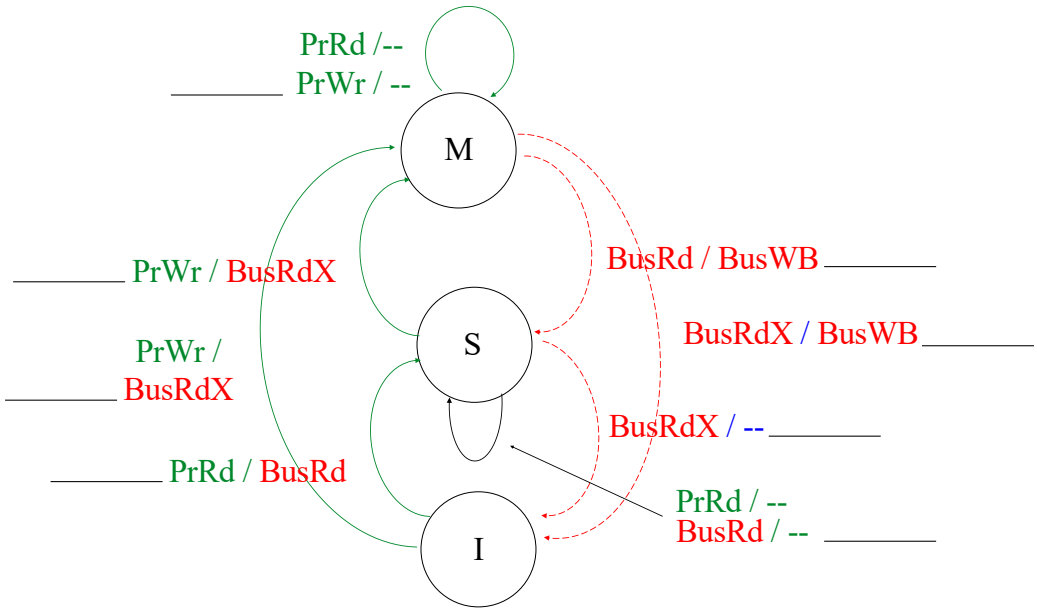
You are already very familiar with the IPSC code for `ispc_sinx()` shown below (it's the example from lecture).

```
export void ispc_sinx(
   uniform int N,
   uniform int terms,
   uniform float* x,
   uniform float* result)  // Assume result is aligned on a cache line boundary
{
   foreach (i = 0 ... N)
   {
     float value = x[i];

     // compute first 'terms' terms of sin(value) here...

     result[i] = value;
   }
}
```

Assume that code is run on a quad-core processor where:

- Each core has its own private cache
- Unlimited cache size
- 32-byte cache line size
- Caches use the MSI protocol to implement coherence. For reference the MSI diagram is given on the next page.

**PROBLEM CONTINUES ON NEXT PAGE...**




A. Given what you know about how the ISPC compiler maps a gang of program instances to hardware resources, and assuming the program only runs a single gang of program instances, please describe why cache coherence is not particularly relevant to the current problem setup. In other words, why is there no need to worry about different cores keeping parts of the array `result` coherent?

B. Now consider that you reimplement the ISPC compiler so that the logic for each program instance is carried out by a **different thread on a different core of the quad-core processor. (The ISPC gang size is 4.)**

Please assume the following:

- Each processor action (PrRd, PrWr shown in green) takes 1 cycle on the core (e.g., A cache hit on a PrWr takes 1 cycle)
- Each bus transaction (BusRd, BusRdX, BusWB shown in red) takes 10 cycles. (Specifically: a PrWr that generates BuxRdX takes a total of 1+10=11 cycles. A PrWr that generates a BusRdX that triggers a remote cache BusWB (write back) requires 1+10+10=21 cycles.)
- Bus transactions and flushes are completely serialized across the cores.
- No other operations manipulate the state of caches
- Only consider operations that write to `result` array.
- All cache lines start in the invalid state.

Consider a program instance carrying out the following loop that **interleaves data elements to program instances**. (Notice we've changed the `foreach` to a regular `for` loop below):

```
// Assume N = 32, programCount = 4
for (uniform int loop_i=0; loop_i<N; loop_i+=programCount)
{
    int i = loop_i + programIndex;
    // do work for iteration i here...

    result[i] = ...
}
```

Please assume that even though the instances are on different threads running in parallel, they progress at the same speed through the loop and their writes are interleaved (instance 0 writes to `result[0]` then instance 1 to `result[1]`, then instance 2 to `result[2]`, then instance 3 to `result[3]`, then instance 0 to `result[4]`, etc.

Given these assumptions, how many cycles does the processor take to access the `result` array given the code above? **Keep in mind each element of `result` is 4 bytes, the cache line size is 32 bytes, and the cache has infinite capacity.** Please list how many times the various coherence protocol transitions occur (e.g. "There are X I->M transitions, and Y S->M transitions, etc."). Your answer should involve work by all cores, over all iterations, to carry out the work of the entire gang.

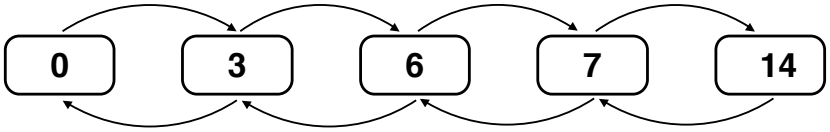
C. Now imagine the program is changed to use **blocked assignment of array elements to program instances**. Under the same assumptions as in part B above (that ISPC is changed to be implemented with one thread per instance), how many cycles does the processor take to access the `result` array given the code below? Please list how many times the various coherence protocol transitions occur. Your answer should involve work by all cores, over all iterations, to carry out the work of the entire gang.

```
// Assume N = 32, programCount = 4;
uniform int count = N / programCount;
int start = programIndex * count;
for (uniform int loop_i=0; loop_i<count; loop_i++)
{
  int i = start + loop_i;
  // do work for iteration i here...
  result[i] = ...
}
```

## PRACTICE PROBLEM 6:

Consider a **SORTED doubly-linked list** that supports the following operations.

- `insert_head`, which traverses the list from the head. The implementation uses hand-over-hand locking just like in class.
- `delete_head`, which deletes a node by traversing from the head, using hand-over-hand locking just like in class.
- `insert_tail`, which traverses the list **backwards from the tail** to insert a node using hand-over-hand locking in the opposite order as `insert_head`.




A. Your friend writes three unit tests that each execute a pair of operations concurrently on the list shown above.

   - Test 1: `insert_head(2)`, `delete_head(14)`
   - Test 2: `insert_head(12)`, `delete_head(6)`
   - Test 3: `insert_head(13)`, `insert_tail(4)`

   The first two unit tests complete without error, but the third test goes badly and it does not terminate with the right answer. Describe what behavior is observed and why the problem occurs. (All unit tests start with the list in the state shown above.)

B. Imagine that locks in this system supported not only `lock()` and `unlock()`, but the ability to query the state of the lock via the call `trylock()` (this call takes the lock if the lock is free, but immediately returns false if the lock is currently locked – it does not block). Given this functionality, describe a fix to the problem you identified in part A? **Your answer should avoid livelock, but it is acceptable to allow for the possibility of starvation.**

**Thinking About AtomicMin**

## PRACTICE PROBLEM 7:

Consider two implementations of `atomicMin` implemented below. For reference, we've also given you the definition of `atomicCAS()`.

```
// As a reminder CAS() performs this logic atomically
int CAS(int* addr, int compare, int val) {
   int old = *addr;
   *addr = (old == compare) ? val : old;
   return old;
}

int atomicMin1(int* addr, int val) {
  int old = *addr;
  int new = min(old, val);
  while (atomicCAS(addr, old, new) != old) {
    old = *addr;
    new = min(old, val);
  }
}

int atomicMin2(int *addr, int val) {
  while (1) {
    int old = *addr;
    int new = min(old, val);
    if (new == old)
       return old;
    if (atomicCAS(addr, old, new) == old)
       return new;
    }
  }
}
```

**The question is on the next page...**

Consider the following code that uses an `atomicMin` operation. For now, we won't say whether it's implemented as `atomicMin1` or `atomicMin2()`. You'll tell us in a second!

```
int globalMin = VERY_LARGE; // initial to very big number
int values[VERY_LARGE];

// initialize values here...

// assume this code is parallelized across many threads with
// iterations of the loop INTERLEAVED across the threads.
for (int i=0; i<VERY_LARGE; i++) {
  atomicMin(&globalMin, values[i]);
}
```

Consider running the code on two possible initializations of the array `values`. In the first initialization, assume `values[i]=VERY_LARGE-i`. In the second, assume `values[i]=i`. Assume that the code is running on a processor with a large number of cores, each with their own private caches that run the MSI coherence protocol. Consider the relative performance of using `atomicMin2` compared to `atomicMin1`.

Do you expect the relative performance of `atomicMin2` compared to `atomicMin1` to be greater for the first data value initialization or the second. Why or why not?

**The Eras Tour Comes to Stanford**

## PRACTICE PROBLEM 8:

Taylor Swift decides to add one final stop to the Eras tour, an exclusive show for CS149 students in NVIDIA auditorium. Inspired by Assignment 3, the crew decides to write a parallel program that renders randomly oriented semi-transparent lines onto a big screen placed behind Taylor. In the code below, **assume that the loop body in the function `renderImage` is parallelized using four threads that are run on a quad-core processor.**

```
// global variables
float pixels[HEIGHT][WIDTH];       // pixels initialized to 0.0
Lock  pixelLocks[HEIGHT][WIDTH];   // per-pixel locks

// draws semi-transparent line connecting the pixels
// xstart, ystart are the starting pixel for the line
// xend, yend are the ending pixel for the line
//
// *** IMPORTANT *** there is no guarantee that xstart <= xend or ystart <= yend
void renderLine(int xstart, int ystart, int xend, int yend) {
  int xcur = xstart;
  int ycur = ystart;

  while (xcur != xend && ycur != yend) {

    LOCK(&pixelLocks[xcur][ycur]);    // lock the lock
    pixels[xcur][ycur] += 0.1f;       // read-modify write
    UNLOCK(&pixelLocks[xcur][ycur]);  // unlock the lock

    // get next pixel to paint in the line
    int xnext, ynext;
    nextPixel(xcur, ycur, &xnext, &next);

    xcur = xnext;
    ycur = ynext;
  }
}

void renderImage() {

  // **** assume the body of this loop is parallelized across threads
  // (local variables defined in the loop body are per-thread variables) ****
  for (int i=0; i<NUM_LINES; i++) {
    int xstart, ystart, xend, yend;

    // assume this method generates random line endpoints (xstart, ystart)
    // and (xend, yend) within (0,0) and (WIDTH,HEIGHT)
    createRandomLine(&xstart, &ystart, &xend, &yend);
    renderLine(xstart, ystart, xend, yend);
  }
}
```

**The questions begin on the next page...**

A. Just before the show starts, a crew member comes up to Taylor and says *"We may have a problem, I think the rendering algorithm for the stage background will deadlock."*. Is the crew member correct? Why or why not? If the crew member is correct that deadlock is possible, provide a description of how you would modify the code to eliminate the deadlock, but preserve the fine-grained locking behavior of the code.

Circle one: Yes No

Explain:

B. Please assume you have a correct solution to part A. Now also assume that the image being produced is very large (many millions of pixels), and that the quad-core processor running the four threads implements the MSI cache coherence protocol to keep per-core caches coherent.

Recall from part A that the line endpoints for all `NUM_LINES` lines are randomly generated, and that there are four threads running the loop iterations in `renderImage` in parallel. Under these conditions, give one reason why using `LOCK_A` (see below) is likely to be **more efficient** than `LOCK_B` when running this program on this workload. Please justify your answer by citing properties of the current workload.

```
// standard atomic compare-and-swap operation from class
// assume this entire function is carried out atomically as a single operation
int CAS(int* mem, int compare, int val) {
  int old = *mem;
  *mem = (old == compare) ? val : old;
  return old;
}

void LOCK_A(int* lk) {
  while (CAS(lk, 0, 1) == 1);
}

void LOCK_B(int* lk) {
  while (true) {
    while (*lk != 0);
    if (CAS(lk, 0, 1) == 1)
      return;
  }
}
```

C. Assume the question in part A is resolved correctly and the show goes on. The CS149 students are having a blast! During intermission, Taylor's boyfriend Travis Kelce sneaks backstage. In the hopes of trying to contribute to the show, he changes the code for the program, modifying the `renderLine()` function to change the displayed line pattern as follows:

```
// *** IMPORTANT *** there is no guarantee that xstart <= xend or ystart <= yend
void renderLine(int xstart, int ystart, int xend, int yend) {
  int xcur = xstart;
  int ycur = ystart;

  while (xcur != xend && ycur != yend) {
    // get next pixel to paint in the line
    int xnext, ynext;
    nextPixel(xcur, ycur, &xnext, &ynext);

    LOCK(&pixelLocks[xcur][ycur]);
    LOCK(&pixelLocks[xnext][ynext]);

    // fabs() is floating-point absolute value
    if (fabs(pixels[xcur][ycur] - pixels[xnext][ynext]) < 0.5)
      pixels[xcur][ycur] -= 0.1f;  // read-modify write
    else
      pixels[xcur][ycur] += 0.1f;  // read-modify write

    UNLOCK(pixelLocks[xcur][ycur]);
    UNLOCK(pixelLocks[xnext][ynext]);

    xcur = xnext;
    ycur = ynext;
  }
}
```

Just before the show resumes, a crew member comes up to Taylor and says *"I'm sorry to bother you again Taylor, but Travis Kelce changed the rendering algorithm, and now it may deadlock."* Is the crew member correct? Why or why not? If the crew member is correct that deadlock is possible, provide a precise description of how you would modify the code to eliminate the deadlock, but preserve the fine-grained locking behavior of the code.

Circle one: Yes No

Explain:

D. Towards the end of the show, Taylor pauses and tells the CS149 audience, tonight, we're doing something special, *"I have a Question...? Imagine that you don't have access to a lock API, but instead you can add a single* `atomic` *block to the code from part C. Please add* `atomic {...}` *to the code in* `renderLine` *so that it performs the same logic as part C, enables maximum concurrency, and is thread safe and deadlock free."*

```
void renderLine(int xstart, int ystart, int xend, int yend) {
  int xcur = xstart;
  int ycur = ystart;

  while (xcur != xend && ycur != yend) {


    // get next pixel to paint in the line
    int xnext, ynext;
    nextPixel(xcur, ycur, &xnext, &ynext);


    if (fabs(pixels[xcur][ycur] - pixels[xnext][ynext]) < .5) {


      pixels[xcur][ycur] -= 0.1f;  // read-modify write


    } else {


       pixels[xcur][ycur] += 0.1f;  // read-modify write


    }



    xcur = xnext;


    ycur = ynext;

  }
}
```

## PRACTICE PROBLEM 9:

English football superstar Harry Kane starts preparing for the next World Cup by developing an algorithm for simulating penalty kicks. The algorithm simulates many penalty kicks in parallel by running a function `simulate_random_kick()` in each thread. When complete, the program prints out the height of the lowest kick found so that Harry has a target in mind for future practice sessions.

The code below uses the function `atomicCAS`, which was discussed in class. **The definition of `atomicCAS` is given for convenience, but please keep in mind this is an atomic operation that is treated as a write by the cache coherence protocol.**

```
// REMEMBER THIS IS EXECUTED ATOMICALLY
int atomicCAS(int* addr, int compare, int val) {
   int old = *addr;
   *addr = (old == compare) ? val : old;
   return old;
}
int lowest_so_far;   // shared global variable among threads,
                     // holds lowest height so far in centimeters

void thread_main() {   // assume this code is run by many threads

   float ball_height;

   for (int i=0; i<1000000; i++) {
     simulate_random_kick(&ball_height);      // runs a sim that fills in ball height
     int old_lowest = lowest_so_far;
     int min_height = min(ball_height, old_lowest);
     while (atomicCAS(&lowest_so_far, old_lowest, min_height) != old_lowest) {
       old_lowest = lowest_so_far;
       min_height = min(ball_height, old_lowest);
     }
   }

   barrier();     // a regular barrier across all threads

   if (get_thread_id() == 0) {  // assume get_thread_id() behaves as expected
      printf("The lowest height is %d cm\n", lowest_so_far);
   }
}
```

A. In your own words, describe the correctness reason for why there is a barrier in this program. (How might the program give incorrect results if there is no barrier?)

B. Imagine the case where **`simulate_random_kick()` is a very expensive function that takes over a second to compute. All calls to `simulate_random_kick()` take the same amount of time**. Please describe whether you think this program will achieve near perfect speedup on a multi-core processor that implements invalidation-based cache coherence. Please describe why or why not? If you think there is a way to improve its speedup substantially, describe how you might improve the code (rough pseudocode is fine).

C. Now imagine the case where **`simulate_random_kick()` is a very lightweight function that takes just a few instructions. All calls to the function take the same amount of time.** Please describe whether you think this program will achieve near perfect speedup on a high core count multi-core processor that implements invalidation-based cache coherence. Please describe why or why not? If you think there is a way to improve its speedup substantially, describe how you might improve the code (rough pseudocode is fine).

D. Now consider the case where `simulate_random_kick` outputs both the lowest ball height AND a video of the simulated kick for Harry to watch. **Harry implements the following program which is intended to print the lowest kick height as before, and should also saves the corresponding video to disk.** Here's the updated code.

```
int lowest_so_far;   // shared global variable among threads,
                     // holds lowest height so far in centimeters

Video lowest_vid;

void thread_main() {   // assume this code is run by many threads

   float ball_height;
   Video vid;

   for (int i=0; i<1000000/num_threads(); i++) {
     // runs a sim that fills in ball height and corresponding video
     simulate_random_kick(&ball_height, &vid);

     int old_lowest = lowest_so_far;
     int min_height = min(ball_height, old_lowest);

     while (atomicCAS(&lowest_so_far, old_lowest, min_height) != old_lowest) {
       old_lowest = lowest_so_far;
       min_height = min(ball_height, old_lowest);
     }

     if (ball_height == min_height) {
        lowest_vid = vid;
     }
   }

   barrier();     // a regular barrier across all threads

   if (get_thread_id() == 0) {  // assume get_thread_id() behaves as expected
      printf("The lowest height is %d cm\n", lowest_so_far);
      saveFile(lowest_vid);
   }
}
```

**Please describe the correctness problem in the code.**

E. (5 pts) Imagine that you are trying to solve the same problem as 3D, but instead of `atomicCAS` you have access to a regular lock via `lock()` and `unlock()`. Please pseudocode a solution that correctly obtains the right answer, while minimizing the amount of time spent in its critical section. Your solution need not rewrite the whole algorithm, just give us the body of the `for` loop.

**Tricky Little Graphs**

## PRACTICE PROBLEM 10:

```
struct Graph_node {
   Lock   lock;
   float  value;
   int    num_edges;    // number of edges connecting to node (its degree)
   int*   neighbor_ids; // array of indices of adjacent nodes
};

// a graph is a list of nodes, just like in assignment 3
Graph_node graph[MAX_NODES];
```

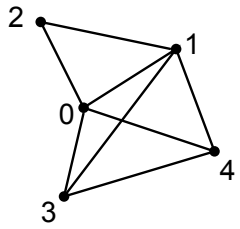
Consider the undirected graph representation shown in the code above.

Your boss asks you to write a program that atomically updates each graph node's `value` field by setting it to the average of all the values of neighboring nodes. The program must obtain a lock on the current node and all adjacent nodes to perform the update. It does so as follows...

```
void update(int id) {
    Graph_node* n = &graph[id];
    LOCK(n->lock);
    for (int i=0; i<n->num_edges; i++)
      LOCK(graph[n->neighbor_ids[i]].lock);
    // now perform computation...
```

Consider running the update code in parallel on nodes 0 and 1 in the two graphs below. For each graph, determine if deadlock occurs. Please describe why or why not. (Note: we do not ask you to solve the deadlock problem, but think about you might avoid it, assuming you must still only use locks. Consider changing the order in which you take the locks.

Graph A



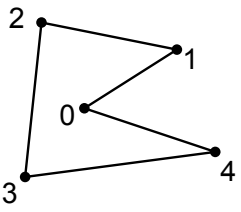

Values in neighbor_ids[]:

0: 3 4 2 1  3: 0 4 1

1: 0 2 3 4  4: 0 3

2: 0 1

Graph B




Values in neighbor_ids[]:

0: 1 4  3: 2 4

1: 0 2  4: 0 3

2: 1 3

**Hash Table Parallelization**

## PRACTICE PROBLEM 11:

A. Consider the following sequence of locking/unlocking operations by two threads.

```
        T0                              T1
==================              ==================

lock(l1);                         lock(l3);
lock(l2);                         lock(l2);
lock(l3);                         lock(l1);

// critical section               // critical section

unlock(l3);                       unlock(l1);
unlock(l2);                       unlock(l2);
unlock(l1);                       unlock(l3);
```

Assuming that both threads must acquire all three locks prior to entering the critical section, please describe the **correctness problem** that can occur when running these two threads. Please also describe a modification to the code that fixes the problem, while preserving mutual exclusion (protects the critical section).

B. Below you will find an implementation of a hash table (a linked list per bin). The hash table has a function called `tableInsert` that takes two strings, and inserts both strings into the table **only if neither string already exists in the table**. Please implement `tableInsert` below in a manner that enables maximum concurrency. You may add locks wherever you wish. (Update the structs as needed.) To keep things simple, your implementation SHOULD NOT attempt to achieve concurrency within an individual list (notice we didn't give you implementations for `findInList` and `insertInList`). **Careful, things are a little more complex than they seem. You should assume nothing about `hashFunction` other than it distributes strings uniformly across the 0 to `NUM_BINS` domain. (HINT: deadlock!)**

```
struct Node {
  string value;
  Node* next;

};

struct HashTable {
   Node* bins[NUM_BINS];  // each bin is a singly-linked list


};

int   hashFunction(string str);           // maps strings uniformly to [0-NUM_BINS]
bool  findInList(Node* n, string str);    // return true is str is in the list
void  insertInList(Node* n, string str);  // insert str into the list

bool tableInsert(HashTable* table, string s1, string s2) {
   int idx1 = hashFunction(s1);
   int idx2 = hashFunction(s2);
   bool result = false;




   if (!findInList(table->bins[idx1], s1) &&
       !findInList(table->bins[idx2], s2)) {



     insertToList(table->bins[idx1], s1);



     insertToList(table->bins[idx2], s2);



     result = true;
   }




   return result;
}
```

**A Concurrent Binary Search Tree**

## PRACTICE PROBLEM 12:

In this problem you'll work with a version of a binary search tree (BST where **locks are associated with the edges of the tree, rather than the nodes**. Edges are represented as a C++ class `Edge`, declared as follows:

```
class Edge {
 private:
    Node *n;           // Pointer to BST node reachable along edge (or NULL)
    Lock plock;        // Lock associated with arc
 public:
    Node *get();       // Retrieve node pointer
    void set(Node *n); // Set node pointer
    void lock();       // Acquire lock
    void unlock();     // Release lock
};
```

The node data structure has a per-node value, plus edges to its two children

```
class Node {           // Nodes in BST
  public:
    Edge left, right;  // Edges to subtrees
    int value;         // Node value

    Node(int v) {      // constructor
        value = v;
        left.set(NULL);
        right.set(NULL);
    }
};
```

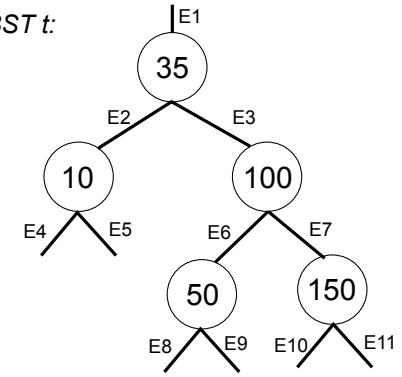
and the tree contains an "edge" to the root: (For an empty tree, the `n` field of the root edge is `NULL`.)

```
class BST {     // BST representation
 private:
    Edge root;
    public:
    // Insert value into BST
    bool insert(int val);

    // Remove maximum value node from BST, and assign its value to *val.
    // Return false if empty.
    bool remove_max(int *val);
};
```

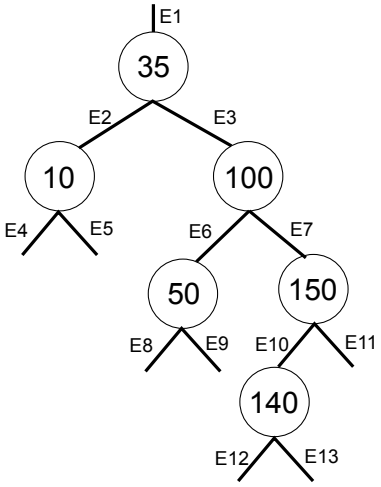
The following BST, which we will call `t` includes labels for all of its arcs. Notice that in a binary search tree, the left subtree of a node $n$ contains nodes with values LESS than N, and the right subtree of a node $n$ contains nodes with values GREATER than $n$.

As a reference, a correct insertion of the value 140 into $t$ would yield the tree on the bottom-left. A correct removal of the maximum value would result in the tree on the bottom-right. We also show the result of removing the maximum element twice.

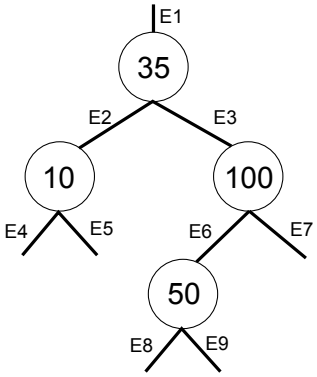
*original BST t:*



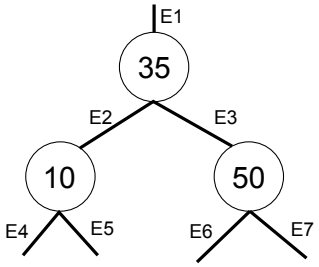

*after inserting 140 to t:*




*after removing max from t:*




*after removing max from t, and then removing max again:*



The following is a function for inserting elements into the tree. **It is intended to be thread safe (but may or may not be).**

```
// Top level insertion code (insert val into BST)
bool BST::insert(int val) {
    bool result = false;

    root.lock();
    result = insert_sync(&root, val);

    return result;
}

// insertion subroutine
bool BST::insert_sync(Edge *e, int val) {
    Node *n = e->get();
    if (n == NULL) {
        e->set(new Node(val));
        e->unlock();
        return true;
    }
    e->unlock();
    if (n->value == val) {
        return false;
    }
    Edge *next = (val < n->value) ? &n->left : &n->right;
    next->lock();
    return insert_sync(next, val);
}
```

The following is a function for removing the maximum element in the tree. (Notice it always traverses to the right child until there are no more right children.) **It is intended to be thread safe (but may or may not be).**

```
// Top level remove code. Returns true if a node exists, and fills in val
bool BST::remove_max(int* val) {
  root.lock();
  return remove_max_sync(root, *val);
}

// removal subroutine
bool BST::remove_max_sync(Edge *e, int *val) {
    Node *n = e->get();
    if (n == NULL) {
        e->unlock();
        return false;
    }
    Edge *next = &n->right;
    next->lock();

    bool found = remove_max_sync(next, val);

    if (!found) {
        // Current node holds the maximum value since there
        // is no right child

        *val = n->value;

        // Replace this node with its left subtree
        Edge *left = &n->left;
        left->lock();
        e->set(left->get());
        left->unlock();
        delete n;
    }
    e->unlock();
    return true;
}
```

A. For BST `t`, assume a thread executes the call `t.insert(40)`. What sequence of lock acquisitions and releases would it cause to occur? (Use the notation `L1` to indicate locking of edge E1, `U2` to indicate unlocking of edge E2, etc.)

B. For the original BST `t` (without any additional insertions), assume a thread executes the call `t.remove_max()`. What sequence of lock acquisitions and releases would occur?

C. Starting with BST `t`, suppose two threads execute the following:

**Thread 1:** `t.insert(140);`

**Thread 2:** `int v; t.remove_max(&v);`

Assume that Thread 1 acquires the lock on edge E1 first. Identify sequences of actions by the two threads that could cause the resulting tree to contain only four nodes, and then answer the following:

(a) Describe the specific locking, unlocking, and update operations:

(b) Draw (or describe in text) the resulting tree.

D. Starting with BST `t`, suppose two threads execute the following:

**Thread 1:** `int v; t.remove_max(&v);`

**Thread 2:** `t.insert(200);`

Assume Thread 1 acquires the lock on edge E1 first. List all possible value(s) that could be assigned to `v`. Explain why this is the complete set of possibilities.

E. Modify the insertion code below to eliminate the problem you identified earlier, **while still allowing fine-grained concurrency.**

```
bool BST::insert_sync(Edge *e, int val) {

  Node *n = e->get();

    if (n == NULL) {


        e->set(new Node(val));


        e->unlock();


        return true;
    }


    e->unlock();


    if (n->value == val) {


        return false;


    }


    Edge *next = val < n->value ? &n->left : &n->right;


    next->lock();


    return insert_sync(next, val);
}
```