Stanford CS149: Parallel Computing
Written Assignment 4

Feeling Relaxed

Problem 1:

A. Consider the following code executed by three threads on a **cache-coherent, relaxed consistency** memory system. Specifically, the system allows reordering of writes (W->W reordering) and in these cases makes no guarantees about when notification of writes is delivered to other processors. **You should assume that all variables are initialized to 0 prior to the code you see below.**

```
P1:                   P2:                   P3:
===================   ===================   ===================
x = 10;               while (!flag);        while (!flag);
flag = true;          print x;              print x;
print x;
```

You run the code and P2 prints "10". List what values might be printed by P1 and P3. **Please also explain why your answer shows that the system does not provide sequentially consistent execution.**

B. Imagine you are given a memory write fence instruction (`wfence`), which ensures that all writes prior to the fence are visible to all processors when the fence operation completes. Please add the **minimal number of write fences** to the code in Part A to ensure that the output is guaranteed to be that same as the output of a machine with sequentially consistent memory.

Understanding Instruction Interleavings (Some of Which are Relaxed)

Problem 2:

Assume that `x` and `y` are memory locations and `r1` and `r2` are per-thread local registers, `M` is a lock (a mutex), and `T0` and `T1` are threads. For each of the following program fragments we want you to compute the number of possible final states of the system. (due to different interleavings) **For each unique final state give the values stored in memory (X,Y) and the registers (T0.r1, T0.r2, T1.r1, T1.r2).**

Assume all fragments start with the initial conditions:
`T0.r1=0, T0.r2=0, T1.r1=0, T1.r2=0, x=0, y=0`

You may assume sequential consistency at all times except for the final part, where we explicitly mention a relaxed consistency model.

Hint: We recommend that you number the instructions, then work out all possible interleavings of the instructions, and then determine the outcomes of those interleavings.

A.

Thread T0	Thread T1

`lock(M)`
`T0.r1 = x`
`unlock(M)`
	`lock(M)`
	`x = 1`
	`y = 1`
	`unlock(M)`

B.

Thread T0	Thread T1

`T0.r1 = x`	`lock(M)`
	`x = 1`
	`y = 1`
	`unlock(M)`

C.

Thread T0	Thread T1
`T0.r1 = x`	`y = 1`
`T0.r2 = y`	`x = 1`
	`y = 2`

D. Assume total store ordering (TSO) relaxed consistency. **TSO relaxes read after write order.** Specifically: A processor running a thread can proceed with a read from address Y THAT IS AFTER a write to address X in program order before the write to X is complete and visible to all processors.

Thread T0	Thread T1
`y = 5`	`T1.r2 = y`
`T0.r1 = x`	`T1.r1 = x`
`T0.r2 = T0.r1 + 1`	`T1.r2 = T1.r1 + T1.r2`
`x = T0.r2`	`x = T1.r2`

Coherence, Consistency, and Locks

Problem 3:

A. 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. 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.)

Problem 4:

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. 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.

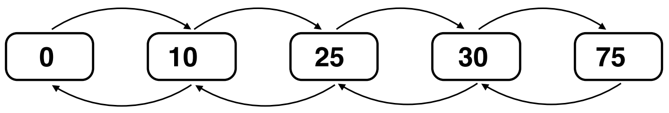
Two threads + a doubly Linked List = trouble

PRACTICE PROBLEM 1:

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. 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. 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. 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. 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.

Load Linked / Store Conditional and Cache Coherence

PRACTICE PROBLEM 2:

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**.

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;
  }
}
```

Consider the function `TryExchange()`, which is implemented using LL and SC as given below. `tryExchange` attempts to atomically read value of `x` and replace it with that value of `y`. It stores the old value at the address pointed to by `x` in the variable `z`, and **returns true if the atomic exchange succeeded.**

```
int TryExchange(int *x, int y, int *z) {
  *z = load_linked(x);
  return store_conditional(x, y);  // return true if swap actually occurred
}
```

A. Please implement a spin lock using `TryExchange`. (Your implementation can assume that calling threads behave reasonably and will not attempt to unlock a lock they they have not previously acquired, or lock a lock they already hold.)

```
void Lock(int* l) {




}

void Unlock(int* l) {


}
```

B. Here is another way to implement a lock by directly using LL and SC. The lock is taken if the LL returns 0 and the SC succeeds, else the code tries again.

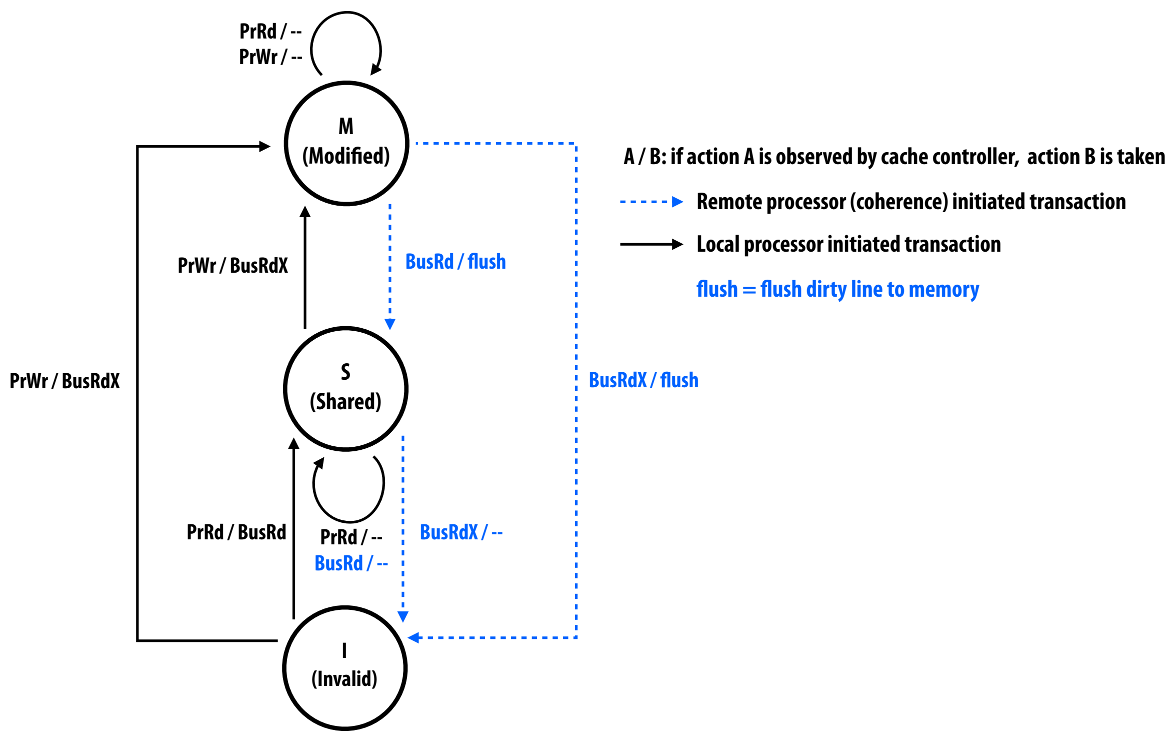
```
void Lock(int* mylock) {
  while (!(load_linked(mylock) == 0 && store_conditional(mylock, 1)));
}
```

We'd like you to analyze the cache coherence behavior of the two implementations of `Lock`: the one directly above, as well as your implementation based on `TryExchange`. Assume you have a multi-core processor that implements cache coherence using the MSI protocol. **(If needed, we provide a MSI state diagram on the next page.)** In addition to MSI, the system implements LL and SC as follows:

- LL is implemented by moving from the I state to the S state via BusRd (and setting an extra LL bit in the S state). If the line is already in M, it moves the line to S. (Note there is no need to issue any bus commands in this case, think about why!) If the line is already in S, no bus traffic is required.
- SC is implemented by checking that the line is in the S state in the local cache with the LL bit set. If it's not, the SC fails. If it is, then the processor moves the line from S to M and issues BusRdX and performs the update.

Assume that cache hits take 1 cycle (no bus traffic required), and bus transactions take 20 cycles (a cache miss is 1+20=21 total cycles). What's the performance difference between the two lock implementations assuming the while loop spins 100 times before the lock is zero? Please state any assumptions you make in your answer. Back-of-the-envelope calculations are fine, we aren't looking for a specific numerical answer, but you can give one if you want to.

Hint: remember that C code "early outs" if an AND expression cannot be true because the first term is false! (It only evaluates `A` when evaluating the expression `(A && B)` if `A` is false.)


PrRd / --
PrWr / --
M
(Modified)
A / B: if action A is observed by cache controller, action B is taken
Remote processor (coherence) initiated transaction
Local processor initiated transaction
flush = flush dirty line to memory
PrWr / BusRdX
BusRd / flush
PrWr / BusRdX
S
(Shared)
BusRdX / flush
PrRd / BusRd
PrRd / --
BusRd / --
BusRdX / --
I
(Invalid)

Implementing Reader-Writer Locks

PRACTICE PROBLEM 3:

After the last problem you are hopefully quite familiar with LL-SC. In this problem, you will provide a simple implementation for read-write locks (which should remind you of the way invalidation-based cache coherence works) using the LL-SC primitives that were defined in the previous problem. (PLEASE SEE THE PREVIOUS PROBLEM FOR A DEFINITION OF LL-SC, BUT DO NOT NEED TO SOLVE THE PREVIOUS PROBLEM TO DO THIS PROBLEM.)

A read-write lock has the property that multiple threads may be holding the read lock, but **only one thread may be holding the write lock. If a thread is holding the write lock, no other thread may be holding a read lock.**

You may assume the follow simplifications:

- Sequentially consistent memory
- Threads will never call lock on a lock they hold, or unlock a lock they do not hold
- Your solutions may spin. We don't care about lock performance or fairness
- You may modify the `read_write_lock` struct if you wish, but you don't need to.

Hints:

- How do you implement atomic increment and decrement using LL/SC?
- You'll need some way to check to see if no other thread has the lock in either the read/write locked state.

The code to fill out is on the next page...

```
struct read_write_lock {
  // assume these two values are initialized to 0
  int num_readers;      // count of readers holding the lock
  int is_write_locked;  // 1 if there is a writer holding the lock
};

void write_lock(read_write_lock *l) {











}

void write_unlock(read_write_lock *l) {








}


void read_lock(read_write_lock *l) {














}

void read_unlock(read_write_lock *l) {








}
```

One more question about Spark

PRACTICE PROBLEM 4:

Consider the following program written using Spark RDDs, in a C-like syntax. Assume that `readRDDFromFile()` generates an RDD with elements of type `int` by reading numbers from a file, and that the functions `addOne()` and `addTwo()` are defined as given below. **You may also assume that `map()`, `readRDDFromFile()`, and `writeRDDToFile()` are THE ONLY transformations allowed on RDDs.**

```
int addOne(int x) { return x+1; }
int addTwo(int x) { return x+2; }

RDD r1 = readRDDFromFile();
RDD r2 = r1.map(addOne);
RDD r3 = r2.map(addTwo);
writeRDDToFile(r3);
```

Assume that there are N numbers in the file, and consider two potential implementations of this program. In the code below, `readIntFromFile()` and `writeIntToFile()` read/write exactly one integer to/from the file.

```
// IMPLEMENTATION 1

int array1[N];
int array2[N];
int array3[N];

for (int i=0; i<N; i++)
   array1[i] = readIntFromFile();
for (int i=0; i<N; i++)
   array2[i] = addOne(array1[i]);
for (int i=0; i<N; i++)
   array3[i] = addTwo(array2[i]);
for (int i=0; i<N; i++)
   writeIntToFile(array3[i]);

// IMPLEMENTATION 2

for (int i=0; i<N; i++) {
  writeIntToFile(addTwo(addOne(readIntFromFile())));
}
```

The second implementation computes elements of the three RDDs in a different order than the first implementation. It also clearly uses far less memory than the first. Are both implementations correct implementations of the Spark RDD abstraction? (In other words do they both compute the expected result?) If your answer is yes, please describe WHAT properties of RDDs and RDD transformations allow for both of these two different implementations. If your answer is no, please describe why. **(Please ignore robustness to node failure in this problem.)**