Stanford CS149: Parallel Computing
Written Assignment 6

Understanding Instruction Interleavings (Some of Which are Relaxed)

Problem 1: (Graded for Correctness - 25 pts)

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. (6 pts)

Thread T0	Thread T1

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

B. (6 pts)

Thread T0	Thread T1

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

C. (6 pts)

Thread T0	Thread T1

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

D. (7 pts) 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.

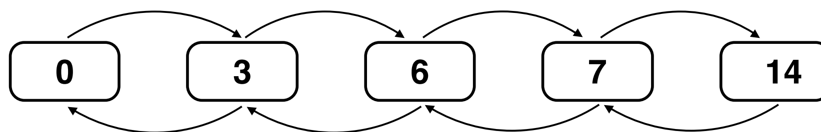
Thread T0	Thread T1
`y = 5`	`T1.r2 = y`
`T0.r1 = x`	`T1.r1 = x`
`T0.r2 = T0.r1 + 1`	`T1.r2 = T1.r1 + T1.r2`
`x = T0.r2`	`x = T1.r2`

Transactions on a Doubly Linked List

Problem 2: (Graded for Correctness - 25 pts)

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

- `insert_front`, which traverses the list from the front.
- `delete_front`, which deletes a node by traversing from the front
- `insert_back`, which traverses the list **backwards from the end** to insert a node in the opposite order as `insert_front`.



In this problem, assume that the entire body of each function `insert_front`, `delete_front`, and `insert_back` is placed in its own atomic block, and the code is run on a system **supporting optimistic (for both reads and writes) transactional memory**.

A. (7 pts) Your friend writes three unit tests that each execute a pair of operations concurrently on the list shown above.

   - Test 1: `insert_front(2),delete_front(14)`
   - Test 2: `insert_front(12),delete_front(6)`
   - Test 3: `insert_front(13),insert_back(4)`

   Assuming all unit tests start with the list in the state shown above, is the code correct? (By correct, we mean there are no race conditions and so all operations will modify the data structure according to their specification.) Why or why not?

B. (6 pts) Consider two transactions performing `insert_front(4)` and `delete_front(14)`. Assume both transactions start at the same time on different cores and the transaction for `insert_front(4)` proceeds to commit while the `delete_front(14)` transaction has just iterated to the node with value 7. Must either of the two transactions abort in this situation? Why? **(Remember this is an optimistic transactional memory system!)**

C. (6 pts) Must either transaction abort if the transaction for `delete_front(14)` proceeds to commit before the transaction for `insert_front(4)` does? Why? **Please assume that at the time of the attempted commit, `insert_front(4)` has iterated to node 3, but has not begun to modify the list.**

D. (6 pts) Must either transaction abort if the situation in part C is changed so that `delete_front(14)` attempts to commit first, but by this time `insert_front(4)` has made updates to the list (although not yet initiated its commit)? Why?

Load Linked / Store Conditional and Cache Coherence

Problem 3: (Graded on Effort Only - 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**.

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.

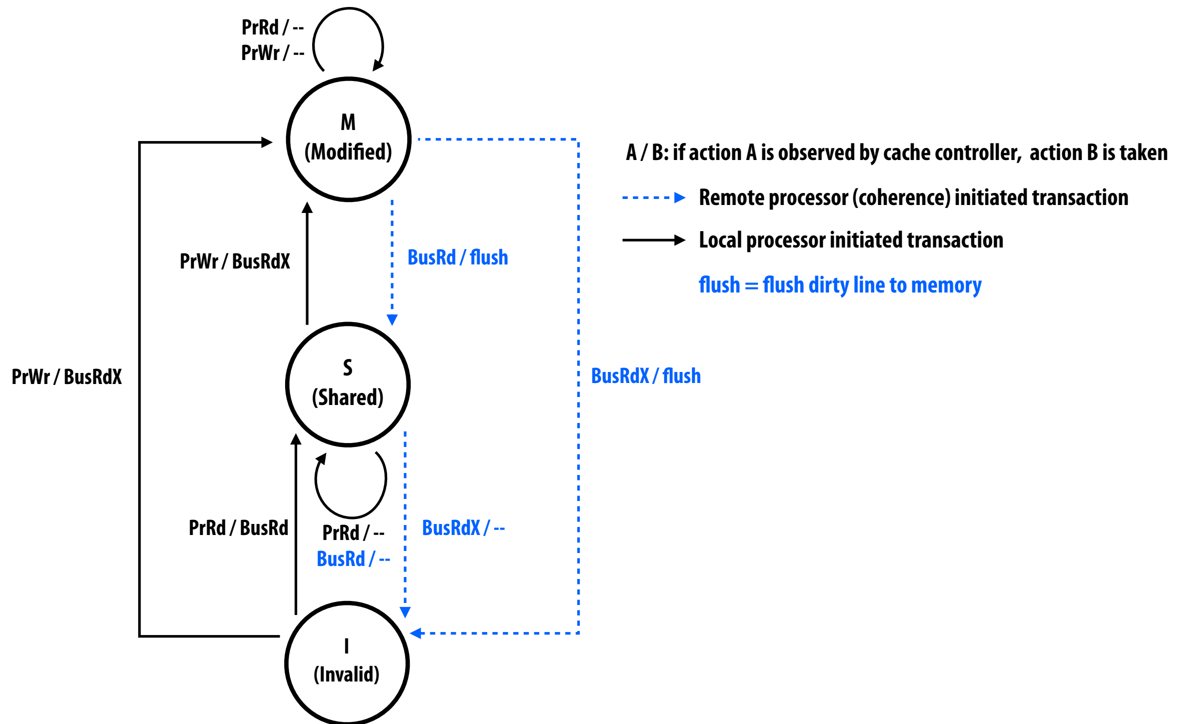
```
void Lock(int* mylock) {
  while (!(load_linked(mylock) == 0 && store_conditional(mylock, 1)));
}
```

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)

Lock-Free and Relaxed Consistency

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

Relaxed memory consistency makes it even trickier to write lock-free data structures. Consider the following code for lock free linked list insertion running on a machine with relaxed write-write ordering. Where would you insert a write fence to ensure correct behavior? **Also describe what problem might happen without the fence?**

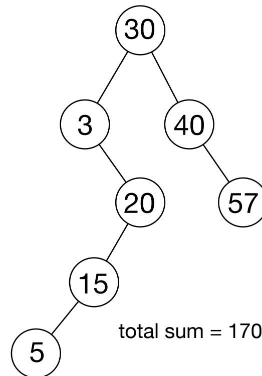
In this problem: a CAS is considered a write operation, disregard the ABA problem, assume that insertion is the only operation on the data structure, and you only need to consider the case of inserting into the middle of the list.

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

struct List {
  Node* head;
};

// insert new node with the value 'value' after the specified node 'after'
void insert_after(List* list, Node* after, int value) {

   Node* n = new Node;
   n->value = value;

   Node* prev = list->head;

   while (prev->next) {


     if (prev == after) {

       while (1) {

         Node* old_next = prev->next;


         n->next = old_next;


         if (compare_and_swap(&prev->next, old_next, n) == old_next) {


           return;


         }

       }

     }

     prev = prev->next;
   }
}
```

Transactions on Trees

PRACTICE PROBLEM 1:

Consider the binary search tree illustrated below.



The operations `insert` (insert value into tree, assuming no duplicates) and `sum` (return the sum of all elements in the tree) are implemented as transactional operations on the tree as shown below.

```
struct Node {
   Node *left, *right;
   int value;
};
Node* root;  // root of tree, assume non-null

void insertNode(Node* n, int value) {
  if (value < n->value) {
    if (n->left == NULL)
       n->left = createNode(value);
    else
       insertNode(n->left, value);
  } else if (value > n->value) {
    if (n->right == NULL)
       n->right = createNode(value);
    else
       insertNode(n->right, value);
  }  // insert won't be called with a duplicate element, so there's no else case
}

int sumNode(Node* n) {
   if (n == null) return 0;
   int total = n->value;
   total += sumNode(n->left);
   total += sumNode(n->right);
   return total;
}

void insert(int value) {  atomic { insertNode(root, value); }   }
int sum()              {  atomic { return sumNode(root); )      }
```

Consider the following four operations are executed against the tree in parallel by different threads.

```
insert(10);
insert(25);
insert(24);
int x = sum();
```

A. Consider different orderings of how these four operations could be evaluated. Please draw all possible trees that may result from execution of these four transactions. (Note: it's fine to draw only subtrees rooted at node 20 since that's the only part of the tree that's effected.)

B. Please list all possible values that may be returned by `sum()`.

C. Do your answers to parts A or B change depending on whether the implementation of transactions is optimistic or pessimistic? Why or why not?

D. Consider an implementation of **lazy, optimistic** transactional memory that manages transactions at the granularity of tree nodes (the read and writes sets are lists of nodes). Assume that the transaction `insert(10)` commits when `insert(24)` and `insert(25)` are currently at node 20, and `sum()` is at node 40. Which of the four transactions (if any) are aborted? **Please describe why.**

E. Assume that the transaction `insert(25)` commits when `insert(10)` is at node 15, `insert(24)` has already modified the tree but not yet committed , and `sum()` is at node 3. Which transactions (if any) are aborted? **Again, please describe why.**

F. Now consider a transactional implementation that is **pessimistic** with respect to writes (check for conflict on write) and **optimistic** with respect to reads. The implementation also employs a "writer wins" conflict management scheme – meaning that the transaction issuing a conflicting write will not be aborted (the other conflicting transaction will). Describe how a **livelock problem** could occur in this code.

G. Give one livelock avoidance technique that an implementation of a pessimistic transactional memory system might use. You only need to summarize a basic approach, but make sure your answer is clear enough to refer to how you'd schedule the *transactions*.

PRACTICE PROBLEM 2:

Consider the following execution, where T1 and T2 occur in parallel. Initially, we have:

```
x = y = a = 0
```

	T1		T2
(1)	x = 1	(3)	y = 2
(2)	a = y + 1	(4)	a = x + y + a

Assume the threads run under a *sequentially consistent* memory model. For each of the questions determine if the given output is possible after all statements have executed and **give an ordering which produces the output. If the output is not possible, write "NOT POSSIBLE".**

A. x = 1, y = 2, a = 4

B. x = 1, y = 2, a = 2

C. Recall that the relaxed consistency model TSO (total store ordering) allows **read-after-write** relaxation, meaning that if T1 performs a write to X and then a read to Y (which is independent of the write to X and therefore can be executed out of order without impacting T1), other processors might be notified that T1 executed the read of Y before they are notified of T1's write to X.

   What is the performance benefit to T1 of relaxing read-after-write ordering?

D. Please give **AT LEAST 4 statement orderings allowed under TSO that are not possible under sequential consistency. +1 extra credit point for answers that list ALL possible orderings.**

E. Suppose you execute the code on a completely relaxed consistency processor such as one with the ARM ISA. What is the minimum number of memory barriers that you must add to the code to ensure that the execution matches sequential consistency? **And where do these barrier(s) go in the code?** (You can assume that the barrier prevents reordering of any memory operation around the barrier (before or after).)

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) {

}
```

Feeling Relaxed

PRACTICE PROBLEM 4:

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.

Coherence and Transactional Memory

PRACTICE PROBLEM 5:

A. Consider a cache coherence protocol that implements an **eager, pessimistic hardware transactional memory**. Each cache line can be one of three states: Invalid (I), Shared (S), or Exclusive (E). The protocol has the following rules:

- A cache miss occurs if the cache line is not present or is in the wrong state.
- Reads change the line in the cache to shared (S) state if it is (I) or not present.
- A line can be in the shared (S) state in multiple caches.
- Writes change the line in the cache to exclusive state
- Only one cache can have the line in exclusive state, in all other caches the line must be invalid (I) or not present.
- **On a cache miss data comes from memory... Unless another processor's cache has the line in exclusive state in which case data comes from that cache.**
- There are processor actions **Tbeg:** begin transaction, **Tend:** end and commit transaction. Remember that when a transaction commits, that might allow a stalled transaction to continue.
- Aborts cause the processor's cache's read and write cache state to be invalidated.
- **If a conflict is detected on a write, the transaction issuing the current action "wins" (abort the other conflicting transaction). If a conflict is detected on a read, the transaction issuing the read should stall waiting for the conflicting transaction to commit. (This is exactly as we discussed in the pessimistic example diagrams in class.**

Given the rules above, show what happens to the cache line state for address X for references made by three processors (P1, P2, P3) by filling in the table below (the first two rows are given). Initially none of the caches contain address X. **If an action causes a processor P to abort or stall a transaction, write "abort" or "stall" in the table entry at the row for that action and the column for P together with the state of P (e.g. "S, stall")**. If a transaction aborts, for simplicity assume that it never resumes for the rest of the duration of the table. (Also, if a transaction has already aborted, assume that the processor executing a **Tend** in a later row of the table does nothing.)

Processor Action	Hit / Miss	P1 state	P2 state	P3 state	Data comes from
P1,P2, P3 Tbeg		--	--	--	
P1 read x	miss	S	--		mem
P3 read x	miss	S	--	S	mem
P2 read x					
P1 Tend, Tbeg					
P2 Tend, Tbeg					
P1 read x					
P3 write x					
P2 read x					
P1 Tend					
P3 Tend					
P2 Tend					

B. Now imagine a cache coherence protocol that implements a **lazy, optimistic** hardware transactional memory system. Each cache line can be one of four states: Invalid (I), Shared (S), Shared Write (SW) or Exclusive (E). The protocol has the following rules:

- A cache miss occurs if the cache line is not present or is in the wrong state.
- Reads change the line in the cache to shared (S) state if it is I or not present.
- A line can be in shared (S) state in multiple caches.
- Writes change the line in the cache to shared write (SW) state; allowed in multiple caches. (Note the SW state is functioning as the write log.)
- SW and S can co-exist in different caches (convince yourself why this is true!)
- Only one cache can have the line in exclusive state, in all other caches the line must be invalid (I) or not present.
- On a cache miss data comes from memory... **unless another processor's cache has the line in exclusive state in which case data comes from that cache.**
- There are processor actions **Tbeg:** begin transaction, **Tend:** end and commit transaction
- Aborts cause read and write cache state to be invalidated
- Transaction commit (Tend) causes cache lines to transition from shared write (SW) to exclusive (E) state and may cause other transactions to abort.

Given the rules above, show what happens to the cache line state for address X for references made by three processors (P1, P2, P3) by filling in the table below. Initially none of the caches contain address X. If an action causes a processor P to abort or stall a transaction, write "abort" or "stall" in the table entry at the row for that action and the column for P together with the state of P (e.g. "S, stall"). If a transaction aborts, assume that Tend does nothing.

Processor Action	Hit / Miss	P1 state	P2 state	P3 state	Data comes from
P1,P2, P3 Tbeg		--	--	--	
P1 read x					
P3 read x					
P2 read x					
P1 Tend, Tbeg					
P2 Tend, Tbeg					
P1 read x					
P3 write x					
P2 read x					
P1 Tend					
P3 Tend					
P2 Tend					

Implementing Transactions

PRACTICE PROBLEM 6:

In this problem we will explore the implementation of an **optimistic read, pessimistic write, eager versioning** software TM (STM). The STM operates over 32-bit values.

In your implementation, each transaction is encapsulated by a `Txn` object that maintains a local timestamp for the transaction as well as the transaction's read and write sets. Your implementation should have the following properties:

1. A global timestamp and a single global lock to protect commits.
2. A transaction's local timestamp is the value of the global timestamp when the transaction starts.
3. A table that maps memory locations to a version number.
4. Writes are stored to a write log `wset` as `(address, value)` pairs.
5. The version of committed writes is the current global timestamp; committing also increments the global timestamp.
6. The read set is validated on commit; if any read location has a version number greater than the local timestamp the transaction retries.

The skeleton code for the transactional memory system is given on the next page. You should write your answers in the space provided on the page after that. **Code for `__begin` is provided, and you should provide code for `read`, `write`, and `commit`.** Don't get hung up on syntax; we don't expect you to pen down flawless, compilable C code - some pseudocode is acceptable as long as its meaning is clear. Example details of importance: What is added to the read and write sets, when are locks taken, when are conflicts validated (and how?).

```
// setjmp stores a snapshot of the registers (stack pointer, instruction pointer, etc.) into
// a buffer (t.rollback). A future call to longjmp restores the saved register values and thus
// restarts control flow at the point when setjmp was called.
#define TXN_BEGIN(t)    \ // TXN_BEGIN is called to begin a transaction.
        setjmp(t.rollback); \
        t.__begin();

typedef uint64_t timestamp_t;

class Txn {
public:
  Txn() {}
  virtual ~Txn() {}
  void retry() { longjmp(rollback, 1); } // return control flow to context saved by setjmp
  void __begin();

  void write(uint32_t* p, uint32_t v); // Students implement this!
  uint32_t  read(uint32_t* p);         // Students implement this!
  void      commit();                  // Students implement this!

  jmp_buf       rollback;
  typedef std::map<uint32_t*, mutex_t> write_lock_t;  // Used for write locks
  write_lock_t   wlock;  // wlock is a map, so wlock[p] is the lock for the object p

private:
  #define TABLE_SZ    4096
  timestamp_t   local_timestamp;

  static timestamp_t get_version(uint32_t* p) {
    return versions[(((intptr_t)(p)) / 4) % TABLE_SZ];
  }
  static void set_version(uint32_t* p, timestamp_t t) {
    versions[(((intptr_t)(p)) / 4) % TABLE_SZ] = t;
  }

  // Used to log writes
  typedef std::map<uint32_t*, uint32_t> write_set_t;
  write_set_t   wset;

  // Used to keep track of reads that this transaction has made
  typedef std::set<uint32_t*> read_set_t;
  read_set_t    rset;

  // Used to map memory addresses to a timestamp (e.g. to indicate most recent use)
  static timestamp_t  versions[TABLE_SZ];
  static timestamp_t  global_timestamp;
  static mutex_t      commit_lock;
};

/////implementation file//////////
timestamp_t  Txn::global_timestamp = 0;  // system-wide global
mutex_t      Txn::commit_lock;           // system-wide global
timestamp_t  Txn::versions[TABLE_SZ];    // system-wide global

void Txn::__begin(void) {
  wset.clear();
  rset.clear();
  local_timestamp = global_timestamp;
}
```

```
void Txn::write(uint32_t* p, uint32_t v) {
  // YOUR CODE HERE
  // Your code can assume that a mutex supports lock(), unlock(),
  // and trylock() operations. Trylock() returns true if the lock
  // is currently locked, false otherwise.
  // e.g., trylock(wlock[p]) checks to see if the lock on object p is currently taken.
















}

uint32_t Txn::read(uint32_t* p) {
  // YOUR CODE HERE!
  // Your code can assume that a mutex supports lock(), unlock(),
  // and trylock() operations. Trylock() returns true if the lock
  // is currently locked, false otherwise
  // e.g., trylock(wlock[p]) checks to see if the lock on object p is currently taken.














}
```

```
void Txn::commit() {

  mutex_lock(&commit_lock);

  // YOUR CODE HERE!
  // Your code can assume that a mutex supports lock(), unlock(),
  // and trylock() operations. Trylock() returns true if the lock
  // is currently locked, false otherwise
  // e.g., trylock(wlock[p]) checks to see if the lock on object p is currently taken.













  mutex_unlock(&commit_lock);
}
```

Implementing Transactions

PRACTICE PROBLEM 7:

Below is an implementation of an **optimistic read, pessimistic write, lazy versioning** STM that tracks reads and writes at the granularity of objects.

```
// Assume TMDesc is a transaction descriptor data structure
//    -- use GetDataVersion(obj) to access the version
// Assume each object maintains a lock, which supports these two ops:
//    -- LockObj(obj) returns true if success, false if failure
//    -- CheckLock(obj) returns true if locked, false otherwise
//    -- ReleaseLock(obj)

// helper routines ///////////////////////////////////////////////////

// add object to read set
void OpenForReadTx(TMDesc tx, object obj) {
  tx.readSet.obj = obj;
  tx.readSet.version = GetDataVersion(obj);
  tx.readSet++;
}

// add object to write set
OpenForWriteTx(TMDexc tx, object obj) {
  if(!LockObj(obj) {      // try to lock object for writing
     AbortTx(tx);         // abort if someone else holds the lock
  }
  tx.writeSet.obj = obj;
  tx.writeSet.version = GetDataVersion(obj);
  tx.writeSet++;
}

// offset denotes what field of the object is being written to and needs to be buffered
// aka... conflict detection is at object granularity, but write logging at field granularity.
void writeBuffIntInsertTx(TMDesc tx, object obj, int offset, int value) {
  tx.writeBuff.obj = obj;
  tx.writeBuff.offset = offset;
  tx.writeBuff.value = value;  // buffer the value
  tx.writeBuff++;
}

  // helper: returns the int corresponding to the appropriate offset in object obj
int ReadDataFromMem(object obj, int offset);

// helper: writes value to the appropriate location in memory
void WriteDataToMem(object obj, int offset, int value);

void AbortTx(TMDesc tx); // call this internal helper to abort the transaction

// you will implement ReadIntTx in part A
int ReadIntTx(TMDesc tx, object obj, int offset) { }

// you will implement CommitTx in part A
bool CommitTx(TMDesc tx) { }

// you will implement UnlockObj in part A
void UnlockObj(object obj, TMVersion version) { }
```

A. Provide implementations for `ReadIntTx` (which reads an `int` value), `CommitTx()` (which commits transactions) and `UnlockObj()` (which commits writes) to ensure correct operation of this STM. Pseudocode is fine, but it must be sufficiently precise for the grader. **Hint: Be careful: how do you ensure your reads get the most up to date data?**

```
int ReadIntTx(TMDesc tx, object obj, int offset) {
  // transaction is performing a read to a field of obj (given by offset)









}

void UnlockObj(object obj, TMVersion version) {
  // this is the symmetric call to LockObj. Remember the lock is taken when
  // a pending transaction writes so unlock must be called when the writing
  // transaction commits.


















}

bool CommitTx(TMDesc tx) {
   // recall *optimistic* reads, *pessimistic* writes
   // if there is a conflict, this transaction should abort
















}
```

B. **THE REST OF THIS PROBLEM IS INDEPENDENT OF YOUR ANSWERS FROM PART A. NOTICE THAT THE TM SYSTEM IS NOW EAGER.** Assuming **optimistic reads, pessimistic writes and eager versioning**, fill in the tables below for the two concurrent transactions X1 and X2. **Assume all data and version numbers are initialized to zero.** Assume the transactions proceed concurrently, and the operations occur in order listed to the left of each statement. e.g., (1) is the first operation to occur in time, (2) is the second, etc. (5) and (7) are the commit times.

```
        X1                      X2
    ------------------------------------------------
        atomic {                atomic {
(1)       obj1.x = 5      (2)     t1 = obj1.x
(4)       obj2.x = 6      (3)     t2 = obj2.x
(5)     }                 (6)     obj3.x = t2 + 1;
                          (7)   }
```

After step (1) in the figure above, the state of the system looks like this: (The table below shows the metadata for all objects, as well as the read/write sets for all transactions, as well as the state of the transaction undo logs.

	X	version	Locked by
Obj1	5	0	X1
Obj2	0	0	-
Obj3	0	0	-

	Read Set	Write Set	Undo Log
X1	{}	{(obj1, 0)}	{(obj1.x, 0)}
X2	{}	{}	{}

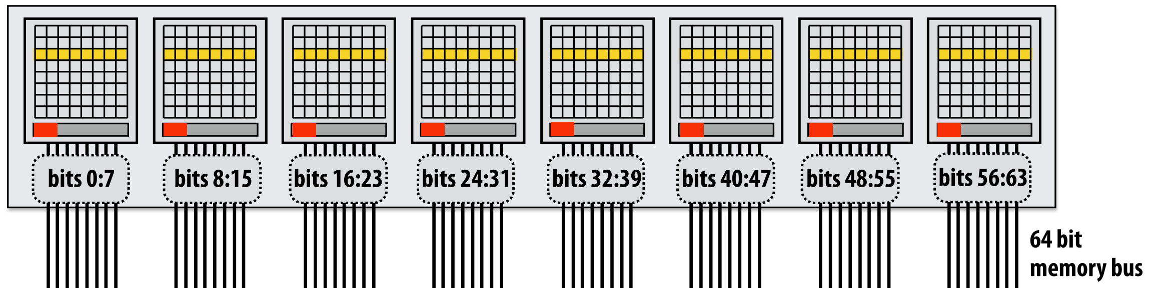
Now please fill in the table to describe the state of the system after step (6) in the figure above:

	X	version	Locked by
Obj1			
Obj2			
Obj3			

	Read Set	Write Set	Undo Log
X1			
X2			

PRACTICE PROBLEM 8:

Consider a DRAM DIMM with 8 chips (8-bit interface per chip) just like what we talked about in class. Physical memory addresses are strided across the chips as in the figure below, so that 64 consecutive bits from the address space can be read in a single clock over the bus. The DRAM row size is 2 kilobits (256 bytes). There is only a single bank per chip. (We ignore banking in this problem.)



The memory controller processes requests with the following logic:

```
int active_row;   // stores active row

handle_64bit_request(void* addr) {

   int row, col;

   compute_row_col(addr, &row, &col);   // compute row/col from addr (0 cycles)

   if (row != active_row)
      activate_row(row);     // this operation takes 15 cycles

   transfer_column(col);     // this operation takes 1 cycle
}
```

Questions are on the next page...

Now consider the following C-program, which executes using two threads on a dual-core processor with a single shared cache.

```
struct ThreadArg {
  int threadId;
  double sum;  // thread-local variable
  int N;       // assume this is very large
  double* A;   // pointer to shared array
};

// each thread processes one half of array A
void myfunc(ThreadArg* arg) {
  arg->sum = 0.f;
  int offset = arg->threadId * arg->N / 2;
  for (int i=0; i<arg->N / 2; i++)
    arg->sum += arg->A[offset + i];
}

/* main code */

ThreadArg args[2];
args[0].threadId = 0; args[1].threadId = 1;
args[0].A = args[1].A = new double[N];

// initialize args[].sum, args[].N, args[].A, and launch two threads here that run myfunc
// Then wait for threads to complete

print("%f\n", args[0].sum + args[1].sum);
```

A. Assume that the two threads run at approximately the same speed, so the memory controller receives requests from the two threads in interleaved order: thread0_req0, thread1_req0, thread0_req1, thread1_req1, etc. Given this stream, what is the effective bandwidth of the memory system as observed by the processor (the rate at which it receives data)? Assume that:

   - The program is bandwidth bound so that the memory system always has a deep queue of requests to process.
   - The granularity of transfer between the memory controller and the cache is 64 bits. (e.g., 8-byte cache line size)
   - Note that array elements are DOUBLES (8 bytes).

B. Modify the program code to significantly improve the effective memory system bandwidth. What is the new bandwidth you observe?

C. Return to the original code given in this assignment (ignore your solution to part B), and assume that requests now arrive at the memory controller every ten cycles. For example...

```
cycle 0: thread 0 req 0
cycle 10: thread 1 req 0
cycle 20: thread 0 req 1
cycle 30: thread 1 req 1
cycle 40: thread 0 req 2
cycle 50: thread 1 req 2
cycle 60: thread 0 req 3
...
```

Write (rough) pseudocode for a memory request scheduling algorithm that allows the memory system to keep up with this request stream. **Your implementation can assume there is an incoming request buffer called `request_buf` that holds up to 4 requests.** (The processor stalls if the request buffer is full.)

D. (TRICKY!) You add hardware multi-threading to your dual-core processor (2 threads-per core) and modify your code to spawn four threads. You assign contiguous blocks of the input array to each thread. Assuming the request arrival rate stays the same (but now requests from four threads, rather than two, are interleaved), how would you change your solution in part C to keep up with the request stream? (you may modify the buffer size if need be). Is overall memory latency higher or lower than in part C? Why?