Stanford CS149: Parallel Computing
Written Assignment 5 SOLUTIONS

Feeling Relaxed

Problem 1. (20 points):

A. (10 pts) 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.**

Solution: P1 will print "10" since there is a dependency of that read on the prior write to X in the same instruction stream.

P3 may print "0" or "10". Even though P3 has observed the write to `flag`*, there is no guarantee that the write to X has been propagated to P3 yet. (Note that P2 observed the write to X before the write to* `flag` *(because it printed "10", but this says nothing about P3's observations due to the relaxed memory ordering.*

If P3 prints "0", then the system is not producing output that is possible under a sequentially consistent memory system, since there is no timeline of memory operations that is consistent with the observed results. P1 and P2 observe results that suggest the write to X occurred prior to the write to `flag`*, whereas P3's observation suggests the write to* `flag` *occurred prior to the write to X.*

A common mistake was to report that P1 can print "0" as well. This is not correct because W->R dependencies are not relaxed. Basically, any variation of the correct solution above would have just received partial credit.

B. (10 pts) 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.

Solution: A single write fence placed between the two write operations of P1 is sufficient. The fence ensures that all processors observe the write to X prior to observing the write to `flag`*. All threads will now print "10".*

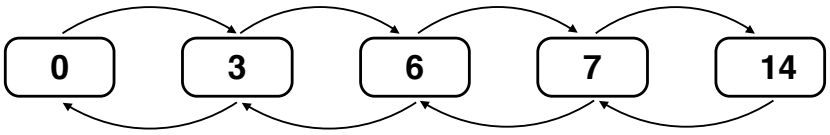
Some students added 2 write fences in P2 and P3 (right after the `while` *loop). Though this may be correct, this is not the minimal number of wfences required. This solution only received partial credit.*

Transactions on a Doubly Linked List

Problem 2. (40 points):

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. (10 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?

   Solution: Yes. There might still be high contention and a loss of concurrent execution due to many transaction aborts, but the optimistic transactional implementation would execute to completion correctly.

B. (10 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!)**

Solution: `delete_front(14)` *aborts,* `insert_front(4)` *does not. In optimistic systems, the first transaction to begin its commit wins. In this case* `insert_front(4)` *writes to values (it updates the data structure) that* `delete_front(14)` *reads. This is a conflict (values accessed by* `delete_front(14)` *were changed by a commit while* `delete_front(14)` *was in progress), so* `delete_front(14)` *is aborted given the conflict detection rules of optimistic transactions.*

C. (10 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.**

Solution: No aborts are necessary. `delete_front(14)` *only reads values touched by* `insert_front(4)`*. Therefore,* `insert_front(3)` *observes data that is consistent with the state of memory at the end of* `delete_front(14)` *and thus there is no conflict and no need to abort. The system's behavior is consistent with the serial order:* `delete_front(14)`*, then* `insert_front(4)`*.*

D. (10 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?

Solution: The answer is the same as in part C. The fact that the transaction for `insert_front(4)` now includes writes is insignificant to the behavior of `delete_front(14)` because `delete_front(14)` commits first. The writes by `insert_front(4)` are not visible outside the transaction at this point (they have not committed) and thus they have not yet logically occurred. In other words, writes by `insert_front(4)` do not cause conflicts with `delete_front(14)` since they logically occur after `delete_front(14)`.

Coherence and Transactional Memory

Problem 3. (40 points):

A. (20 pts) 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.)

Solution:

Processor Action	Hit or 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	Miss	S	S	S	Mem
P1 Tend, Tbeg		S	S	S	
P2 Tend, Tbeg		S	S	S	
P1 read x	Hit	S	S	S	P1 cache
P3 write x	Miss	I, abort	I	E	P3 cache
P2 read x	Miss	I	I, stall	E	
P1 Tend		I	I, stall	E	
P3 Tend		I	S	S	P3 cache
P2 Tend		I	S	S	

Note: an assumption of the solution above is that the second transaction by P1 (the one that is aborted) is not restarted during the time interval spanned by the table, and thus the second Tend for P1 is treated as a no-op following the instructions in the problem. Students that attempted to fill out the rest of the table assuming that P1's transaction resumes immediately and issues a Read of X (and thus should need to stall until P3 commits just like P2 stalls), and indicated this on their solution, may want to confirm their solution was graded correctly.

B. (20 pts) 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.

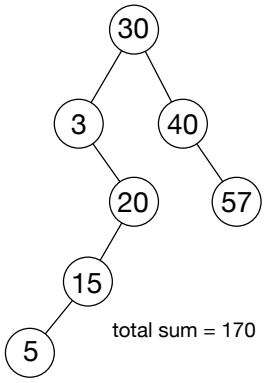
Given the rules above, show what happens to the cache line state for address X for references made by three processors (P1, P2, P3) by filling in the table below. Initially none of the caches contain address X. If an action causes a processor P to abort or stall a transaction, write "abort" or "stall" 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.

Solution:

Processor Action	Hit or 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	Miss	S	S	S	Mem
P1 Tend, Tbeg		S	S	S	
P2 Tend, Tbeg		S	S	S	
P1 read x	Hit	S	S	S	P1 cache
P3 write x	Hit	S	S	SW	P3 cache
P2 read x	Hit	S	S	SW	P2 cache
P1 Tend		S	S	SW	
P3 Tend		I	I, abort	E	
P2 Tend		I	I	E	

Practice Problem 1: Transactions on Trees

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.)

There are only two. 20 –> 25 –> 24 or 20 –> 24 –> 25

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

Solution: 170, 180, 194, 195, 204, 205, 219, 229

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?

*Solution: Definitely not! The choice of how to **implement** a transaction cannot change the **semantics** of the transactional abstraction.*

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.**

Solution: Only `sum` *is aborted since the write set of the committing transaction (which is node 5) conflicts with the read set of* `sum`*. Note that there is no conflict with the other insertions since they read no data written by* `insert(10)`*.*

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.**

Solution: In this case, `insert(10)` *is aborted since 20 is in its read set, and it was modified by the committing transaction.* `insert(24)` *is also aborted since it's read and write sets conflict with the write set of the committing transaction.* `sum` *is not aborted since it hasn't progressed enough to reach node 20.*

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.

Solution: The problem is that `insert(24)` *can write to* `n->right` *of node 20, which conflicts with* `insert(25)`*'s read/write of the same node.* `insert(25)` *will abort and restart, and then it's own update of* `n->right` *on the retry will cause* `insert(24)` *to abort if that transaction did not have time to commit.*

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*.

Solution: Any standard answer from the implementation of locks lecture would work, but in this context the solutions are used for implementing transactions, not locks: you could try random back-off, give priority to a transaction that's been aborted too many times in the past, put transactions that have aborted in a list and process the list serially (like a ticket lock), etc.

Practice Problem 2: Implementing Transactions

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?).

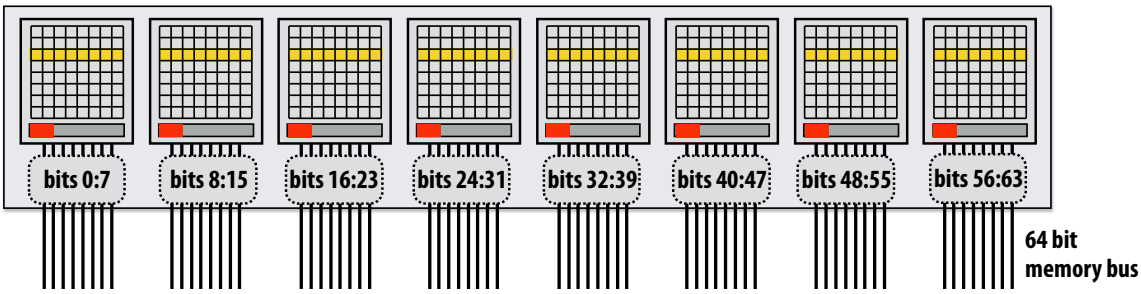
```
// setjmp stores a snapshot of the registers (stack pointer, instruction pointer, etc.) into
// a buffer (t.rollback). A future call to longjmp restores the saved register values and thus
// restarts control flow at the point when setjmp was called.
#define TXN_BEGIN(t)    \ // TXN_BEGIN is called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

```
uint32_t Txn::read(uint32_t* p) {

  // even though its an optimistic system, must check at this point
  // for read-write conflict with a non-committed transaction with
  // the same local version number that might have already
  // performed a write. (since it's eager, that write has gone out to
  // and we're about to observe it's effects.  Note that if a conflicting
  // write has not yet happened, we continue optimistically and check for
  // read-write conflicts upon commit -- true to the eager specification.)

  if (mutex_trylock(wlock[p])) {
    // another transaction is writing, so there's a conflict. Must abort,
    // so have to roll back all writes
    foreach (it, wset.begin(), wset.end()) {
        uint32_t* p = it->first;
        *p = it->second;
        mutex_unlock(wlock[p]);
    retry();
  }

  rset.insert(p);
  return *p;
}

void Txn::write(uint32_t* p, uint32_t v) {
  if (mutex_trylock(wlock[p])) {
    // another transaction is writing, so there's a conflict. Must abort,
    // so have to roll back all writes
    foreach (it, wset.begin(), wset.end()) {
        uint32_t* p = it->first;
        *p = it->second;
        mutex_unlock(wlock[p]);
    }
    retry();
  }
  // take lock on object and add to write set
  mutex_lock(wlock[p]);
  wset[p] = *p;
  *p = v;
}

void Txn::commit() {
  mutex_lock(&commit_lock);
  // validate, must check if the read set doesn't intersect with
  // another committed transaction
  if (global_timestamp > local_timestamp) {
    foreach (it, rset.begin(), rset.end()) {
      if (get_version(*it) > local_timestamp) {
            // abort, so roll back
            foreach (it, wset.begin(), wset.end()) {
                uint32_t* p = it->first;
                *p = it->second;
                mutex_unlock(wlock[p]);
            }
      }
      mutex_unlock(&commit_lock);
      retry();
    }
  }
}
```

```
  /* write update log */
  global_timestamp++;
  foreach (it, wset.begin(), wset.end()) {
    uint32_t* p = it->first;
    set_version(p, global_timestamp);
    mutex_unlock(wlock[p]);
  }
  mutex_unlock(&commit_lock);
}
```

Practice Problem 3: Controlling DRAM

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).

   Solution: Because of the interleaved request streams, each memory access will trigger a row-buffer miss. As a result, the memory system needs 16 clocks to provide 64 bits (8 bytes) back to the processor. The overall effective bandwidth is 8/16 = 0.5 bytes per clock (or 4 bits per clock)

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

Solution: Change the blocked assignment of array elements to threads to an interleaved assignment. As a result of this change the memory system receives a stream of contiguous addresses, and the system achieves maximum row buffer locality. The system will obtain very near its peak bandwidth of 8 bytes per clock.

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.)

Solution: The main idea is to buffer requests and reorder how they are serviced so that two requests from the same stream are handled back-to-back. As a result, the second request of the pair will be a row-buffer hit. Specifically this scheme will complete the incoming requests at the following times:

```
t0, req 0: 16      buffer: t1:r0
t1, req 0: 32      buffer: t0:r1, t1:r1
t1, req 1: 33      buffer: t0:r1
t0, req 1: 49      buffer: t0:r2
t0, req 2: 50      buffer: t1:r2
t1, req 2: 66      buffer: t0:r3 ...
```

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?

Solution: The same scheme used in part C still works (including a length 4 buffer), but now requests to adjacent addresses (in the same DRAM row) come every 4th request and are separated in time by 40 cycles. On average memory latency is a bit longer than in part C.

```
t0, req0: 16   buffer = t1:r0
t1, req0: 32   buffer = t2:r0, t3:r0
t2, req0: 48   buffer = t3:r0, t0:r1
t3, req0: 64   buffer = t0:r1, t1:r1, t2:r1
t0, req1: 80   buffer = t1:r1, t2:r1, t3:r1, t0:r2
t0, re12: 81   buffer = t1:r1, t2:r1, t3:r1
t1, req1: 97   buffer = t2:r1, t3:r1, t1:r2
t1, req2: 98   buffer = r2:r1, t3:r1 ...
```