15-418/618, Spring 2018
Exercise 2

Assigned:	Mon., Mar. 26
Due:	Fri., Mar. 30, 1:30 pm

You will submit an electronic version of this assignment to Gradescope as a PDF file. For those of you familiar with the LaTeX text formatter, you can download a template file at:

http://www.cs.cmu.edu/~418/exercises/ex2-answer.tex

Instructions for how to use this template are included as comments in the file. Otherwise, you can use the solution template:

http://www.cs.cmu.edu/~418/exercises/ex2-answer.pdf

You can either: 1) electronically modify the PDF file, or 2) print it out, write your answers by hand, and scan it. In any case, we expect your solution to follow the formatting of the solution template.

Problem 1: Lock-Free Programming

Rather than implementing a compare-and-swap instruction, some processors provide a pair of instructions: `LL` (for "load link") and `SC` (for "store conditional"). These can be described by the following functions. In the following, we use type `data_t` to indicate an arbitrary data type, e.g., these functions can operate on data of type `int`, pointer, etc.

```
data_t load_link(data_t *addr) {
    data_t val = *addr;
    clean[addr] = true;
    return val;
}

boolean store_conditional(data_t *addr, data_t val) {
    if (clean[addr]) {
        *addr = val;
        return true;
    }
    return false;
}
```

In this code, the array of flags `clean` is local to the thread, but if any instruction executed by any thread modifies the value at `addr`, the flag must be set to `false`.

1A: Implementing LL/SC

Suppose each CPU has a private L1 cache and that cache coherency is maintained by a bus-based MSI protocol. For each cache block, we add a one-bit flag to implement the flag `clean` for the entire line. Here is how the instructions could be implemented:

Load: When an ordinary memory read instruction must load the cache line, set the `clean` flag to false. When the line is already in the cache, don't change the value of its `clean` flag.

Store: Any memory write instruction should set the `clean` flag to false.

BusRdX: If an external request comes for exclusive use of this line, set the line to invalid.

`LL`**:** Like a regular load, except that it sets the `clean` flag for the line to true.

`SC`**:** Perform the store only if the cache line is valid and the `clean` flag is true, by first upgrading the line to the exclusive state and then updating the copy. Set the `clean` flag to false.

1. Suppose the CPU supports two execution contexts. Describe a scenario in which the proposed implementation would fail.

2. Describe how you would modify the design to support $T > 1$ execution contexts per CPU.

1B: Implementing Atomic Primitives

The CAS ("compare and swap") operation implements the following functionality, but it does so atomically.

```
int CAS(int *addr, int compare, int new) {
    int old = *addr;
    if (old == compare)
        *addr = new;
    return old;
}
```

1. Fill in the code below to implement CAS with LL/SC, in a way that *appears* to operate atomically.

```
int CAS(int *addr, int compare, int new) {




}
```

2. Is your solution guaranteed to terminate? Explain.

1C: Lock-free Stack Implementation

Referring to the lecture on fine-grained synchronization and lock-free programming:

http://www.cs.cmu.edu/~418/lectures/17_lockfree.pdf.

Slide 28 provides code to implement a lock-free stack using compare and swap. The following is the code for the `pop` operation

```
Node* pop(Stack* s) {
  while (1) {
    Node* old_top = s->top;
    if (old_top == NULL)
      return NULL;
    Node* new_top = old_top->next;
    if (compare_and_swap(&s->top, old_top, new_top) == old_top)
      return old_top;  // Assume that consumer then recycles old_top
  }
}
```

1. Rewrite the code to use load link/store conditional.

   ```
   Node* pop(Stack* s) {



   }
   ```

2. The compare-and-swap code of Slide 28 was shown to have an ABA problem, which could cause it to fail to maintain the stack properly. Does your implementation using load-link/store conditional have an ABA problem? Explain.

Problem 2: Synchronization and Transactional Memory

Suppose we have a transactional memory system with the following library operations:

```
// Begin transaction
void xbegin();

// (Attempt to) commit transaction.
// Returns true if successful, false if failed
boolean xend();
```

When using transactional memory, it is important that any read or write of any shared variable take place within a transaction. In addition, the only way to be sure a shared variable has been written is to check that the `xend` operation returns true.

2A: Implementing Locks

Suppose we want to implement a simple, mutual-exclusion lock with the following API:

```
// Block until lock is acquired
void acquire(int *lock);

// Release lock
void release(int *lock);
```

Fill in the following code to show how you could do this with transactional memory

1. Acquire

   ```
   void acquire(int *lock) {



   }
   ```

2. Release

   ```
   void release(int *lock);



   }
   ```

2B: A Banking System with Locks

Consider the following very simple banking system, where all account values are whole dollars, and you are provided a mutual exclusion lock for each account:

```
// Account balances in whole dollars
int balance[NACCT];
// Per account locks.
// Support operations acquire() and release()
int lock[NACCT];
```

The only supported banking operations are to transfer money between accounts, and to check that the sum of all balances matches some expected value. If a transfer operation would cause an account to become overdrawn, it should return false without changing any balances. (You can assume that `amount` is not negative.)

```
// Transfer money between accounts.
// Do not allow from_acct to become negative
boolean transfer(int to_acct, int from_acct, int amount);

// Check that sum of all accounts matches expected value
boolean check_sum(int expected);
```

Function `check_sum` must be implemented in such a way that it gets an accurate picture of sum of all of the balances, even while transfer operations are taking place.

Fill in code for these two operations using locking. You may not add any additional locks. Make sure your code cannot deadlock.

1. Transfer

```
boolean transfer(int to_acct, int from_acct, int amount) {




}
```

2. Check Sum

```
boolean check_sum(int expected);




}
```

2C: A Banking System with Transactional Memory

Now show a possible implementation using the transactional memory primitives.

1. Transfer

   ```
   boolean transfer(int to_acct, int from_acct, int amount) {



   }
   ```

2. Check Sum

   ```
   boolean check_sum(int expected);



   }
   ```

3. If the system performs millions of transactions per second over thousands of bank accounts, what problem to you forsee for the implementation based on transactional memory?