Full Name: Harry Q. Bovik
Andrew Id: `bovik`

15-418/618
Exercise 2 SOLUTION

Problem 1: Lock-Free Programming

1A: Implementing LL/SC

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

   Thread 1 executes load-link. Then Thread 2 writes to the same address and executes load-link. If Thread 1 now executes store conditional, it will succeed, even though the value at the address has changed.

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

   For each cache line, have T copies of the `clean` flag. The protocol operates as described, with the load, store, and BusRdX operations updating all T flags, while the load-link and store conditional operations only operating on the flag for the particular thread.

1B: Implementing Atomic Primitives

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) {
    while (true) {
        int old = load_link(addr);
        if (old == compare) {
            if (store_conditional(addr, new))
                return old;
        } else
            return old;
    }
}
```

2. Is your solution guaranteed to terminate? Explain.

   No. The while loop could continue indefinitely if writes to the the address cause the store conditional to fail.

1C: Lock-free Stack Implementation

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

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

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 problems? Explain.

   There is no ABA problem here, since the store conditional would fail if the value at `s->top` were altered, even if it then reverted back.

Problem 2: Synchronization and Transactional Memory

2A: Implementing Locks

1. Acquire

```
void acquire(int *lock) {
    while (true) {
        xbegin();
        boolean done = lock > 0;
        if (done)
            lock = 0;
        if (xend() && done)
            return;
    }
}
```

2. Release

```
void release(int *lock);
    while (true) {
        xbegin();
        lock = 1;
        if (xend())
            return;
    }
}
```

2B: A Banking System with Locks

1. Transfer

```
boolean transfer(int to_acct, int from_acct, int amount) {
    int amin = min(from_acct, to_acct);
    int amax = max(from_acct, to_acct);
    // Acquire locks in ascending order of account number
    acquire(&lock[amin]);  acquire(&lock[amax]);
    boolean ok = balance[from_acct] >= amount;
    if (ok) {
        balance[from_acct] -= amount;
        balance[to_acct] += amount;
    }
    release(&lock[amax]); release(&lock[amin]);
    return ok;
}
```

2. Check Sum

```
boolean check_sum(int expected);
    int a;
    int sum = 0;
    // Must lock down all accounts.  Can sum as acquire locks
    for (a = 0; a < NACCT; a++) {
        acquire(lock[a]);
        sum += balance[a];
    }
    for (a = 0; a < NACCT; a++)
        release(lock[a]);
    return sum == expected;
}
```

2C: A Banking System with Transactional Memory

1. Transfer

```
boolean transfer(int to_acct, int from_acct, int amount) {
    while (true) {
        xbegin();
        boolean ok = balance[from_acct] >= amount;
        if (ok) {
            balance[from_acct] -= amount;
            balance[to_acct] += amount;
        }
        if (xend())
            return ok;
    }
}
```

2. Check Sum

```
boolean check_sum(int expected)
    while (true) {
        int a;
        int sum = 0;
        xbegin();
        for (a = 0; a < NACCT; a++) {
            sum += balance[a];
        }
        boolean ok = sum == expected;
        if (xend())
            return ok;
    }
}
```

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?

   Chances are that check sum would keep failing, because ongoing transfers would keep causing writes to the array `balance`.