There is some input to `handle_transactions` below that will cause a deadlock.

```
struct Bank {
  int num_accounts;
  int *balances; // one for each account
  Mutex *account_mutexes; // one for each account
}

Bank bank;

struct Transaction {
  int from_index;
  int to_index;
  int amount;
}

void handle_transaction(Transaction *trans) {
  lock(&bank.account_mutexes[trans->from_index]);
  lock(&bank.account_mutexes[trans->to_index]);
  bank.balances[trans->from_index] -= trans->amount;
  bank.balances[trans->to_index] += trans->amount;
  unlock(&bank.account_mutexes[trans->to_index]);
  unlock(&bank.account_mutexes[trans->from_index]);
}

void handle_trasactions(Transaction *transactions, int len) {
  cilk_for (int i = 0; i < len; i++) {
    handle_transaction(&transactions[i]);
  }
}
```

True **False**
If true, explain why and how to fix it. If false, explain.

Any mutex for n threads which is implemented using a Compare-and-Swap (CAS) atomic operation requires $\Omega(n)$ space.
True **False**

Suppose that a Cilk program behaves deterministically when executed on a single processing core. If each concurrent access to shared data in the program is protected by the same mutex, then the program continues to behave deterministically when executed in parallel.

True **False**

Consider a dual-core x86 processor with two processing cores, named Albert and Bernard. The two cores concurrently execute the following instructions, operating on volatile variables x and y, which are both initialized to 0.

Albert

```
if (x == 0)
  y = x + 1;
```

Bernard

```
if (y == 0)
  x = y + 1;
```

After Albert and Bernard have both finished executing all of their instructions, which of the following final values of x and y are possible?

A. $x = 0$, $y = 1$

B. $x = 0$, $y = 2$

C. $x = 1$, $y = 1$

D. $x = 1$, $y = 2$

E. $x = 2$, $y = 2$

Explicit memory fences are unnecessary on a processor that provides sequentially consistent memory.

True **False**

A memory fence is expensive, because it prohibits all other processing cores in the system from using the memory system until the core executing the fence has committed all of its outstanding stores.

True **False**