TEMPLATE: FILL THIS IN TO MAKE YOUR OWN LOCK

```
typedef struct __lock_t {
    // whatever data structs you need goes here
} lock_t;

void init(lock_t *lock) {
    // init code goes here
}

void acquire(lock_t *lock) {
    // lock acquire code goes here
}

void release(lock_t *lock) {
    // lock release code goes here
}
```

FIRST PRIMITIVE: TEST-AND-SET (or ATOMIC EXCHANGE)

```
// given ptr, sets *ptr to new value; returns the old value at *ptr
int TestAndSet(int *lock, int new) {
    int old = *lock;
    *lock = new;
    return old;
}
```

SECOND PRIMITIVE: COMPARE-AND-SWAP

```
int CompareAndSwap(int *ptr, int expected, int new) {
    int actual = *ptr;
    if (actual == expected)
            *ptr = new;
    return actual;
}
```

THIRD PRIMITIVE(S): LOAD-LINKED, STORE-CONDITIONAL

```
int LoadLinked(int *ptr) {
    return *ptr;
}

int StoreConditional(int *ptr, int value) {
    if (no one has updated *ptr since LoadLinked to this address) {
            *ptr = value;
            return 1; // success
    } else {
            return 0; // fail (does not do the store)
    }
}
```

FOURTH PRIMITIVE: FETCH-AND-ADD

```
int FetchAndAdd(int *ptr) {
    int old = *ptr;
    *ptr = old + 1;
    return old;
}
```

EXAMPLE USING FETCH-AND-ADD: TICKET LOCKS

```
typedef struct __lock_t {
    int ticket;
    int turn;
} lock_t;

void lock_init(lock_t *lock) {
    lock->ticket = 0;
    lock->turn   = 0;
}

void acquire(lock_t *lock) {
    int myturn = FetchAndAdd(&lock->ticket);
    while (lock->turn != myturn)
        ; // spin
 }

void release(lock_t *lock) {
    FetchAndAdd(&lock->turn);
}
```