CMSC 23000	**Operating Systems**	**Homework 1**
Autumn 2006		**Due October 10**

Software transactional memory is a popular programming abstraction for concurrent systems. The basic idea is that operations on shared variables are grouped into a *transaction* that either *commits* as a group, if there is no interference with other transactions, or *aborts*. The following API defines an STM mechanism that could be used in C programs:

```
typedef ... stm_var_t;
typedef ... transaction_t;

transaction_t atomic_begin ();
bool_t atomic_end (transaction_t);

void stm_init (stm_var_t *);
void stm_update (transaction_t, stm_var_t *, void *);
void *stm_read (transaction_t, stm_var_t *);
```

The type `stm_var_t` is an STM variable that can hold a pointer-sized value. STM variables can only be read and written inside an atomic transaction, which are identified by the opaque type `transaction_t`. The function `atomic_begin` initiates a transaction and returns the ID of the transaction and `atomic_end` terminates the transaction and returns `true` if it successfully committed. For example, the following code fragment uses an STM variable to implement a shared counter.

```
// shared counter
  stm_var_t counter;
  ...
// atomic increment
  bool success = false;
  do {
      transaction_t t = atomic_begin();
          int x = (int)stm_read (t, &counter);
          stm_write (t, &counter, (void *)(x+1));
      success  = atomic_end ();
  } while (! success);
```

Your assignment is to implement the STM API from above using the following operations:[1]

```
typedef ... mutex_t;
void mutex_init (mutex_t *);
void mutex_lock (mutex_t *);
void mutex_unlock (mutex_t *);
```

Hint: use a log to record STM updates and then apply them at the end of the transaction, but lookout for deadlock!

[1] These are a simplification of the POSIX thread operations.