5 Lock-free Synchronization

Design data structures in a way that allows safe concurrent access.

- No mutual exclusion necessary.
- No possibility of deadlock.

Approach: use a single atomic operation to

- commit all changes in a state that satisfies the preconditions
- thus safely moving the shared data structure from one consistent state to another

Example 1: Wait-free bounded buffer for one producer and one consumer: (uses busy-waiting for condition synchronization)

```
char buffer[BUF-SIZE];
int head = 0;
int tail = 0;

void Produce(char item) {
    while(((tail + 1) % BUF-SIZE) == head);
    buffer[tail] = item;
    tail = (tail + 1) % BUF-SIZE;
}

char Consume() {
    char item;
    while(tail == head);
    item = buffer[head];
    head = (head + 1) % BUF-SIZE;
    return item;
}
```

Example 2: Wait-free singly-linked queue for any number of threads

```
QElem *queue;

void Insert(Item item) {
    QElem *new = malloc(sizeof(QElem));
    new->item = item;
    do {
        new->next = ldl(&queue);
    } while(!stc(&queue, new));
}

bool isIn(Item item) {
    QElem *current = queue;
    while(current != NULL) {
        if (current->item == item) return TRUE;
        current=current->next;
    }
    return FALSE;
}

QElem *remove() {
do {
    if (queue == NULL) return NULL;
    QElem *current = ldl(&queue);
} while(!stc(&queue, current->next));
return current;
}
```

When is it safe to free the QElem returned by remove()?

This only works for simple data structures where changes can be committed by updating one variable and the precondition can be expressed in terms of the state of that variable.

For instance, it is more difficult to allow the removal in the middle of the list, because we would have to set `previous->next` to `current->next` with the

precondition that neither variable was concurrently written. Requires a more powerful atomic primitive, like CAS2 (compare and swap for two words).

Here is a general (but unfortunately also more expensive) approach:

Maintain a pointer to the "master copy" of the shared lock-free data structure. Readers require no synchronization. To modify the data structure, writers must

1. safe the current version of the master pointer
2. copy the shared data structure to a scratch location
3. modify the scratch copy
4. *atomically:*
   - verify that the master pointer has not been concurrently written
   - write master pointer to refer to the scratch copy (which is now new master copy)
5. if verification fails (i.e., another thread interfered), start over at step 1.
6. deallocate the old master once we can be sure all concurrent readers or writers are done with it

Note: We need to make sure the master pointer was not written during steps 1–4. It is not sufficient to compare the values at step 1 and 4, because another thread could have changed the master pointer and then changed it back to the original value (this is called the ABA problem.) Using an LL/SC primitive will do the right thing, but a CAS primitive is not sufficient.

Works for arbitrarily complex data structure, but: copying cost may be a concern.

Read-copy update (RCU) is a related technique, where readers are allowed to access a snapshot of a shared data structure without aynchronization or copying. Writers need to lock and copy. Works for data structures that are mostly read, and where it is OK if readers may see stale data.

Lock-free synchronization can be very efficient, but is extremely difficult to get right.