Transactions

Transactions are another way of providing atomicity, and are significantly different from locks in their handling of concurrency. Recall that for atomicity, we need to ensure that two threads do not interleave accesses to shared data within a critical section. Locks ensure this by requiring the programmer to apriori announce her intention of accessing shared data, and the semantics of a lock prevent concurrent execution within a critical section. This is a *pessimistic* method of dealing with concurrency, because you always need to acquire the lock, irrespective of whether another thread actually tried to enter the critical section or not. This is analogous to saying that whenever you enter a room where you need privacy, you must always first lock it from inside, before you start using the room.

A more optimistic method may be to just enter the room (without locking). If another thread tries to enter the same room, it will likely notice that somebody else is already using the room and will rollback its execution (or return from the room in the physical analogy). This is more optimistic because you assume that in the common case, it is unlikely that somebody will try to enter the room at the same time. Hence, you have avoided the overhead of having to acquire a lock. In software, this is done by enclosing the act of accessing the shared resource in a *transaction*. All accesses to the shared resource within a transaction are *tentative*. If during the accesses, the thread notices that the same resource is being concurrently accessed by another thread (thus violating atomicity), the transaction is *rolled back*. Let's look at some examples.

Bank account example with transactions

Recall the bank account example where the `transfer()` function was used to transfer a unit of money from one account to another. Let's recall the code which used fine-grained locking.

```
void transfer(account *a, account *b) {
  if (a->account_id < b->account_id) {
    acquire(&a->mutex);
    acquire(&b->mutex);
  } else {
    acquire(&b->mutex);
    acquire(&a->mutex);
  }
  if (a->money > 0) {
    a->money--;
    b->money++;
  }
  release(&a->mutex);
  release(&b->mutex);
}
```

In this code which uses fine-grained locks, whenever we access a shared account, we always acquire the corresponding lock (mutex) before the access. However, in the common case, it is unlikely (though possible) that another thread will be accessing the same accounts at the same time. But we pay the overhead of lock acquisition each time!

It would be nice if we could structure our code as a transaction that can be rolled back if required, as follows:

```
void transfer(account *a, account *b) {
  int am, bm;

  do {
    tx_begin();
    am = tx_read(a->money);
    bm = tx_read(b->money);
    if (am > 0) {
      am--;
      bm++;
    }
    success = tx_commit(am, &a->money, bm, &b->money);
  } while (!success);
}
```

Here `tx_begin` indicates the start of a transaction, and `tx_commit` indicates the end of a transaction. At the end, the commit operation *atomically* tries to update the values of shared variables `a->money` (with value stored in local variable `am`), and `b->money` (with value stored in local variable `bm`). The commit succeeds, if no concurrent transaction had read these shared variables during the time the transaction was executing. It fails otherwise.

Notice that to implement an atomic commit (and rollback if needed), the runtime system needs to track, which locations were read and written. Also notice that this mechanism of a transaction is only useful if we expect that the probability of a commit failure is small. Because if there are likely to be a lot of rollbacks, then the performance of this system may actually be worse than that of coarse-grained locks.

The probability of a commit failure depends on the size of the transaction and on the expected concurrency of this code region.

Transactions are very useful for databases, where the read and commit operations are performed on disk blocks. Because disk accesses are anyways slow, it is relatively cheap to maintain this information about which disk blocks were read or written (in memory). However, in the case of operating systems, maintaining such information for memory bytes is much more expensive. Recently, new processors (e.g., Intel's Haswell) are providing hardware support for implementing transactions for memory accesses (also called *transactional memory*).

Compare and Swap (List insert example, hits example)

While most code today uses locks for atomicity, there is a special class of operations that have been using transactions for a long time. These are operations that can be written as atomic updates on a single memory location. The common primitive for single memory location transactions is a *compare-and-swap* instruction.

```
int cmpxchg(int *addr, int old, int new) {
  int was = *addr;
  if (was == old) {
    *addr = new;
  }
  return was;
}
```

In the instruction form, this can be written as:

```
cmpxchg Rold, Rnew, Mem
```

This instruction compares the value at location `Mem` with register `Rold`; if the values are equal, it replaces the value at `Mem` with `Rnew`. Else it does nothing. It returns the old value at location `Mem` in register `Rold`. This instruction is atomic if prefixed with the `lock` prefix..

For example, if we want to increment a shared variable `hits` atomically using `cmpxchg`, we can write the code as:

```
int l_hits, l_new_hits;
retry:
l_hits = hits;
l_new_hits = l_hits + 1;
if (cmpxchg(&hits, l_hits, new_l_hits) != l_hits) {
  goto retry;
}
```

Here, we first read the shared `hits` variable into a local variable `l_hits`. We perform some computation on `l_hits` to obtain a new value `l_new_hits`. Now, we wish to write the new value `l_new_hits` to the shared variable `hits`, but only if the `hits` variable has not been modified in between (i.e., between the first read to `hits` and this point). To do this, we can use the `cmpxchg` instruction on the memory address of shared variable `hits` with the old value as `l_hits` and the new value as `l_new_hits`. If the atomic `cmpxchg` instruction returns that the current read value of `hits` is still equal to its previous read value (`l_hits`), the increment operation was performed atomically. On the other hand, if the current read value of `hits` differs from the previous read value, it indicates a concurrent access to the `hits` variable, and so the operation needs to be retried (or rolled-back).

Simlarly, our list insert example can be written using compare-and-swap (or *CAS*) as follows:

```
void insert(struct list *l, int data) {
  struct list_elem *e;
  e = new list_elem;
  e->data = data;
retry:
  e->next = l->head;
  if (cmpxchg(&l->head, e->next, e) != e->next) {
    goto retry;
  }
}
```

Once again, if two threads try to insert into a list concurrently, one of them will execute the atomic `cmpxchg` instruction first. Whichever thread executes `cmpxchg` first, will succeed (as it will see the same current value of `l->head` as the one that it previously read). The successful thread will also atomically update the value of `l->head` to its new value (with the inserted element). The thread which executes `cmpxchg` second, will notice that the value of `l->head` has changed from its last read value, and this will cause a rollback.

Reader-Writer Locks

Finally, often there is a situation where some parts of the code *only* read shared data, while other parts of the code read and write to it. The threads that only read the data do not need to be serialized with respect to each other. In other words, it is okay to allow multiple reader threads to execute concurrently. But it is still incorrect to allow a writer thread to execute concurrently with a reader thread. It is also incorrect to allow a writer thread to execute concurrently with another writer thread.

To capture this programming pattern, an abstraction called *read-write locks* can be used. Read-write locks are defined as follows:

```
struct rwlock rwlock;

void read_acquire(struct rwlock *);
void write_acquire(struct rwlock *);
void release(struct rwlock *);
```

The `read_acquire` function acquires the `rwlock` in read mode, while the `write_acquire` function acquires the `rwlock` in read/write mode. The semantics are that an `rwlock` can be acquired in read mode multiple times simultaneously. However, it can only be acquired in write mode, if

it is not currently held in either read or write mode by any other thread.