Locking

Locking Discipline

Consider the following code where the system maintains multiple accounts (e.g., bank accounts) and provides a function to transfer one unit of money from one account to another.

```
#define N 100
struct account {
  int account_id;
  int money;
};

struct account accounts[N];

bool transfer(struct account *src, struct account *dst)
{
  if (src->money > 0) {     // A
    src->money--;           // B
    dst->money++;           // C
    return true;
  }
  return false;
}
```

This code is correct if run in a sequential manner, but can result in many problems if multiple threads could execute it simultaneously. Here are two of the many problems that could occur:

- If multiple threads execute `transfer()` on the same `dst` account, there is a race condition at statement C, where an update can be lost (similar to how it was lost in the `hits` example discussed previously).
- If multiple threads execute `transfer()` on the same `src` account, there is a race condition at statements A and B. For example, it is now possible for an account to have negative money: consider the case where the `src` account initially has one unit of money. The first thread checks if `src` has any money; simultaneously, the second thread checks if `src` has any money; both succeed, and both decrement the money eventually leading resulting in the final value of -1 for `src->money`.

We need to use locks to fix this code. One option is to use a global lock for all accounts (*coarse-grained* locking) in the following way:

```
#define N 100
struct account {
  int account_id;
  int money;
};

struct account accounts[N];
struct lock global_lock;

bool transfer(struct account *src, struct account *dst)
{
  acquire(&global_lock);
  if (src->money > 0) {     // A
    src->money--;           // B
    dst->money++;           // C
    release(&global_lock);
    return true;
  }
  release(&global_lock);
  return false;
}
```

This code is correct. However, it is not necessarily efficient, as it serializes *all* calls to the `transfer()` function. For example, if two threads are operating on separate accounts (one on accounts A and B, and another on accounts C and D), then they do not need to be serialized.

A better option may be to use *finer-grained* locks. In doing so, the programmer must ensure that whenever an account's data is touched, the corresponding lock is held. This will allow much better concurrency and throughput in the system. However, as we will see next, fine-grained locking can be tricky.

We first discuss some incorrect ways of using per-account locks, before converging on the correct implementation. In our first try, the programmer tries to use separate locks for `src` operations and `dst` operations, namely `global_lock1` and `global_lock2`.

```
//ATTEMPT 1
#define N 100
struct account {
  int account_id;
  int money;
};
```

```
struct account accounts[N];
struct lock global_lock1, global_lock2;

bool transfer(struct account *src, struct account *dst)
{
  acquire(&global_lock1);
  if (src->money > 0) {     // A
    src->money--;           // B
    release(&global_lock1);
    acquire(&global_lock2);
    dst->money++;           // C
    release(&global_lock2);
    return true;
  }
  release(&global_lock1);
  return false;
}
```

Here, `global_lock1` is intended to protect operations on `src` account, and `global_lock2` is intended to protect operations on `dst` account. However, this code is hopelessly incorrect. Consider the case, when one thread wants to transfer money from account 0 to account 1, and the second thread wants to transfer money from account 1 to account 0.

```
void thread1(void) {
   ...
   transfer(&accounts[0], &accounts[1]);
   ...
}

void thread2(void) {
   ...
   transfer(&accounts[1], &accounts[0]);
   ...
}
```

For `thread1`, `src` refers to `accounts[0]` and `dst` refers to `accounts[1]`. For `thread2`, it is vice-versa. In this case, if the threads run concurrently, both threads could be accessing the same account (e.g., account 0) at the same time, resulting in a race condition. For example, thread1 could be at statement B while thread 2 could simultaneously be at statement C, both operating on the same account (account 0), which will result in potentially incorrect behaviour (e.g., lost update).

This problem occurred because we associated locks with function arguments, but arguments can be aliased to different accounts at the same time. A better strategy will be to associate locks with the accounts themselves, by using per-account locks. Here is another attempt at writing correct code.

```
//ATTEMPT 2
#define N 100
struct account {
  int account_id;
  int money;
  struct lock lock;
};

struct account accounts[N];

bool transfer(struct account *src, struct account *dst)
{
  acquire(&src->lock);
  if (src->money > 0) {     // A
    src->money--;           // B
    release(&src->lock);
                            // B1
    acquire(&dst->lock);
    dst->money++;           // C
    release(&dst->lock);
    return true;
  }
  release(&src->lock);
  return false;
}
```

This code is almost correct, but it has a problem. At statement B1, no locks are held, but the total amount of money in the system appears to be less than its actual value (by one). In other words, this transfer operation is *not* atomic. A race-free implementation of `transfer()` would look like the following:

```
bool transfer(struct account *src, struct account *dst)
{
  acquire(&src->lock);
  acquire(&dst->lock);
  if (src->money > 0) {     // A
```

```
    src->money--;           // B
    dst->money++;           // C
    release(&src->lock);
    release(&dst->lock);
    return true;
  }
  release(&src->lock);
  release(&dst->lock);
  return false;
}
```

Consider another thread running the `sum()` function below:

```
int sum(void)
{
  int i, ret;
  ret = 0
  for (i = 0; i < N; i++) {
    ret += accounts[i].money;
  }
  return ret;
}
```

Clearly, this function is also accessing shared data (`accounts`), and should thus use locks to ensure correctness. One way to ensure correctness is to take the locks for all accounts, before proceeding with the computation of sum, like this:

```
int sum(void)
{
  int i, ret;
  ret = 0

  for (i = 0; i < N; i++) {
    acquire(&accounts[i].lock);
  }
  for (i = 0; i < N; i++) {
    ret += accounts[i].money;
  }
  for (i = 0; i < N; i++) {
    release(&accounts[i].lock);
  }
  return ret;
}
```

This will ensure that the computation of `sum` is atomic with respect to other operations (e.g., `transfer`) in the system. In general, the following locking discipline ensures race-freedom:

- Associate a lock with every shared region (granularity is determined by the choice of shared regions and locks).
- Ensure that before accessing a shared region, the corresponding associated lock is held.
- If an atomic operation involves accessing multiple shared regions, acquire the locks of all the shared regions apriori. Release all these locks after the operation is finished.

In our examples on `transfer()` and `sum()`, this discipline ensured race-freedom.

However, there is a second big problem with locks, namely *deadlocks*. In the previous example, consider the situation where `thread1` attempts to transfer money from account0 to account1, while `thread2` attempts to transfer money from account1 to account0. Hence, `thread1` will first acquire account0's lock, and then account1's lock. Similarly, `thread2` will first acquire account1's lock, and then acquire account0's lock. In a bad schedule, it can so happen that before thread1 acquires account1's lock (after acquiring account0's lock), thread2 acquires account1's lock. Thus, thread1 will have to wait for thread2 to release account1's lock before it proceeds further. However thread2 will next attempt to acquire account0's lock and will have to wait for thread1 to release it.

At this stage, both thread1 and thread2 will be waiting for each other to release a lock that they need. And so they will keep waiting forever (as none of them will be able to proceed to the release statement). This is an example of a deadlock.

In an operating system the usual solution is to avoid deadlocks by establishing an order on all locks and making sure that every thread acquires its locks in that order. In this example, the rule may be to lock the account with the lowest account ID first. Picking and obeying the order on *all* locks requires that modules make public their locking behavior, and requires them to know about other module's locking. This can be painful and error-prone. Here is the correct code which is both race-free and deadlock-free.

```
bool transfer(struct account *src, struct account *dst)
{
  if (src->account_id < dst->account_id) {
    acquire(&src->lock);
    acquire(&dst->lock);
  } else {
    acquire(&dst->lock);  //reverse the order
    acquire(&src->lock);
  }
```

```
  if (src->money > 0) {     // A
    src->money--;           // B
    dst->money++;           // C
    release(&src->lock);
    release(&dst->lock);
    return true;
  }
  release(&src->lock);
  release(&dst->lock);
  return false;
}

int sum(void)
{
  int i, ret;
  int s[N];
  ret = 0

  //sort the accounts and store the sorted order in s[]
  s = sort_by_account_id(accounts);

  for (i = 0; i < N; i++) {
    acquire(&accounts[s[i]].lock);
  }
  for (i = 0; i < N; i++) {
    ret += accounts[i].money;
  }
  for (i = 0; i < N; i++) {
    release(&accounts[s[i]].lock);
  }
  return ret;
}
```