Advanced Concurrency/Synchronization

Andrew Quinn

Learning Objectives:

1. How do we make synchronization *scale*?
2. What tradeoffs do we face when performing these optimizations?

Announcements:

1. HW1 due tonight!

ACCELERATING TOO MUCH MILK

We concluded Too Much Milk with the following implementation:

```
acquire(lock);
if (no_milk) {
  buy_milk();
}
release(lock);
```

In class, we mentioned that this approach has a major performance issue. You and your housemate will hold the lock while buying milk, which ensures safety. However, buying milk is a slow operation, so whoever is not buying milk will spend a lot of time waiting. We call this *lock contention*: it refers to the scenario when one process/thread waits while attempting to acquire a lock currently held by another process/thread.

In general, it is a good idea to limit lock contention to improve performance. For too much milk, we want to do this by not holding the lock while calling `buy_milk()`. We can do this by revisiting the "note" idea from last class—use a note to indicate whether you went to the store, but use a lock to provide critical sections around writing the note. Here's a possible solution:

```
to_buy = False;
acquire(lock);
if (no_note && no_milk) {
  to_buy = True;
  leave_note();
}
release(lock);
if (to_buy)
  buy_milk();

acquire(lock);
if (to_buy)
  remove_note();
release(lock);
```

Since checking and leaving a note is much faster than buying milk, this solution will drastically reduce lock contention.

LOCKING GRANULARITY

To introduce the next section, let's look at the concurrent linked list described in your book:

```c
typedef struct __node_t {
  int key;
  struct __node_t *next;
} node_t;

typedef struct __list_t {
  node_t *head;
  lock l;
} list_t;

void List_Init(list_t *L) {
  L->head = NULL;
}

bool List_Insert(list_t *L, int key) {
  node_t *new = malloc(sizeof(node_t));
  if (new == NULL) {
    return false;
  }
  new->key = key;

  acquire(L->l);
  new->next = L->head;
  L->head = new;
  release(L->l);

  return true;
}

bool List_Lookup(list_t *L, int key) {
  bool found = False;

  acquire(L->l);
  node_t *curr = L->head;
  while (curr) {
    if (curr->key == key) {
      rv = True
      break;
    }
    curr = curr->next;
  }
  release(L->l);

  return found;
}
```

This solution provides a safe concurrent linked-list, but it will have a lot of lock contention: there can only be at most one thread looking up or inserting items in the list at a time. The reason for this limitation is that this approach uses *coarse-grained locking*—it uses a single lock for the entire list. We could reduce lock contention by implementing *fine-grained locking*, where we use a lock per node. We could update the node struct definitions to:

```
typedef struct __node_t {
  int key;
  struct __node_t *next;
  lock l;
} node_t;
```

Can we change the insert code to the following:

```
bool List_Insert(list_t *L, int key) {
  node_t *new = malloc(sizeof(node_t));
  if (new == NULL) {
    return false;
  }

  acquire(L->head->l);
  new->next = L->head;
  L->head = new;
  release(new->next->l);

  return true;
}
```

Nope, although the reason is quite subtle. The problem is that updating the head of the list means that the lock that is intended to protect the head of the list changes during the critical section. The change means that any thread waiting to acquire the head's lock may have actually hold an intermediate node's lock by the time acquire actually returns!

OK, so we won't be able to reduce lock contention on insert, at least not for a singly-linked list. But, we should be able to use our per-node lock to reduce lock contention when doing a lookup operation. To make this work we have to use a technique called *hand-over-hand* locking [1]. The idea is that, while iterating through a list, you acquire the next lock *before* releasing the current one. This ensures that the code does not have any violations related to the `next` pointers:

[1] We actually do not need this technique since we do not have a delete implementation. But, its a good example of the technique nonetheless

```
bool List_lookup(list_t *L, int key) {
  bool found = False;

  acquire(L->l);

  node_t *curr = L->head;
  lock *prevl = L->l;

  while (curr) {

    // hand-over-hand locking here:
    acquire(curr->l);
    release(*prevl);
    prevl = &curr->l;

    if (curr->key == key) {
      found = True
      break;
    }
    curr = curr->next;
  }

  release(prevl);
  return found;
}
```

Would this design work for a doubly-linked list with a reverse iterator? Why or why not?

CONCLUSION

Before we wrap up, we should point out the tradeoffs here: there is significantly less lock contention when performing a lookup. But, how many more operations need to occur per lookup? You have an addition `N` calls to both `acquire` and `release`! Is the additional *scalability* worth all of that extra overhead?

For a linked list, almost certainly not. The list needs to be huge and you need to have an extreme number of threads. Even then, it is rarely worth it.