Concurrency/Synchronization Review

Andrew Quinn

Learning Objectives:

1. Review: Why do we need synchronization?
2. Review: What synchronization primitives can you use?
3. Tips for using synchronization primitives in your code.

Announcements:

1. HW1 extended to Wednesday!

Why do we need synchronization?

Recall that the key role of an operating system is to provide multi-programming: having multiple programs that seem to operate *at the same time* over a shared system. Unfortunately, bad things can happen if we don't share system resources in a safe way. For example, the 2003 Blackout in the Norther-eastern United States and the radiation overdoses from the Therac-25 medical device were both caused by a particular class of issues, data-races, that lead to issues in the behavior of software programs. How can we share resources without bad things happening?

example: too much milk. To illustrate this issue, we will use a classic problem in computer systems, "too much milk". Imagine that you live in an apartment with a housemate. The two of you share a fridge and want to ensure that there is exactly one gallon of milk in your fridge at all times. Imagine that you and your housemate both execute the following algorithm, can anything bad happen?

```
if (no_milk())
  buy_milk();
```

The answer is yes! Suppose that you observe that you have no milk so you buy some. While you're buying milk, suppose that your housemate also notices that there is no milk and goes to buy some. You'll wind up with *too much milk*!

We need to ensure that our solution has a property called *mutual exclusion*—in this case, we need to ensure that at most one person is buying milk at a given time. To make this happen, we need to make checking for milk and buying milk a *critical section*—we need to make it an atomic compound operation.

Before we talk about how you can use synchronization primitives to solve this, let's stay in the physical world. A common way you might solve this issue with your housemate is by using notes:

```
if (no_note) {
  leave_note();
  if (no_milk) {
    buy_milk();
  }
  take_note();
}
```

Does this work? Not quite, we can still have too much milk. The issue is that checking for a note and leaving a note is not an atomic operation, so you have basically the same problem that you had before. One way you could try to do better is to change the order of leaving notes and checking notes (and, have separate notes for separate people):

```
leave_note(me);
if (no_note(housemate)) {
  if (no_milk) {
    buy_milk();
  }
  take_note(me);
}
```

Ah, you say, but this also does not work. We can wind up with a situation in which we have no milk! If we both leave a note at the same time, then neither of us will even check to see if we have milk.

OK, so you might be thinking, *is this even possible*? And, of course the answer is yes. One solution, which does not require any special primitives, is the following. You elect one person (and only one) to wait around when the other leaves a note. This makes it asynchronous (i.e., you and your housemate do not execute the same code). You would execute:

```
leave_note(me);
while (no_note(housemate)) {}
if (no_milk) {
  buy_milk();
}
take_note(me);
```

while your housemate would execute:

```
leave_note(housemate);
if (no_note(you)) {
  if (no_milk) {
    buy_milk();
  }
  take_note(housemate);
}
```

There are some bad things about this solution. First, it is surprisingly difficult to reason about, even for a small algorithm. Second, you have this busy-waiting operation, in which you spend a lot of time waiting. Third, the asymmetry is inelegant, and, adds to the difficult reasoning. And, finally, it is not clear how you might scale this to three people. We call these types of solutions "ad hoc" synchronization; it is rarely a good idea to use ad hoc synchronization in your programs.

SYNCHRONIZATION PRIMITIVES

We have synchronization primitives top help us. They are provided by operating systems and libraries for user applications to use; they are also provided as modules in most operating systems for use by the various tasks that an operating system must perform.

LOCKS

The first primitive that we'll discuss is a lock. A lock ensures that there is at most one *holder* of it—so it provides the mutual exclusion property that we discussed. Locks provide two functions, `acquire` (sometimes called `lock`) and `release` (sometimes called `unlock`). You can easily solve too much milk with locks:

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

This is nice, simple, and correct. But, buying milk might take a long time, and your housemate (or you) have to sit around and wait to acquire the lock while the other one actually goes to the store. Can you think of a solution that would prevent these long waits? We'll discuss more about this later in the week.

CONDITION VARIABLES

In addition to enforcing mutual exclusion, we also sometimes need to ensure certain *ordering* conditions—i.e., that a specific event happens-before another event. This is where condition variables come into play. A condition variable provides three functions: `wait`, which forces

the calling thread to wait until it is woken up, `signal`, which wakes up a thread waiting on the condition variable, and `broadcast`, which wakes up all of the threads waiting on the condition variable. Condition variables are linked to a lock—`wait` takes a lock parameter; it releases before waiting and reacquires upon wakeup.

```
char buffer[SIZE];
int count = 0, in = 0, out = 0;
lock l;
cv pushcv, popcv

void push(char c) {
    acquire(l);
    while (count == SIZE)
        wait(pushcv, l);

    count++;
    buffer[in] = c;
    in = (in + 1) % SIZE;
    signal(popcv, l);
    release(l);
}

char pop() {
    char c;
    acquire(l);
    while (count == 0)
        wait(popcv, l);

    count--;
    c = buffer[out];
    out = (out + 1) % SIZE;
    signal(pushcv, l);
    release(l);
    return c;
}
```

SEMAPHORES

The final synchronization primitive that you will encounter are semaphores. Each semaphore has an associated non-negative integer and supports two operations: `down`, which waits until the semaphore's value is positive, then decrements the value and returns, and `up`, which increments the semaphore's value. You probably will not use semaphores much in this class, so we won't elaborate on them more during this lecture.

SYNCHRONIZATION GUIDELINES

We'll finish today's lesson with some helpful guidelines to follow when using locks. This list is adapted from a list put together at the University of Michigan, which was itself adapted from a list from people at Cornell.

1. Name your synch variables properly.
2. Use Locks and CVs instead of a semaphore whenever possible.
3. Stick with either semaphores or CVs: don't mix the two.
4. Do not busy wait.
5. Protect all shared state.
6. Acquire (or grab) the lock upon entry to procedures; release the lock at the end of the procedure.
7. Guard wait predicates with a while loop