Topic 5: Synchronization: The Too-Much-Milk Problem

Reading: Section 5.1

Next reading: 5.1-5.5

- The "too much milk" problem.

	Roommate A	Roommate B
3:00	Look in fridge. Out of milk.	
3:05	Leave for store.	
3:10	Arrive at store.	Look in fridge. Out of milk.
3:15	Leave store.	Leave for store.
3:20	Arrive home, put milk away.	Arrive at store.
3:25		Leave store.
3:30		Arrive home. OH, NO!

- What does correct mean? Somebody gets milk, but we don't get too much milk.
- *The most important thing in synchronization is to figure out* ***precisely*** *what you want to achieve*.
    - **Safety**: the program never enters a bad state.
    - **Liveness**: the program eventually enters a good state. Opposite of liveness is *starvation*.
    - E.g., what are the constraints on a traffic light?

- Roommates as a program:

```
1. if (Milk == 0) {
2.      Milk++;
3. }
```

- *Synchronization*: using atomic operations to ensure correct cooperation (safety) between processes (eliminate *race conditions*).
    - E.g. buyer and drinker of milk; cars going through an intersection.
    - Does not rely on scheduling priorities.
    - Also known as *inter-process communication (IPC)*.
    - *Atomic operation* -- an operation that is indivisible by other operations.
        - Alternatively, an operation whose effects can be observed either before or after the operation, but not during.
- *Mutual exclusion*: Mechanisms that ensure that only one person or process is doing certain things at one time (others are excluded). E.g. only one person goes shopping at a time.
- *Critical section*: Code in which only one process may be executing simultaneously. E.g. shopping, intersection.
    - Note: critical sections are independent from functions -- a single function may have several critical sections, or a critical

section may span several functions. It is often desirable to have them map one-to-one to make the program more understandable, but it is not required.

- *Mutual exclusion* is the mechanism, a *critical section* is the result.
- There are many ways to achieve mutual exclusion.
    - In kernel mode, you can disable interrupts (uniprocessor only) and not invoke the dispatcher.
    - In user mode, most solutions involve some sort of *locking* mechanism: prevent a process from doing something. If a process tries to acquire a lock that is already held it must either wait, or do something else that doesn't require the lock.
- Three elements of locking:
    1. Lock before using. (leave note)
    2. Unlock when done. (remove note)
    3. Wait (or skip) if locked. (don't shop if note)
- First attempt at computerized milk buying:

Processes A & B

```
1. if (Note == FALSE) {
2.      if (Milk == 0) {
3.           Note = TRUE;
4.          Milk++;
5.          Note = FALSE;
6.     }
7. }
```

- This doesn't always work: A1 A2 B1 B2 B3 A3 B4 A4...
- This solution works for people because lines 1 - 3 are performed atomically: you'll see the other person at the refrigerator and make arrangements. Typically, computers can't both test (look for note) and set (leave note) at the same time.
    - Looking for the note requires a memory read, and leaving a note requires a memory write, which are separate instructions.
    - In this case, we haven't eliminated the problem; we've just moved it and made it a little less likely (i.e. a little more insidious). This is typical of first attempts at solutions to synchronization problems.
    - What happens if we leave the note before looking for a note: does this make everything work? No, no-one will buy milk.
    - What if we check for milk after leaving the note? Still incorrect (A1 B1 A2 B2). In this case, we haven't eliminated