Today:

- Discuss Mutex, Locks
- Work Up to Concept of Threads

### 7.1 Synchronization

Threads must communicate to ensure consistency. Safe communication will require that we communicate consistent values. For example, two people trying to write to a buffer may wind up with interleaved data:

```
Processor_1  Processor_2  Processor_3
write:  x=2  x=4  read x
        y=3  y=5  read y
```

It is legal for Processor_3 to see certain combinations of values for x and y. Some architectures may read the x first or the y first. Our goal is atomicity: if Processor_1 is going to act, it should write values for x and for y (and similarly for Processor_2).

"Too Much Milk Problem"-Every time you see no milk in the fridge, you go to the grocery store to buy it–while you are at the store, your roommate notices there is no milk and goes to the store to buy it...

Our dilemma is that for most simple actions we could want a process to do, we can invariably break them down into smaller parts that should not be allowed to overlap with parts from another process.

```
\[
P_1 \quad P_2
\]

\[
\begin{array}{c}
milk++ & milk++ \\
\downarrow & \downarrow \\
v=milk & v=milk \\
v++ & v++ \\
milk=v & milk=v \\
\end{array}
\]
```

Our solution is mutual exclusion ("mutex"), which will act as a gate that only lets one process in (whoever gets there first).

**Lock:** mechanism for mutual exclusion
- lock on entering critical section, accessing shared data
- unlock when complete
- wait if locked
Milk Problem

Correctness properties

Safety—“nothing bad happens”—only one buys milk
Progress—“something good eventually happens”—someone buys milk if needed

First idea: use atomic loads and stores as building blocks
leave a note, remove when done, wait if there is note
problem: might see no milk and no note at same time

Second idea: use labeled notes
problem: both could leave notes, see each other’s note, and neither would get the milk

Third idea: one waits while the other has a note out
this works, but we must try all possibilities to verify
problems:

• Complicated (hard to prove correctness) — \(2^N\) tests for \(N\) lines of 2 processes

• Asymmetrical — different code for processes A and B (what happens when we add even more processes to the mix?)

• Poor utilization — waiting consumes CPU, no useful work

• Possibly non-portable

Language Support
Synchronization complicated
Better – provide language-level support

Locks
Provide mutual exclusion to shared data via 2 atomic routines
Lock: acquire – wait for lock, take
Lock: release – unlock, wake up waiters (who are asleep)

Thread A (same as B)
Lock.acquire();
if(no milk)
buy milk;
Lock.release();

Implementing Locks
-requires hardware support
-can build on atomic operations
Load/Store
Disable Interrupts - uniprocessors only
Test/Set, Compare, Swap

Dekker’s Algorithm (for more on this, visit your local library!)
Disabling Interrupts

- Prevent scheduler from switching processes in the middle of critical sections
  - ignores quantum expiration (timer interrupt)
  - no handling of I/O
- only useful in kernel

Locks are horrible and buggy. You can encounter problems such as the double locking bug (where a process could lock then accidentally try to lock again, notice the lock it just put on, then go to sleep).