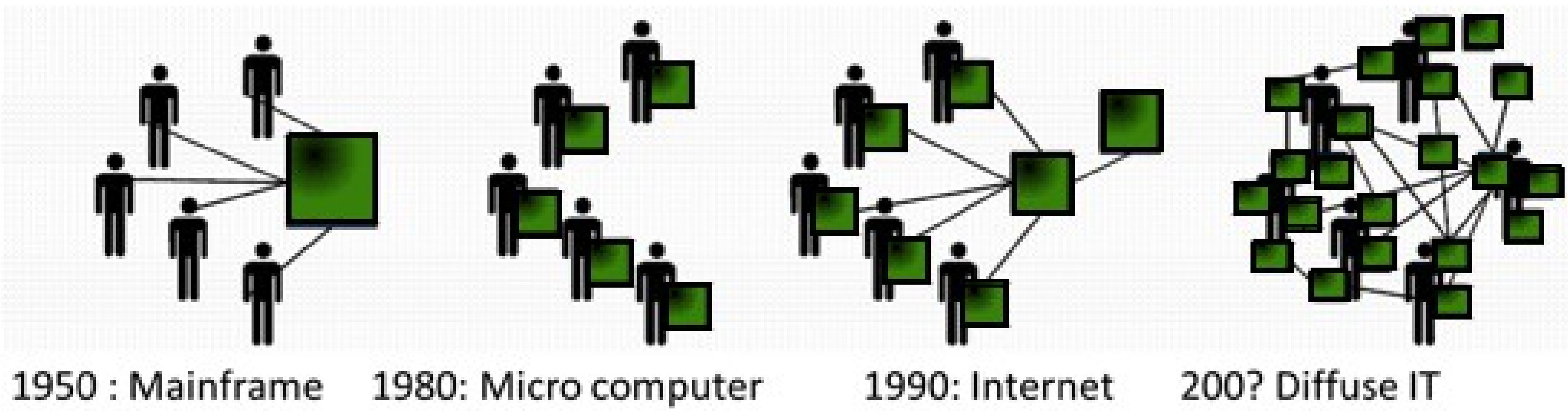
Disciplina
Sistemas de Computação

Aula 12



Motivation: too much milk problem

Motivation: too much milk problem

- People need to coordenate
  - You love milk, but don't want too much

Time	Person A	Person 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	Buy milk	Leave for store
3:20	Arrive home, put milk away	Arrive at store
3:25		Buy milk
3:30		Arrive home, put milk away

Motivation: too much milk problem

Synchronization Terminology

- **Synchronization:** use of atomic operations to ensure cooperation between threads
- **Atomic Operation:** an operation that always runs to completion or not at all!
  - It is indivisible: it cannot be stopped in the middle and state cannot be modified by someone else in the middle!
  - Fundamental building block – if no atomic operations, then have no way for threads to work together!
- On most machines, memory references and assignments (i.e. loads and stores) of words are atomic!

Synchronization Terminology

- **Mutual Exclusion:** ensure that only one thread does a particular activity at a time and excludes other threads from doing it at that time
- **Critical Section:** piece of code that only one thread can execute at a time
- **Lock:** mechanism to prevent another process from doing something
  - Lock before entering a critical section, or before accessing shared data.
  - Unlock when leaving a critical section or when access to shared data is complete
  - Wait if locked

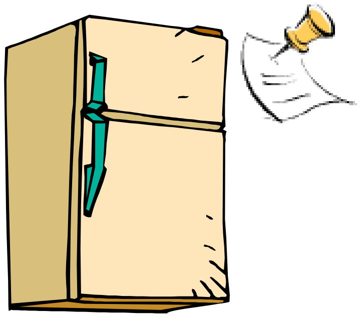
=> All synchronization involves waiting.

Too Much Milk Problem: conditions and correctness properties

- What are the wished conditions we want to achieve?
  - Neither ”too much milk” nor ”no milk”
- What are the correctness properties for this problem?
  - Only one person buys milk at a time.
  - Someone buys milk if you need it.

Too Much Milk Problem: Possible solution?

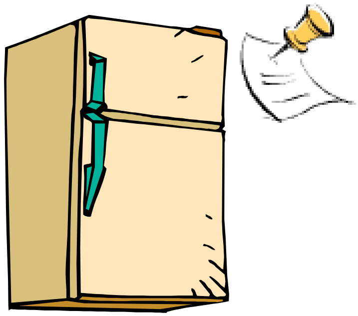
- Restrict ourselves to atomic loads and stores as building blocks.
  - Leave a note (a version of lock)
  - Remove note (a version of unlock)
  - Do not buy any milk if there is note (wait)



Too Much Milk: Solution 1

Thread A

```
if (noMilk & NoNote) {
    leave Note;
    buy milk;
    remove note;
}
```



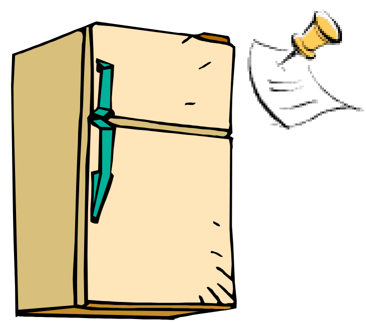
Thread B

```
if (noMilk & NoNote) {
    leave Note;
    buy milk;
    remove note;
}
```

- Does this work?

Too Much Milk: Solution 2

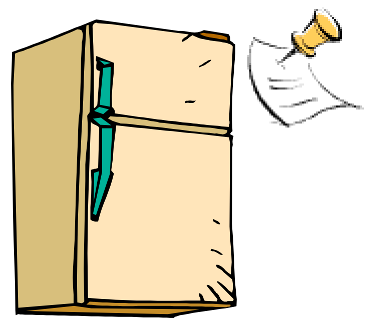
- How about using labeled notes so we can leave a note before checking the milk?



Thread A	Thread B
leave note A	leave note B
if (noNote B) {	if (noNote A) {
if (noMilk){	if (noMilk){
buy milk;	buy milk;
}	}
}	}
remove note;	remove note;

  - Does this work?

Too Much Milk: Solution 3



Thread A

```
    leave note A
X: while (Note B) {
        do nothing;
    }
    if (noMilk){
        buy milk;
    }
    remove note A;
```

Thread B

```
    leave note B
Y: if (noNote A) {
        if (noMilk){
            buy milk;
        }
    }
    remove note B;
```

- Does this work?

Correctness of Solution 3

- At point Y, either there is a note A or not.

  1. If there is no note A, it is safe for thread B to check and buy milk, if needed. (Thread A has not started yet).

  2. If there is a note A, then thread A is checking and buying milk as needed or is waiting for B to quit, so B quits by removing note B.

- At point X, either there is a note B or not.

  1. If there is not a note B, it is safe for A to buy since B has either not started or quit.

  2. If there is a note B, A waits until there is no longer a note B, and either finds milk that B bought or buys it if needed.

- Thus, thread B buys milk (which thread A finds) or not, but either way it removes note B. Since thread A loops, it waits for B to buy milk or not, and then if B did not buy, it buys the milk.

Is Solution 3 a good solution?

- It is too complicated - it was hard to convince ourselves this solution works.
- It is asymmetrical - thread A and B are different. Thus, adding more threads would require different code for each new thread and modifications to existing threads.
- A is busy waiting - A is consuming CPU resources despite the fact that it is not doing any useful work.

=> This solution relies on loads and stores being atomic.

Language Support for Synchronization

- Have your programming language provide atomic routines for synchronization?

- Locks: one process holds a lock at a time, does its critical section releases lock.
- Semaphores: more general version of locks.
- Monitors: connects shared data to synchronization primitives.

=> All of these require some hardware support, and waiting.

Locks

- Locks: provide mutual exclusion to shared data with two “atomic” routines:
  - Lock.Acquire - wait until lock is free, then grab it.
  - Lock.Release - unlock, and wake up any thread waiting in Acquire.
- Rules for using a lock:
  - Always acquire the lock before accessing shared data.
  - Always release the lock after finishing with shared data.
  - Lock is initially free.

Implementing Too Much Milk with Locks

Thread A

```
Lock.Acquire();
if (noMilk){
    buy milk;
}
Lock.Release();
```

Thread B

```
Lock.Acquire();
if (noMilk){
        buy milk;
    }
Lock.Release();
```

- This solution is clean and symmetric.
- How do we make Lock.Acquire and Lock.Release atomic?

Hardware Support for Synchronization

- Implementing high level primitives requires low-level hardware support
- What we have and what we want

	Concurrent programs
Low-level atomic operations (hardware)	load/store interrupt disable test&set
High-level atomic operations (software)	lock semaphore monitors send & receive

Summary

- Communication among threads is typically done through shared variables.
- Critical sections identify pieces of code that cannot be executed in parallel by multiple threads, typically code that accesses and/or modifies the values of shared variables.
- Synchronization primitives are required to ensure that only one thread executes in a critical section at a time.
  - Achieving synchronization directly with loads and stores is tricky and error-prone
  - Solution: use high-level primitives such as locks, semaphores, monitors