## Thread Synchronization: Too Much Milk

Implementing Critical Sections in Software Hard

- The following example will demonstrate the difficulty of providing mutual exclusion with memory reads and writes
  - Hardware support is needed
- The code must work all of the time
  - Most concurrency bugs generate correct results for some interleavings
- Designing mutual exclusion in software shows you how to think about concurrent updates
  - Always look for what you are checking and what you are updating
  - A meddlesome thread can execute between the check and the update, the dreaded race condition

### Thread Coordination

<table>
<thead>
<tr>
<th>Jack</th>
<th>Jill</th>
</tr>
</thead>
<tbody>
<tr>
<td>Look in fridge; out of milk</td>
<td>Look in fridge; out of milk</td>
</tr>
<tr>
<td>Go to store</td>
<td>Go to store</td>
</tr>
<tr>
<td>Buy milk</td>
<td>Buy milk</td>
</tr>
<tr>
<td>Arrive home; put milk away</td>
<td>Arrive home; put milk away</td>
</tr>
<tr>
<td></td>
<td>Oh, no!</td>
</tr>
</tbody>
</table>

Fridge and milk are shared data structures

### Formalizing “Too Much Milk”

- Shared variables
  - “Look in the fridge for milk” – check a variable
  - “Put milk away” – update a variable
- Safety property
  - At most one person buys milk
- Liveness
  - Someone buys milk when needed
- How can we solve this problem?

### How to think about synchronization code

- Every thread has the same pattern
  - Entry section: code to attempt entry to critical section
  - Critical section: code that requires isolation (e.g., with mutual exclusion)
  - Exit section: cleanup code after execution of critical region
  - Non-critical section: everything else
- There can be multiple critical regions in a program
  - Only critical regions that access the same resource (e.g., data structure) need to synchronize with each other

```c
while(1) {
    Entry section
    Critical section
    Exit section
    Non-critical section
}
```

### The correctness conditions

- Safety
  - Only one thread in the critical region
- Liveness
  - Some thread that enters the entry section eventually enters the critical region
  - Even if some thread takes forever in non-critical region
- Bounded waiting
  - A thread that enters the entry section enters the critical section within some bounded number of operations.
- Failure atomicity
  - It is OK for a thread to die in the critical region
  - Many techniques do not provide failure atomicity

```c
while(1) {
    Entry section
    Critical section
    Exit section
    Non-critical section
}
```
Too Much Milk: Solution #0

```java
while(1) {
  if (noMilk) { // check milk (Entry section)
    if (noNote) { // check if roommate is getting milk
      leave Note; // Critical section
      buy milk;
      remove Note; // Exit section
    } // Non-critical region
  }
}
```

- Is this solution
  1. Correct
  2. Not safe
  3. Not live
  4. No bounded wait
  5. Not safe and not live

It works sometime and doesn't sometimes other times.
Threads can be context switched between checking and leaving note
Live, note left will be removed
Bounded wait ('buy milk' takes a finite number of steps)

What if we switched the order of checks?

Too Much Milk: Solution #1

```java
while(1) {
  while(turn ≠ Jack) ; //spin
  while (Milk) ; //spin
  buy milk;      // Critical section
  turn := Jill  // Exit section
  // Non-critical section
}
```

- Is this solution
  1. Correct
  2. Not safe
  3. Not live
  4. No bounded wait
  5. Not safe and not live

At least it is safe.

Too Much Milk: Solution #2 (a.k.a. Peterson's algorithm):
combine ideas of 0 and 1

Variables:
- \( i_j \): thread \( T_j \) is executing, or attempting to execute, in CS
- \( \text{turn} \): id of thread allowed to enter CS if multiple want to

Claim: We can achieve mutual exclusion if the following invariant holds before thread \( i \) enters the critical section:

\[
\neg (i_1 \land \neg \text{turn} = i) \lor \neg i_0 \\
(\neg i_0 \lor (i_0 \land \text{turn} = 1)) \land i_1 \\
(\neg i_1 \lor (i_1 \land \text{turn} = 0)) \land i_0 \\
((\text{turn} = 0) \lor (\text{turn} = 1)) = false
\]

Intuitively: \( j \) doesn't want to execute or it is \( i \)'s turn to execute

Peterson's Algorithm

```
Jack
while (1) { 
  in_0 := true; 
  turn := Jill; 
  while (turn == Jill && in_1) // wait 
  Critical section 
  in_0 := false; 
  Non-critical section 
}
```

```
Jill
while (1) { 
  in_1 := true; 
  turn := Jack; 
  while (turn == Jack && in_0) // wait 
  Critical section 
  in_1 := false; 
  Non-critical section 
}
```

Spin!

Turn=Jack, \( i_0 = false, i_1 = true \)

Safe, live, and bounded waiting
But, only 2 participants

Too Much Milk: Lessons

- Peterson's works, but it is really unsatisfactory
  1. Limited to two threads
  2. Solution is complicated; proving correctness is tricky even for the simple example
  3. While thread is waiting, it is consuming CPU time

- How can we do better?
  1. Use hardware to make synchronization faster
  2. Define higher-level programming abstractions to simplify concurrent programming