### Thread Synchronization: Too Much Mílk

### 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
  - > A meddlesome thread can execute between the check and the update, the dreaded race condition

### Thread Coordination

### Too much milk!

### Jack

- · Look in the fridge; out of milk
- Go to store
- Buy milk
- Arrive home; put milk away

### Jill

- Look in fridge; out of milk
- Go to store
- Buy milk
- Arrive home; put milk away
- Oh, no!

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
- - > 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

```
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
  - > 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 while(1) {

```
Entry section
Critical section
Exit section
Non-critical section
```

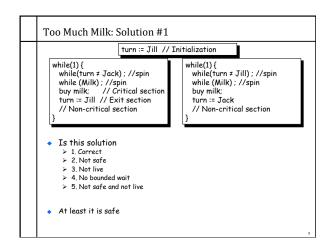
```
Too Much Milk: Solution #0
             while(1) {
  if (noMilk) {
   if (noNote) {
                                    // check milk (Entry section)
                                    // check if roommate is getting milk
                   leave Note; //Critical section
                   buy milk;
                   remove Note; // Exit section
               // Non-critical region

    Is this solution

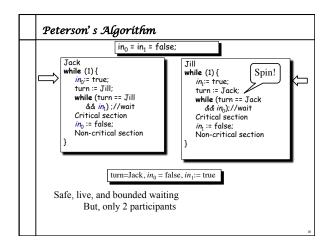
       1. Correct2. Not safe
       > 3. Not live
       > 4. No bounded wait
                                                               What if we switch the
       > 5. Not safe and not live
                                                               order of checks?

    It works sometime and doesn't some 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)
```



## Solution #2 (a.k.a. Peterson's algorithm): combine ideas of 0 and 1 Variables: > in; thread $T_i$ is executing, or attempting to execute, in CS > 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: $\{(\neg in_0 \lor (in_0 \land turn = 1)) \land in_i\}$ $\{(\neg in_1 \lor (in_1 \land turn = 0)) \land in_0\}$ $\Rightarrow$ $((turn = 0) \land (turn = 1)) = false$ Intuitively: j doesn't want to execute or it is i's turn to execute



## Too Much Milk: Lessons

- · Peterson's works, but it is really unsatisfactory
  - > Limited to two threads
  - > Solution is complicated; proving correctness is tricky even for the simple example
  - > While thread is waiting, it is consuming CPU time
- How can we do better?
  - ➤ Use hardware to make synchronization faster
  - Define higher-level programming abstractions to simplify concurrent programming

# Towards a solution The problem boils down to establishing the following right after entry; (¬in₁ ∨ (in₁ ∧ tum = i)) ∧ in₁ = (¬in₁ ∨ tum = i) ∧ in₁ Or, intuitively, right after Jack enters: Jack has signaled that he is in the entry section (in₁) - And Jill isn't in the critical section or entry section (¬in₁) - Or Jill is also in the entry section but it is Jack's turn (in₁ ∧ tum = i) How can we do that? Entry₁ = in₁ := true; while (in₁ ∧ turn ≠ i):

```
Thread T_0
while (Iterminate) {

Thread T_1
while (Iterminate) {

In:= true

In:= t
```

```
Safe?
\begin{bmatrix} \text{Thread } T_0 \\ \text{while (iterminate) { } \\ \textit{in_i} \equiv \text{ true: } \\ \alpha_0 \text{ turn} := 1; \\ \textit{(in_0)} \\ \text{while } (\textit{it_i}, \land turn \neq 0); \\ \textit{(in_0} \land (\neg in_1 \lor turn \neq 0) \lor \text{ at}(\alpha_1))\} \\ \textit{CS}_0 \\ \textit{in_0} := \text{false: } \\ \textit{NCS}_0 \\ \textit{\}} \end{bmatrix}
\text{If both in CS, then}
\textit{in}_0 \land (\neg in_1 \lor \text{ at}(\alpha_1) \lor turn = 0) \land \textit{in}_1 \land (\neg in_0 \lor \text{ at}(\alpha_0) \lor turn = 1) \land \land \neg \text{ at}(\alpha_0) \land \neg \text{ at}(\alpha_1) = (turn = 0) \land (turn = 1) = \text{false}}
```

```
 \begin{array}{|c|c|c|} \hline \textbf{Live?} \\ \hline \hline & \text{Thread } T_0 \\ \hline \textbf{while } & \text{(Iterminate) } \{ \\ & \{S_1; -in_0 \land (turn = 1 \lor turn = 0)\} \\ & in_0 : \text{ true:} \\ & \{S_2; in_0 \land (turn = 1 \lor turn = 0)\} \\ & \alpha_0 & turn := 1 : \\ & \{S_3\} \\ & \text{while } & (in_0 \land turn = 1 \lor turn = 0)\} \\ & \alpha_0 & turn := 1 : \\ & \{S_3\} \\ & \text{while } & (in_0 \land turn = 1 \lor turn = 0)\} \\ & \alpha_0 & turn := 1 : \\ & \{S_3\} \\ & \text{while } & (in_0 \land turn = 1 \lor turn = 0)\} \\ & \alpha_0 & turn := 0 : \\ & \{S_3; in_0 \land (in_0 \lor at(\alpha_1) \lor turn = 0)\} \\ & \alpha_1 & turn := 0 : \\ & \{S_3\} \\ & \text{while } & (in_0 \land turn = 1)\} \\ & (R_3) & \text{while } & (in_0 \land turn = 1)\} \\ & (S_3) & \text{while } & (in_0 \land turn = 1)\} \\ & (S_3) & \text{while } & (in_0 \land turn = 1)\} \\ & (S_3) & \text{while } & (in_0 \land turn = 1)\} \\ & (S_3) & \text{while } & (in_0 \land turn = 1)\} \\ & (S_3) & \text{while } & (in_0 \land turn = 1)\} \\ & (S_3) & \text{while } & (in_0 \land turn = 1)\} \\ & (S_3) & \text{while } & (in_0 \land turn = 1)\} \\ & (S_3) & \text{while } & (in_0 \land turn = 1)\} \\ & (S_3) & \text{while } & (in_0 \land turn = 1)\} \\ & (S_3) & \text{while } & (in_0 \land turn = 1)\} \\ & (S_3) & \text{while } & (in_0 \land turn = 1)\} \\ & (S_3) & \text{while } & (in_0 \land turn = 1)\} \\ & (S_3) & \text{while } & (in_0 \land turn = 1)\} \\ & (S_3) & \text{while } & (in_0 \land turn = 1)\} \\ & (S_3) & \text{while } & (in_0 \land turn = 1)\} \\ & (S_3) & \text{while } & (in_0 \land turn = 1)\} \\ & (S_3) & \text{while } & (in_0 \land turn = 1)\} \\ & (S_3) & \text{while } & (in_0 \land turn = 1)\} \\ & (S_3) & \text{while } & (in_0 \land turn = 1)\} \\ & (S_3) & \text{while } & (in_0 \land turn = 1)\} \\ & (S_3) & \text{while } & (in_0 \land turn = 1)\} \\ & (S_3) & \text{while } & (in_0 \land turn = 1)\} \\ & (S_3) & \text{while } & (in_0 \land turn = 1)\} \\ & (S_3) & \text{while } & (in_0 \land turn = 1)\} \\ & (S_3) & \text{while } & (in_0 \land turn = 1)\} \\ & (S_3) & \text{while } & (in_0 \land turn = 1)\} \\ & (S_3) & \text{while } & (in_0 \land turn = 1) \\ & (S_3) & \text{while } & (in_0 \land turn = 1) \\ & (S_3) & \text{while } & (in_0 \land turn = 1) \\ & (S_3) & \text{while } & (in_0 \land turn = 1) \\ & (S_3) & \text{while } & (in_0 \land turn = 1) \\ & (S_3) & \text{while } & (in_0 \land turn = 1) \\ & (S_3) & \text{while } & (in_0 \land turn = 1) \\ & (S_3) & \text{while } & (in_0 \land turn
```