Too Much Milk

Don Porter

Portions courtesy Emmett Witchel
Critical Sections are Hard, Part 2

- 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

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

• 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
}
```
Solution #0

```c
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

What if we switch the 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 #1

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

```c
while(1) {
    while(turn ≠ Jill) ; //spin
    while (Milk) ; //spin
    buy milk;
    turn := Jack
    // 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
Solution #2: Peterson’s Algorithm

Variables:
- \( in_i \): 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:

\[
\{ (\neg in_j \lor (in_j \land turn = i)) \land in_i \}
\]

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

**Jack**

```java
while (1) {
    in₀ := true;
    turn := Jill;
    while (turn == Jill && in₁) {//wait
        Critical section
    }
    in₀ := false;
    Non-critical section
}
```

**Jill**

```java
while (1) {
    in₁ := true;
    turn := Jack;
    while (turn == Jack && in₀);//wait
        Critical section
    }
    in₁ := false;
    Non-critical section
}
```

Save, live, and bounded waiting; but only 2 threads
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