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

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

The Correctness Conditions

- Safety
  - Only one thread in the critical region
- Liveness
  - A thread that enters the entry section eventually enters the 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
}
Solution #0

```c
while(1) {
    if (noMilk) {  // check milk (Entry section)
        buy milk;
    } else {  // Critical section
        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 some other times
  - Live: note left will be removed
  - Bounded wait (buy milk takes a finite number of steps)

What if we switch the order of checks?

Solution #1

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

- 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: thread T 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:

\[ ((\neg \text{in}_j \cup (\text{turn} = i)) \times \text{in}_i) \]

Intuitively: j doesn’t want to execute or it is i 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