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

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
  - 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

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

```
while(1) {
    if (noMilk) { // check milk (Entry section)
        buy milk;
    } else { // Critical section
        while (Milk) { // spin
            if (noNote) { // check if roommate is getting milk
                leave Note; // Exit section
            } else { // Non-critical region
                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 sometimes 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:
- \( i_j \): thread \( T \) 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 entering the critical section:

\[
(\neg i_j \land (i_j \land \text{turn} = 0) \land \text{in}) \land \text{CS} \\
\land i_j = \text{false} \\
\implies ((\text{turn} = 0) \land (\text{turn} = 1) = \text{false}
\]

Peterson’s Algorithm

```
while (1) { 
    if (i_j) { true: 
        turn = Jill; // initialization
        while (turn = Jill) { // spin
            Critical section
            while (i_j) { // wait
                i_j = false; // Non-critical section
            }
        } 
    } else { false: 
        while (turn = Jack) { // spin
            Critical section
            i_j = false; // Non-critical section
            turn = Jill; // Initialization
        }
    }
}
```

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

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:

\[ (\neg i_j \land (i_j \land \text{turn} = 0) \land \text{in}) = (\neg i_j \land \text{turn} = 0) \land \text{in} \]

Or, intuitively, right after Jack enters:

- Jack has signaled that he is in the entry section (\( i_j \))
- And
  - \( j \) was in the critical section or entry section (\( i_i \))
  - Or
    - \( j \) is also in the entry section but it is Jack’s turn (\( i_j \land \text{turn} = j \))

How can we do that?

```
entry = \( i_j \land \text{false} = \text{true} 
while (i_j \land \text{true} = 0) 
```
**Safe?**

Thread T0
while (terminate)
{ ...
  (m0 = turn = 0); CS0
...
}

Thread T1
while (terminate)
{ ...
  (m1 = turn = 0); CS1
...
}

If both in CS, then

\[ m_0 \land (-m_0 \lor \text{at}(m_0) \lor \text{turn} = 0) \land m_1 \land (-m_1 \lor \text{at}(m_1) \lor \text{turn} = 1) \land \
\text{at}(m_0) \land -\text{at}(m_1) = (\text{turn} = 0) \land (\text{turn} = 1) = \text{false}. \]

**Live?**

Thread T0
while (terminate)
{ ...
  m0 = true; CS0
...
}

Thread T1
while (terminate)
{ ...
  m1 = true; CS1
...
}

Non-blocking: T0 before NCS0, T1, stuck at while loop

\[ \neg \text{R}_1 \land m_0 \land (\text{turn} = 0) = \neg \text{m}_0 \land \text{at}(m_0) \land (\text{turn} = 0) = \text{false}. \]

Deadlock-free: T0 and T1 at while, before entering the critical section

\[ \text{R}_2 \land m_0 \land (\text{turn} = 0) \land (m_1 \land (\text{turn} = 1)) \rightarrow (\text{turn} = 0) \land (\text{turn} = 1) = \text{false}. \]

**Bounded waiting?**

Yup!