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)
        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 some other times

What if we switch the order of checks?

Too Much Milk: Solution #1

turn := Jill // Initialization

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
Solution #2 (a.k.a. Peterson’s algorithm): combine ideas of 0 and 1

Variables:
- \( i_n \): 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 entering the critical section:

\[
\neg (i_0 \vee (i_1 \land turn = 1)) \land i_n
\]

\[
\Rightarrow ((turn = 0) \land (turn = 1)) = false
\]

Peterson’s Algorithm

\[
in_0 = in_1 = false;
\]

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

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

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

\[
\neg in_i \vee (in_j \land turn = i) \land in_i = (\neg in_j \vee turn = i) \land in_i
\]

How can we do that?

```
entry_i := in_i := true;
while (in_j \land turn \neq i):
```
We hit a snag

Thread T₀
while (!terminate) {
    in₀ := true;
    \(\{in₀\}\)
    while \((in₁ \land turn \neq 0)\);
    \(\{in₀ \land (\neg in₁ \lor turn = 0)\}\)
    CS₀
    ........
}\n
Thread T₁
while (!terminate) {
    in₁ := true;
    \(\{in₁\}\)
    while \((in₀ \land turn \neq 0)\);
    \(\{in₁ \land (\neg in₀ \lor turn = 1)\}\)
    CS₁
    ........
}\n
The assignment to \(in₀\) invalidates the invariant!

What can we do?

Add assignment to \(turn\) to establish the second disjunct

Thread T₀
while (!terminate) {
    in₀ := true;
    \(\alpha₀\)
    turn := 1;
    \(\{in₀\}\)
    while \((in₁ \land turn \neq 0)\);
    \(\{in₀ \land (\neg in₁ \lor turn = 0 \lor at(α₀))\}\)
    CS₀
    in₀ := false;
    \(\text{NCS}_₀\)
}\n
Thread T₁
while (!terminate) {
    in₁ := true;
    \(\alpha₁\)
    turn := 0;
    \(\{in₁\}\)
    while \((in₀ \land turn \neq 1)\);
    \(\{in₁ \land (\neg in₀ \lor turn = 1 \lor at(α₀))\}\)
    CS₁
    in₀ := false;
    \(\text{NCS}_₁\)
}
Safe?

Thread T0
while (terminate) {
    in0 := true;
    α0
    turn := 1;
    (in0)
    while (in0 ∧ turn ≠ 0):
        (in0 ∧ (~ in0 ∨ turn = 0 ∨ at(α0)))
    CS0
    in0 := false;
    NCS0
}

Thread T1
while (terminate) {
    in1 := true;
    α1
    turn := 0;
    (in1)
    while (in1 ∧ turn ≠ 1):
        (in1 ∧ (~ in1 ∨ turn = 1 ∨ at(α0)))
    CS1
    in1 := false;
    NCS1
}

If both in CS, then

\[ in_0 \land (\neg in_1 \lor at(\alpha_1) \lor turn = 0) \land in_1 \land (\neg in_0 \lor at(\alpha_0) \lor turn = 1) \land
\neg at(\alpha_0) \land \neg at(\alpha_1) = (turn = 0) \land (turn = 1) = false \]

Live?

Thread T0
while (terminate) {
    (S1: ¬ in0 ∧ (turn = 1 ∨ turn = 0))
    in0 := true;
    (S2: in0 ∧ (turn = 1 ∨ turn = 0))
    α0
    turn := 1;
    (S3)
    while (in0 ∧ turn ≠ 0):
        (S3: in0 ∧ (~ in0 ∨ at(α1) ∨ turn = 0))
    CS0
    (S3)
    in0 := false;
    (S4)
    NCS0
}

Thread T1
while (terminate) {
    (R1: ¬ in1 ∧ (turn = 1 ∨ turn = 0))
    in1 := true;
    (R2: in1 ∧ (turn = 1 ∨ turn = 0))
    α1
    turn := 0;
    (R3)
    while (in1 ∧ turn ≠ 1):
        (R3: in1 ∧ (~ in1 ∨ at(α0) ∨ turn = 1))
    CS1
    (R3)
    in1 := false;
    (R4)
    NCS1
}

Non-blocking: T0 before NCS0, T1 stuck at while loop
\[ S_1 \land R_2 \land in_0 \land (turn = 0) = \neg in_0 \land in_1 \land in_0 \land (turn = 0) = false \]
Deadlock-free: T1 and T0 at while, before entering the critical section
\[ S_2 \land R_2 \land (in_0 \land (turn = 0)) \land (in_1 \land (turn = 1)) \Rightarrow (turn = 0) \land (turn = 1) = false \]
Bounded waiting?

Thread T₀
while (!terminate) {
  indie := true;
  turn := 1;
  while (inde ∨ turn ≠ 0):
    CS₀
  indie := false;
  NCS₀
}

Thread T₁
while (!terminate) {
  indie := true;
  turn := 0;
  while (inde ∨ turn ≠ 1):
    CS₀
  indie := false;
  NCS₀
}

Yup!