Thread Synchronization: Too Much Milk

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.

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
}
```
Too Much Milk: Solution #0

```java
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
        }
    }
}
```

- 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
  - 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 (Peterson’s algorithm):

```java
while(1) {
    turn := Jill; // Initialization
    while(turn == Jill) { //spin
        while(Milk) { //spin
            Jack; //spin
        }
    }
}
```

- 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:
- \(i_j\): 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 thread \(i\) enters the critical section:

\[
((\neg i_j \land (i_j \land \text{turn} = i)) \land (\neg i_\text{turn} = \text{turn} = i)) \implies \text{false}
\]

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

Towards a solution

The problem boils down to establishing the following right after entry:

\[
(\neg i_j \land (i_j \land \text{turn} = i)) \land (\neg i_\text{turn} \land \text{turn} = j) \land i_\text{turn} = \text{false}
\]

Or, intuitively, right after Jack enters:
- Jack has signaled that he is in the entry section (in \(i\))
  - And
    - Jack isn't in the critical section or entry section (\(\neg i_j\))
    - Or
      - Jill is also in the entry section but it is Jack's turn (in \(i_j\) \land \text{turn} = j)

How can we do that?

\[
\text{entry} = i_j = \text{true};
\]

while (in \(i_j\) \land \text{turn} = j)
We hit a snag

Thread T₀
while (terminate)
  \( i₀ = true \)
  while (\( i₀ \lor \neg turn \))
    \( (i₀ = true \lor \neg turn) \land \neg at(i₀) \land at(i₀) \land (turn = 0) \land (turn = 1) = false \)
    \( CS₀ \)
      \( \ldots \)
    \( \neg at(i₀) \land \neg at(i₀) = (turn = 0) \land (turn = 1) = false \)

The assignment to \( i₀ \)
invalidates the invariant!

What can we do?

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

Thread T₁
while (terminate)
  \( i₁ = true \)
  \( \neg turn = 0 \)
  \( \neg i₁ = turn = 0 \lor at(i₁) \)
  \( CS₁ \)
  \( \ldots \)

Thread T₂
while (terminate)
  \( i₂ = true \)
  \( \neg turn = 0 \)
  \( \neg i₂ = turn = 0 \lor at(i₂) \)
  \( CS₂ \)
  \( \ldots \)

Safe?

Thread T₀
while (terminate)
  \( i₀ = true \)
  \( \neg turn = 0 \)
  \( at(i₀) \land \neg at(i₀) \land (turn = 0) \land (turn = 1) = false \)
    \( CS₀ \)
      \( \ldots \)

Thread T₁
while (terminate)
  \( i₁ = true \)
  \( \neg turn = 0 \)
  \( at(i₁) \land \neg at(i₁) \land (turn = 0) \land (turn = 1) = false \)
    \( CS₁ \)
      \( \ldots \)

If both in CS, then
\( i₀ = \neg at(i₀) \land \neg at(i₀) \lor turn = 0 \land turn = 1 \land \neg at(i₀) \land \neg at(i₀) = (turn = 0) \land (turn = 1) = false \)

Live?

Thread T₀
while (terminate)
  \( i₀ = true \)
  \( \neg turn = 0 \)
  \( \neg i₀ = turn = 0 \lor at(i₀) \)
  \( CS₀ \)
  \( \ldots \)

Thread T₁
while (terminate)
  \( i₁ = true \)
  \( \neg turn = 0 \)
  \( \neg i₁ = turn = 0 \lor at(i₁) \)
  \( CS₁ \)
  \( \ldots \)

Non-blocking: \( T₀ \) before NC \( S₀ \), \( T₁ \) stuck at while loop
\( S₀ = R₀ \land \neg i₀ \land \neg \neg at(i₀) \land turn = 0 \land \neg at(i₀) = false \)

Deadlock-free: \( T₀ \) and \( T₁ \) at while, before entering the critical section
\( S₀ = R₀ \land ((i₀ \land (turn = 0)) \land (i₀ \land (turn = 1)) \Rightarrow (turn = 0) \land (turn = 1) = false \)

Bounded waiting?

Thread T₀
while (terminate)
  \( i₀ = true \)
  \( \neg turn = 0 \)
  \( \neg i₀ = turn = 0 \lor at(i₀) \)
  \( CS₀ \)
  \( \ldots \)

Thread T₁
while (terminate)
  \( i₁ = true \)
  \( \neg turn = 0 \)
  \( \neg i₁ = turn = 0 \lor at(i₁) \)
  \( CS₁ \)
  \( \ldots \)

Yup!