Thread Synchronization: Too Much Milk

Concurrent Problem

- Order of thread execution is non-deterministic
  - Multiprocessing
    - A system may contain multiple processors cooperating
    - threads/processes can execute simultaneously
  - Multi-programming
    - Thread/process execution can be interleaved because of time-slicing
- Operations are often not "atomic"
  - Example: \( x = x + 1 \) is not atomic!
- Goal:
  - Ensure that your concurrent program works under ALL possible interleavings

The Fundamental Issue

- In all these cases, what we thought to be an atomic operation is not done atomically by the machine.
- **Definition:** An atomic operation is one that executes to completion without any interruption or failure.
- An atomic operation has "an all or nothing" flavor:
  - Either it executes to completion, or
  - it did not execute at all, and
  - it executes without interruptions

Atomic = no one can see a partially-executed state! Key challenge: how to implement atomic semantics?

Critical Sections

- A critical section is an abstraction that
  - consists of a number of consecutive program instructions
  - all code within the section executes atomically
- Critical sections are used profusely in an OS to protect data structures (e.g., queues, shared variables, lists, ...)
- A critical section implementation must be:
  - correct: for a given \( k \), only \( k \) thread can execute in the critical section at any given time (usually, \( k = 1 \))
  - efficient: getting into and out of critical section must be fast
  - concurrency control: a good implementation allows maximum concurrency while preserving correctness
  - flexible: a good implementation must have as few restrictions as practically possible
**Safety and Liveness**

- **Safety property:** "nothing bad happens"
  - holds in every finite execution prefix
    - Windows™ never crashes
    - if one general attacks, both do
    - a program never terminates with a wrong answer

- **Liveness property:** "something good eventually happens"
  - no partial execution is irreparable
    - Windows™ always reboots
    - both generals eventually attack
    - a program eventually terminates

---

**A really cool theorem**

*Every property is a combination of a safety property and a liveness property*

(Alpern and Schneider)

---

**Nice, but... what's your point?**

- **Safety:** At most k threads are concurrently in the critical section

- **Liveness:** A thread that wants to enter the critical section, will eventually succeed

- **Anything else?**
  - Bounded waiting: If a thread i is in entry section, then there is a bound on the number of times that other threads are allowed to enter the critical section before thread i's request is granted

  *Is bounded waiting a safety or a liveness property?*

---

**CriticalSection: Implementation**

- **Basic idea:**
  - Restrict programming model
    - Permit access to shared variables only within a critical section

- **General program structure**
  - Entry section
    - "Lock" before entering critical section
    - Wait if already locked
    - Key point: synchronization may involve wait
  - Critical section code
  - Exit section
    - "Unlock" when leaving the critical section

- **Object-oriented programming style**
  - Associate a lock with each shared object
  - Methods that access shared object are critical sections
  - Acquire/release locks when entering/leaving a method that defines a critical section
**Thread Coordination: Reality TV!**

**Too much milk!**

Jack  
- Look in the fridge; out of milk  
- Leave for store  
- Arrive at store  
- Buy milk  
- Arrive home; put milk away

Jill  
- Look in fridge; out of milk  
- Leave for store  
- Arrive at 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?

---

**Too Much Milk: Solution #0**

```
while (turn ≠ Jack); // relax  
while (Milk); // relax  
buy milk;  
turn := Jill
```

- Will this solution work?  
- Safe? Yes!  
  - Must have turn to buy milk
- Live?  
  - What if the other guy never comes around to check the milk...  
- Bounded waiting?  
  - Sure, and the bound is 1!

---

**Too Much Milk: Solution #1**

```
if (noMilk) {  // check milk  
  if (noNote) { // check if roommate is getting milk  
    leave Note;  
    buy milk;  
    remove Note;  
  }
}
```

- Will this solution work?  
- Safe? No!  
  - Threads can get context switched after checking whether there is a note, but before leaving a note
- Live? Yes  
  - A note left will be eventually removed
- Bounded waiting?  
  - Sure, and the bound is 1!

---

*What if we switch the order of checks?*
**Too Much Milk: Solution #2**

- Safe?
- Live?
- What happens if note has no color?

**Solution #3 (a.k.a. Peterson's algorithm): combine ideas of 0 and 2**

Variables:
- \( \tau \): 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:

\[
\begin{align*}
&\neg \text{in}_\tau \lor (\text{in}_\tau \land \text{turn} = \tau) \\
&\neg \text{in}_0 \lor (\text{in}_0 \land \text{turn} = 0) \\
&\text{in}_1 = \text{false}
\end{align*}
\]

Towards a solution

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

\( \neg \text{in}_\tau \lor (\text{in}_\tau \land \text{turn} = \tau) \lor \neg \text{in}_0 \lor (\neg \text{in}_0 \land \text{turn} = 0) \lor \text{in}_1 \)

How can we do that?

```
entry: in_0 = true; while (in_0 && turn != 0);
```

We hit a snag

The assignment to \( \text{in}_0 \) invalidates the invariant!
What can we do?

Add assignment to turn to establish the second disjunct

Thread T

while (terminate) {
    a₀ = true;
    turn := 1;
    in
    while (a₀ && turn = 0):
        (in) = a₀; turn := 0; at());
        CS;
        a₀ := false;
        NCSTₙ;
    }


Safe?

Thread T

while (terminate) {
    a₀ = true;
    turn := 1;
    (in)
    while (a₀ && turn = 0):
        (in) = a₀; turn := 0; at());
        CS;
        a₀ := false;
        NCSTₙ;
    }

If both in CS, then

\[ (a₀ && \neg a₀) \implies (\text{turn} = 0) \implies (a₀ && \neg a₀) \implies (\text{turn} = 1) \implies (\neg a₀ && \text{turn} = 0) \implies (a₀ && \text{turn} = 0) \implies (\text{turn} = 1) = \text{false} \]

Live?

Thread T

while (terminate) {
    a₀ = true;
    turn := 1;
    in
    while (a₀ && turn = 0):
        (in) = a₀; turn := 0; at());
        CS;
        a₀ := false;
        NCSTₙ;
    }

Non-blocking: Tₙ before NCSTₙ, Tₙ stuck at while loop
\[ Sₙ \land Rₙ \land aₙ \land (\text{turn} = 0) \land \neg aₙ \land \neg (\text{turn} = 0) = \text{false} \]

Deadline-free: Tₙ and Tₙ at while, before entering the critical section
\[ Sₙ \land Rₙ \land (\text{turn} = 0) \land (aₙ \land (\text{turn} = 1)) \land (\text{turn} = 0) \land (\text{turn} = 1) = \text{false} \]

Bounded waiting?

Thread T

while (terminate) {
    a₀ = true;
    turn := 1;
    while (a₀ && turn = 0):
        (in) = a₀; turn := 0; at());
        CS;
        a₀ := false;
        NCSTₙ;
    }

Thread T

while (terminate) {
    a₀ = true;
    turn := 1;
    while (a₀ && turn = 0):
        (in) = a₀; turn := 0; at());
        CS;
        a₀ := false;
        NCSTₙ;
    }

Yup!
Too Much Milk: Lessons

- Last solution works, but it is really unsatisfactory
  - 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?
  - Define higher-level programming abstractions to simplify concurrent programming.
  - Use hardware features to eliminate busy waiting.
  - Stay tuned...