Thread Synchronization: Too Much Milk

Concurrency 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 interleaving

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 irremediable
    - 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?

Critical Section: 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/exiting a method that defines a critical section

Textbook shows non-OO examples; much easier to think OO

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
```

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

- 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!

Introduce the concept of a note
- Leave a note = lock
- Remove note = unlock
- Don't buy if note = wait

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?
- This solution is worse than before!!
  - It works sometime and doesn't some other times

What if we switch the order of checks?

Too Much Milk: Solution #2

```
Jack
Leave Blue note
If (noNote Pink)
    if (noMilk) {
        buy milk;
    }
}
Remove Blue note
```

```
Jill
Leave Pink note
If (noNote Blue) {
    if (noMilk) {
        buy milk;
    }
}
Remove Pink note
```

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

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

Variables:

- in_i: 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:

$\{(\neg in_j \quad (in_j \square\ turn = i)) \square\ in_i\}$

CS

....

$in_i = false$

$((\neg in_0 \quad (in_0 \square\ turn = 1)) \square\ in_1) \square$
$((\neg in_1 \quad (in_1 \square\ turn = 0)) \square\ in_0)$
$\square$
$((turn = 0) \square\ (turn = 1)) = false$

Towards a solution

The problem boils down to establishing the following right after $entry_i$

$(\neg in_j \quad (in_j \square\ turn = i)) \square\ in_i = (\neg in_j \quad turn = i) \square\ in_i$

How can we do that?

$entry_i$ = in_i := true;
while (in_j $\square$ $turn \neq i$);

We hit a snag

Thread T_0
while (!terminate) {
in_0:= true
$\{in_0\}$
while (in_1 $\square$ $turn \neq 0$);
$\{in_0 \square\ (\neg in_1 \quad turn = 0)\}$
CS_0
....
}

Thread T_1
while (!terminate) {
in_1:= true
$\{in_1\}$
while (in_0 $\square$ $turn \neq 1$);
$\{in_1 \square\ (\neg in_0 \quad turn = 1)\}$
CS_1
....
}

The assignment to in_0 invalidates the invariant!

What can we do?

Add assignment to *turn* to establish the second disjunct

```
Thread T0
while (!terminate) {
□0  in0:= true;
    turn := 1;
    {in0}
    while (in1 □turn ≠ 0);
    {in0 □ (¬ in1   turn = 0   at(□1) )}
    CS0
    in0 := false;
    NCS0
}
```

```
Thread T1
while (!terminate) {
□1  in1:= true;
    turn := 0;
    {in1}
    while (in0 □turn ≠ 1);
    {in1 □ (¬ in0   turn = 1   at(□0) )}
    CS1
    in1 := false;
    NCS1
}
```


Safe?

```
Thread T0
while (!terminate) {
□0  in0:= true;
    turn := 1;
    {in0}
    while (in1 □turn ≠ 0);
    {in0 □ (¬ in1   turn = 0   at(□1) )}
    CS0
    in0 := false;
    NCS0
}
```

```
Thread T1
while (!terminate) {
□1  in1:= true;
    turn := 0;
    {in1}
    while (in0 □turn ≠ 1);
    {in1 □ (¬ in0   turn = 1   at(□0) )}
    CS1
    in1 := false;
    NCS1
}
```

If both in CS, then

in_0 □ (¬in_1 at(□$_1$) $turn$ = 0) □ in_1 □ (¬in_0 at(□$_0$) $turn$ = 1) □

□ ¬ at(□$_0$) □ ¬ at(□$_1$) = ($turn$ = 0) □ ($turn$ = 1) = false

Live?

```
Thread T0
while (!terminate) {
    {S1: ¬in0 □ (turn = 1   turn = 0)}
    in0:= true;
□0  {S2: in0 □ (turn = 1   turn = 0)}
    turn := 1;
    {S2}
    while (in1 □turn ≠ 0);
    {S3: in0 □ (¬ in1   at(□1)   turn = 0)}
    CS0
    {S3}
    in0 := false;
    {S1}
    NCS0
}
```

```
Thread T1
while (!terminate) {
    {R1: ¬in0 □ (turn = 1   turn = 0)}
    in1:= true;
□1  {R2: in0 □ (turn = 1   turn = 0)}
    turn := 0;
    {R2}
    while (in0 □turn ≠ 1);
    {R3: in1 □ (¬ in0   at(□0)   turn = 1)}
    CS1
    {R3}
    in1 := false;
    {R1}
    NCS1
}
```

Non-blocking: T_0 before NCS_0, T_1 stuck at **while** loop

S_1 □ R_2 □ in_0 □ ($turn$ = 0) = ¬in_0 □ in_1 □ in_0 □ ($turn$ = 0) = false

Deadlock-free: T_1 and T_0 at **while,** before entering the critical section

S_2 □ R_2 □ (in_0 □ ($turn$ = 0)) □ (in_1 □ ($turn$ = 1)) □ ($turn$ = 0) □ ($turn$ = 1) = false

Bounded waiting?

```
Thread T0
while (!terminate) {
    in0:= true;
    turn := 1;
    while (in1 □turn ≠ 0);
    CS0
    in0 := false;
    NCS0
}
```

```
Thread T0
while (!terminate) {
    in1:= true;
    turn := 0;
    while (in0 □turn ≠ 1);
    CS0
    in1 := false;
    NCS0
}
```

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…