Thread Synchronization: Too Much Milk

Critical Sections
- A critical section is a segment of code involved in reading and writing a shared data area.
- Critical sections are used profusely in an OS to protect data structures (e.g., queues, shared variables, lists...)
- Key assumptions:
  - Finite Progress Axiom: Processes execute at a finite, but otherwise unknown, speed.
  - Processes cannot halt (by failing, or just terminating) inside critical section.

Safety and Liveness
- Safety property: “nothing bad happens”
  - holds in every finite execution prefix
    - Windows™ never crashes
    - No patient is ever given the wrong medication
    - A program never terminates with a wrong answer
- Liveness property: “something good eventually happens”
  - no partial execution is irremediable
    - Windows™ always reboots
    - Medications are eventually distributed to patients
    - A program eventually terminates

A really cool theorem
Every property defined on an execution of a program is a combination of a safety property and a liveness property

(Alpern and Schneider)
Nice, but... what's your point?

- Mutual Exclusion: At most $k$ threads are concurrently in the critical section (Safety)
- Access Opportunity: A thread that wants to enter the critical section, will eventually succeed (Liveness)
- 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 (Safety)

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
    - Processes do not halt in CS, Entry or Exit Sections
- 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

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 in fridge

Jill
- Look in fridge; out of milk
- Leave for store
- Arrive at store
- Buy milk
- Arrive home; put milk in fridge
- Oh, no!

Milk provision is shared state!

Formalizing “Too Much Milk”

- Shared variables
  - "Look in the fridge for milk" - check a variable
  - "Put milk in fridge" - 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: Taking turns

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!

```c
while (turn ≠ Jack); // relax
while (Milk); // relax
buy milk;
```

To avoid deadlock:
- If turn = Jill, then
  ```
  while (Milk); // relax
  ```

```
turn := Jill;
```

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

Too Much Milk: Solution #1: Notes

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 sometimes and doesn’t some other times

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

Solution #2 (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 for process i before i enters the critical section:

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

CS

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

in\_i = false

in\_i \land (turn = i) = false
Towards a solution

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

\[ (\neg \text{in}_j \land (\text{turn} = i)) \land \text{in}_j = \neg (\neg \text{in}_j \land \text{turn} = j) \land \text{in}_j \]

How can we do that?

\[
\text{entry}, \; \text{in}_i = \text{true} ; \\
\text{while} (\text{in}_i \land \text{turn} = j) ;
\]

We hit a snag

Thread T_0
\[
\text{while} (\neg \text{terminate}) \{ \\
\text{in}_0 = \text{true}; \\
\text{turn} = 1; \\
\text{while} (\text{in}_1 \land \text{turn} = 0) ; \\
(\text{in}_1 \land \neg \text{in}_0 \land \text{turn} = 1 \lor \text{at}(\alpha_1)) \} \\
\text{CS}_0 \\
\text{NCS}_0 \\
\}
\]

Thread T_1
\[
\text{while} (\neg \text{terminate}) \{ \\
\text{in}_1 = \text{true}; \\
\text{turn} = 0; \\
\text{while} (\text{in}_0 \land \text{turn} = 1) ; \\
(\text{in}_0 \land \neg \text{in}_1 \land \text{turn} = 0 \lor \text{at}(\alpha_0)) \} \\
\text{CS}_1 \\
\text{NCS}_1 \\
\}
\]

The assignment to \text{in}_0 invalidates the invariant!

What can we do?

Add assignment to \text{turn} to establish the second disjunct

Thread T_0
\[
\text{while} (\neg \text{terminate}) \{ \\
\text{in}_0 = \text{true}; \\
\text{turn} = 1; \\
\text{while} (\text{in}_1 \land \text{turn} = 0) ; \\
(\text{in}_1 \land \neg \text{in}_0 \land \text{turn} = 1 \lor \text{at}(\alpha_1)) \} \\
\text{CS}_0 \\
\text{NCS}_0 \\
\}
\]

Thread T_1
\[
\text{while} (\neg \text{terminate}) \{ \\
\text{in}_1 = \text{true}; \\
\text{turn} = 0; \\
\text{while} (\text{in}_0 \land \text{turn} = 1) ; \\
(\text{in}_0 \land \neg \text{in}_1 \land \text{turn} = 0 \lor \text{at}(\alpha_0)) \} \\
\text{CS}_1 \\
\text{NCS}_1 \\
\}
\]

Safe?

Thread T_0
\[
\text{while} (\neg \text{terminate}) \{ \\
\text{in}_0 = \text{true}; \\
\text{turn} = 1; \\
\text{while} (\text{in}_1 \land \text{turn} = 0) ; \\
(\text{in}_1 \land \neg \text{in}_0 \land \text{turn} = 1 \lor \text{at}(\alpha_1)) \} \\
\text{CS}_0 \\
\text{NCS}_0 \\
\}
\]

Thread T_1
\[
\text{while} (\neg \text{terminate}) \{ \\
\text{in}_1 = \text{true}; \\
\text{turn} = 0; \\
\text{while} (\text{in}_0 \land \text{turn} = 1) ; \\
(\text{in}_0 \land \neg \text{in}_1 \land \text{turn} = 0 \lor \text{at}(\alpha_0)) \} \\
\text{CS}_1 \\
\text{NCS}_1 \\
\}
\]

If both in CS, then
\[
\text{in}_0 \land (\neg \text{in}_1 \lor \text{at}(\alpha_1) \lor \text{turn} = 0) \land \text{in}_1 \land (\neg \text{in}_0 \lor \text{at}(\alpha_0) \lor \text{turn} = 1) \land \\
\neg \text{at}(\alpha_0) \land \neg \text{at}(\alpha_1) \land (\text{turn} = 0 \land (\text{turn} = 1) = \text{false})
\]
Live?

Thread T₀

while (!terminate) {
  {S₁: ¬in₀ ∧ (turn = 1 ∨ turn = 0)}
  in₀ := true;
  {S₁}
  turn := 1;
  {S₂: in₀ ∧ (¬in₁ ∧ at(α₁) ∨ turn = 0)}
  CS₀
  in₀ := false;
  {S₂}
  NCS₀
}

Thread T₁

while (!terminate) {
  {R₁: ¬in₀ ∧ (turn = 1 ∨ turn = 0)}
  in₁ := true;
  {R₁}
  turn := 0;
  {R₂: in₀ ∧ (¬in₁ ∧ at(α₀) ∨ turn = 0)}
  CS₁
  in₁ := false;
  {R₂}
  NCS₁
}

Non-blocking: T₀ before NCS₀, T₁ stuck at while loop
S₁ ∧ R₁ ∧ ¬α₀ ∧ (turn = 0) ⇒ ¬in₀ ∧ ¬in₁ ∧ ¬α₀ ∧ (turn = 0) ∧ false
Deadlock-free: T₁ and T₀ at while, before entering the critical section
S₂ ∧ R₂ ∧ ¬α₀ ∧ (turn = 0) ∧ (in₀ ∧ (turn = 1)) ⇒ (turn = 0) ∧ (turn = 1) ∧ false

Bounded waiting?

Thread T₀

while (!terminate) {
  {R₀: ¬in₀ ∧ (turn = 1 ∧ turn = 0)}
  in₀ := true;
  {R₀}
  turn := 1;
  while (in₁ ∧ turn ≠ 0);
  CS₀
  in₀ := false;
  {R₀}
  NCS₀
}

Thread T₁

while (!terminate) {
  {R₁: ¬in₁ ∧ (turn = 1 ∧ turn = 0)}
  in₁ := true;
  {R₁}
  turn := 0;
  while (in₀ ∧ turn ≠ 1);
  CS₁
  in₁ := false;
  {R₁}
  NCS₁
}

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...