Thread Synchronization:
Too Much Milk

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, 1987)

Safety and Liveness
Properties defined over an execution of a program

- **Safety**: “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**: “something good eventually happens”
  - no partial execution is irreparable
    - Windows™ always reboots
    - Medications are eventually distributed to patients
    - A program eventually terminates

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 can halt only outside of the critical section (by failing, or just terminating)
Critical Section

- Mutual Exclusion: At most $k$ threads are concurrently in the critical section
- Access Opportunity: A thread that wants to enter the critical section will eventually succeed
Critical Section

- **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

Critical Section: General Program Structure

- **Entry section**
  - "Lock" before entering critical section
  - Wait if already locked
- **Critical Section code**
- **Exit section**
  - "Unlock" when leaving the critical section

**OO programming style**
- Associate a lock with each shared object
- Methods that access shared objects are critical section
- Acquire/release locks when entering/leaving a method that defines a critical section

Too Much Milk!

- **Jack**
  - Look in the fridge: out of milk
  - Leave for store
  - Arrive at store
  - Buy milk
  - Arrive at home: put milk away
- **Jill**
  - Look in fridge: out of milk
  - Leave for store
  - Arrive at store
  - Buy milk
  - Arrive at home: put milk away
  - Oh no!
Formalizing “Too Much Milk”

- Shared variables
  - “Look in the fridge for milk” - check a variable
  - “Put milk away” - update a variable
- Safety
  - At most one person buys milk
- Liveness
  - Someone buys milk when needed

Solution #0: Taking Turns

```plaintext
procedure Check-Milk
while(turn ! Jack) relax;
while (Milk) relax;
buy milk;
turn := Jill
```

Solution #0: Taking Turns

```plaintext
procedure Check-Milk
while(turn ! Jill) relax;
while (Milk) relax;
buy milk;
turn := Jack
```

Safe? Why?
- True, False
Live? Why?
- True, False
Bounded waiting?
- True, False

Safe? Yes!
- it is either Jack’s or Jill turn
Live? No
- what if the other guy stops checking milk?
Bounded waiting? Yes
- ... and the bound is 1!
Solution #1: Leave a note

- Leave note = lock
- Remove note = unlock
- If you find a note from your roommate—don’t buy!

Safe? Live? Bounded waiting? Why?

Solution #2: Colors

- Jack
  - Leave Blue note if (noPinknote) {
    if (noMilk) {
      buy milk;
    }
  }
  - Remove Blue note

- Jill
  - Leave Pink note if (noBlueNote) {
    if (noMilk) {
      buy milk;
    }
  }
  - Remove Pink note

Safe? Live? Bounded waiting? Why?

BTW, what if a note has no color?

Solution #3 (Peterson’s): combine ideas from #0 & #2

- We introduce two variables:
  - \( in_i \): thread \( T_i \) is executing in CS, or trying to do so
  - \( turn_i \): id of thread allowed to enter CS under contention

- Claim: If the following invariant holds when \( T_i \) enters the critical section, so does mutual exclusion

\[
in_i \land (\neg in_j \lor (in_j \land turn = i))
\]
Towards a solution

- The problem then boils down to establishing the following:
  \[ \text{in}_i \land (\neg \text{in}_j \lor (\text{in}_j \land \text{turn} = i)) = \text{in}_i \land (\neg \text{in}_j \lor \text{turn} = i) \]

- How can we do that?

  \[
  \begin{array}{l}
  \text{entry}_i : \text{in}_i := \text{true} \\
  \text{while} \ (\text{in}_j \land \text{turn} \neq i)
  \end{array}
  \]

A first fix

- Add assignment to turn to establish the second disjunct.

  **Thread T₀**
  while (in₀ := true)
  \[
  \begin{array}{l}
  \text{in}_0 := \text{true}; \\
  \text{turn} := 1; \\
  \{\text{in}_0\} \\
  \text{while} \ (\text{in}_1 \land \text{turn} \neq 0) \land (\text{in}_0 \land (\neg \text{in}_1 \lor \text{turn} = 0)) \\
  \text{CS}_0 \\
  \text{in}_0 := \text{true}; \\
  \text{NCS}_0
  \end{array}
  \]

  **Thread T₁**
  while (in₁ := true)
  \[
  \begin{array}{l}
  \text{in}_1 := \text{true}; \\
  \text{turn} := 0; \\
  \{\text{in}_1\} \\
  \text{while} \ (\text{in}_0 \land \text{turn} \neq 1) \land (\text{in}_1 \land (\neg \text{in}_0 \lor \text{turn} = 1)) \\
  \text{CS}_1 \\
  \text{in}_1 := \text{true}; \\
  \text{NCS}_1
  \end{array}
  \]

We hit a snag

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

A dirty trick

- To establish the invariant, we add an auxiliary variable \( \alpha \) that tracks the position of the PC.

  **Thread T₀**
  while (in₀ := true)
  \[
  \begin{array}{l}
  \text{in}_0 := \text{true}; \\
  \text{turn} := 0; \\
  \{\text{in}_0\} \\
  \text{while} \ (\text{in}_0 \land \text{turn} \neq 0) \lor (\text{in}_0 \land (\neg \text{in}_0 \lor \text{turn} = 0) \\
  \text{CS}_0 \\
  \text{in}_0 := \text{false}; \\
  \text{NCS}_0
  \end{array}
  \]

  **Thread T₁**
  while (in₁ := true)
  \[
  \begin{array}{l}
  \text{in}_1 := \text{false}; \\
  \text{turn} := 1; \\
  \{\text{in}_1\} \\
  \text{while} \ (\text{in}_0 \land \text{turn} \neq 1) \lor (\text{in}_1 \land (\neg \text{in}_0 \lor \text{turn} = 1) \\
  \text{CS}_1 \\
  \text{in}_1 := \text{false}; \\
  \text{NCS}_1
  \end{array}
  \]
Is Peterson safe?

while (!terminate) {
  io := true;
  turn := 1;
  {io}
  while (io ∧ turn ≠ 0);
  {io ∧ (¬io ∨ turn = 0 ∨ at(α1))}
  CS0
  io := false;
  NCS0
}
NCS0

If both in the critical section, then:
\[\text{io} \land (\neg\text{io} \lor \text{turn} = 0 \lor \text{at}(\alpha_1)) = (\text{turn} = 0) \land (\text{turn} = 1) = \text{false}\]

Live: Non-blocking

while (!terminate) {
  io := true;
  turn := 0;
  {io}
  while (io ∧ turn ≠ 0);
  {io ∧ (¬io ∨ turn = 1 ∨ at(α0))}
  CS0
  io := false;
  NCS0
}

Blocking Scenario: T0 before NCS0, T1 stuck at while loop
\[S_1 ∧ R_2 ∧ io ∧ (\text{turn} = 0) = \neg io ∧ io ∧ (\text{turn} = 0) = \text{false}\]

Live: Deadlock-free

while (!terminate) {
  io := true;
  turn := 0;
  {io}
  while (io ∧ turn ≠ 0);
  {io ∧ (¬io ∨ turn = 1 ∨ at(α0))}
  CS0
  io := false;
  NCS0
}
NCS0

Deadlock-scenario: T1 and T0 at while, before entering the critical section
\[S_1 ∧ R_2 ∧ (io ∧ (\text{turn} = 0)) ∧ (io ∧ (\text{turn} = 1)) ⇒ (\text{turn} = 0) \land (\text{turn} = 1) = \text{false}\]

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