Thread Synchronization: Foundations

Properties

Property: a predicate that is evaluated over a run of the program (a trace)

“every message that is received was previously sent”

Not everything you may want to say about a program is a property:

“the program sends an average of 50 messages in a run”

Safety properties

“Nothing bad happens”

- No more than \( k \) processes are simultaneously in the critical section
- Messages that are delivered are delivered in FIFO order
- No patient is ever given the wrong medication
- Windows never crashes

A safety property is “prefix closed”:

- if it holds in a run, it holds in its every prefix
Liveness properties

- “Something good eventually happens”
  - A process that wishes to enter the critical section eventually does so
  - Some message is eventually delivered
  - Medications are eventually distributed to patients
  - Windows eventually boots
- Every run can be extended to satisfy a liveness property
  - If it does not hold in a prefix of a run, it does not mean it may not hold eventually

A really cool theorem

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

(Alpern & Schneider)

Critical Section

- A segment of code involved in reading and writing a shared data area
- 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, positive, but otherwise unknown, speed.
  - Processes can halt only outside of the critical section (by failing, or just terminating)
  - Wait-free synchronization (Herlihy, 1991)

Critical Section

- Mutual Exclusion: At most \( k \) threads are concurrently in the critical section (Safety)
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)

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/exiting a method that defines a critical section

Too Much Milk
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: no 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 variable “milk”
  - “Put milk away” - increment “milk”

- **Safety**
  - At most one person buys milk

- **Liveness**
  - If milk is needed, eventually somebody buys milk

Solution #0: Taking Turns

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

Jill
```
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
Solution #0: Taking Turns

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

- 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

```pseudocode
procedure Check-Milk
if (noMilk) {
  if (noNote) {
    leave Note;
    buy milk;
    remove Note
  }
}
```

- Leave note = lock
- Remove note = unlock
- If you find a note from your roommate don’t buy!
- Safe? Live? Bounded waiting? Why?
Solution #1: Leave a note

- If you find a note from your roommate don't buy!
  - Leave note ≈ lock
  - Remove note ≈ unlock

Safe?

Jack/Jill

if (milk==0) {
  if (!note) {
    note = True;
    milk++;
    note = False;
  }
}

T1
if (milk==0) {
  if (note) {
    note = True;
    milk++;
    note = False;
  }
}

T2
Oh no!

Solution #2: Colored Notes

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

What if we leave the note first?

if (noMilk) {
  if (noNote) {
    leave Note;
    buy milk;
    remove Note
  }
}

This "solution" makes the problem worse!
- sometime it works, sometime it doesn't
Solution #2: Colored Notes

**Jack**

```java
BlueNote = 1;
A1 if (PinkNote == 0) {
    if (milk == 0) {
        milk++;
    }
}
BlueNote = 0;
```

**Jill**

```java
PinkNote = 1;
A1 if (BlueNote == 0) {
    if (milk == 0) {
        milk++;
    }
}
PinkNote = 0;
```

**Proof of Safety**

By contradiction:
Suppose Jack and Jill both buy milk
Consider state of variables (PinkNote, milk) at A1

Case 1: PinkNote == 1
Impossible, since Jack ends up buying milk.

Case 2: PinkNote == 0, milk > 0
Impossible. Milk > 0 is a stable property, so Jack would fail test A2 and never buy milk.

Case 3: PinkNote == 0, milk == 0
Impossible. Jill cannot be executing in B1-B2 (PinkNote is not 1!)
Since (BlueNote==1 or milk>0) is stable, then Jill will not pass B2.

Solution #3

**Jack**

```java
BlueNote = 1;
A1 if (PinkNote == 0) {
    if (milk == 0) {
        milk++;
    }
}
BlueNote = 0;
```

**Jill**

```java
PinkNote = 1;
A1 if (BlueNote == 0) {
    if (milk == 0) {
        milk++;
    }
}
PinkNote = 0;
```

**Proof of Safety**

Similar to previous case.

**Proof of Liveness**

Jill will eventually set PinkNote = 0 (no loops)
Jill will then reach line A1
If Jack finds milk, done
If still no milk, Jack will buy it.

Too Much Milk: Lessons

- Last solution works, but it is really unsatisfactory:
  - Complicated; proving correctness is tricky even for the simple example
  - Inefficient: while thread is waiting, it is consuming CPU time
  - Asymmetric: hard to scale to many threads
  - Incorrect(?): instruction reordering can produce surprising results
Solution #3.1 (Peterson's):
combine ideas from #0 & #2

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

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

\[
in_i \land (\neg in_j \lor (in_j \land \text{turn} = i))
\]

How do we prove it?

Towards a solution

- The problem then boils down to establishing the following:

\[
in_i \land (\neg in_j \lor (in_j \land \text{turn} = i)) = in_i \land (\neg in_j \lor \text{turn} = i)
\]

- How can we do that?

\[
\text{entry}_i : \ in_i := \text{true}
\]

while \((in_j \land \text{turn} \neq i)\)

A first fix

- Add assignment to \( \text{turn} \) to establish second disjunct

Thread \( T_0 \)

while(!terminate) {
\[
\begin{align*}
  &in_0 := \text{true} \\
  &in_1 := \text{true} \\
  &\text{turn} = 1 \\
  &\text{turn} = 0 \\
  &\text{while} \((in_1 \land \text{turn} \neq 0)\) \\
  &\text{while} \((in_0 \land \text{turn} \neq 1)\) \\
  &CS_0 \\
  &\ldots \\
  &in_0 = \text{false} \\
  &NCS_0
\end{align*}
\]

Thread \( T_1 \)

while(!terminate) {
\[
\begin{align*}
  &in_0 := \text{true} \\
  &in_1 := \text{true} \\
  &\text{turn} = 1 \\
  &\text{turn} = 0 \\
  &\text{while} \((in_1 \land \text{turn} \neq 0)\) \\
  &\text{while} \((in_0 \land \text{turn} \neq 1)\) \\
  &CS_0 \\
  &\ldots \\
  &in_0 = \text{false} \\
  &NCS_0
\end{align*}
\]

A dirty trick

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

Thread \( T_0 \)

while(!terminate) {
\[
\begin{align*}
  &in_0 := \text{true} \\
  &in_0 \\
  &\text{turn} = 1 \\
  &\text{turn} = 0 \\
  &\text{while} \((in_1 \land \text{turn} \neq 0)\) \\
  &\text{while} \((in_0 \land \text{turn} \neq 1)\) \\
  &\{in_0 \land (\neg in_0 \lor \text{turn} = 1) \land \text{at}(\alpha_0)\} \\
  &CS_0 \\
  &\ldots \\
  &in_0 = \text{false} \\
  &NCS_0
\end{align*}
\]

Thread \( T_1 \)

while(!terminate) {
\[
\begin{align*}
  &in_1 := \text{true} \\
  &in_0 \\
  &\text{turn} = 1 \\
  &\text{turn} = 0 \\
  &\text{while} \((in_0 \land \text{turn} \neq 1)\) \\
  &\text{while} \((in_1 \land \text{turn} \neq 0)\) \\
  &\{in_1 \land (\neg in_0 \lor \text{turn} = 1) \land \text{at}(\alpha_0)\} \\
  &CS_1 \\
  &\ldots \\
  &in_1 = \text{false} \\
  &NCS_1
\end{align*}
\]

but these invariants do not hold!
Is Peterson safe?

Thread T₀
while(!terminate)
    \(i_{n₀} := \text{true}\)
    \(i_{n₀}, \alpha_₁ \) turn = 0
while(\(i_{n₁} \land \neg \alpha_₁\))
    \(i_{n₁}, \alpha_₁ \) turn = 0
\(i_{n₀} \land \neg \alpha_₁ \land \neg \alpha_0 \land \neg \text{at}(α₁)\) = \((\text{turn} = 0) \land (\text{turn} = 1) = \text{false}\)

Thread T₁
while(!terminate)
    \(i_{n₁} := \text{true}\)
    \(i_{n₁}, \alpha_₁ \) turn = 0
while(\(i_{n₀} \land \neg \alpha_₁\))
    \(i_{n₀}, \alpha_₀ \) turn = 0
\(i_{n₁} \land \neg \alpha_₀ \land \neg \text{at}(α₀)\) = \((\text{turn} = 0) \land (\text{turn} = 1) = \text{false}\)

Live: Non-blocking

while(!terminate)
    \(i_{n₀} = \text{true}\)
    \(i_{n₀}, \alpha_₀ \) turn = 1
while(\(i_{n₁} \land \neg \alpha_₀\))
    \(i_{n₁}, \alpha₁ \) turn = 1
\(i_{n₀} \land \neg \alpha₁ \land \neg \text{at}(α₁)\) = \((\text{turn} = 0) \land (\text{turn} = 1) = \text{false}\)

Live: Deadlock-free

while(!terminate)
    \(i_{n₀} = \text{true}\)
    \(i_{n₀}, \alpha_₀ \) turn = 1
while(\(i_{n₁} \land \neg \alpha_₀\))
    \(i_{n₁}, \alpha₁ \) turn = 1
\(i_{n₀} \land \neg \alpha₁ \land \neg \text{at}(α₁)\) = \((\text{turn} = 0) \land (\text{turn} = 1) = \text{false}\)

Blocking Scenario: T₀ before NCS₀, T₁ stuck at while loop
\(R₁ \land S₂ \land i_{n₀} \land (\text{turn} = 0) = \neg i_{n₀} \land i_{n₁} \land i_{n₀} \land (\text{turn} = 0) = \text{false}\)

A better way

🤔 How can we do better?
- Define higher-level programming abstractions (shared objects, synchronization variables) to simplify concurrent programming
- lock.acquire() - wait until lock is free, then grab it • atomic
- lock.release() - unlock, waking up a waiter, if any • atomic

```java
JackJill(even Dome Bub) {
    lock.acquire();
    if (milk == 0) {
        milk++;
        lock.release();
    }
}
```

- Use hardware to support atomic operations beyond load and store