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 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

Edsger’s perspective

Testing can only prove the presence of bugs...

...not their absence!
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!

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
```

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

```plaintext
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

```plaintext
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?

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

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

This “solution” makes the problem worse!

sometime it works, sometime it doesn’t

What if we leave the note first?

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

Solution #2: Colored Notes

Jack

```java
if (noPinknote) {
    if (noMilk) {
        buy milk;
    }
}
```

Jill

```java
if (noBluenote) {
    if (noMilk) {
        buy milk;
    }
}
```

Remove Pink note

Remove Blue note
Solution #2: Colored Notes

Jack
BlueNote = 1;
if (PinkNote == 0) {
A₁ if (milk == 0) {
milk++;
}
A₂
A₃
} 
BlueNote = 0;

Jill
PinkNote = 1;
if (BlueNote == 0) {
B₁ if (milk == 0) {
milk++;
B₂
B₃
} 
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-B₃
(PinkNote is not 1)
Since (BlueNote==1 or milk>0) is stable, then
Jill will not pass B₁

Solution #3

Jack
BlueNote = 1;
while (PinkNote == 1) {
;
}
A₁ if (milk == 0) {
milk++;
}
A₂
A₃
} 
BlueNote = 0;

Jill
PinkNote = 1;
if (BlueNote == 0) {
B₁ if (milk == 0) {
milk++;
B₂
B₃
} 
PinkNote = 0;

Proof of Safety
Similar to previous case

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

Solution #2: Colored Notes

Jack
BlueNote = 1;
if (PinkNote == 0) {
A₁ if (milk == 0) {
milk++;
}
A₂
A₃
} 
BlueNote = 0;

Jill
PinkNote = 1;
if (BlueNote == 0) {
B₁ if (milk == 0) {
milk++;
B₂
B₃
} 
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-B₃
(PinkNote is not 1)
Since (BlueNote==1 or milk>0) is stable, then
Jill will not pass B₁

Solution #3

Jack
BlueNote = 1;
while (PinkNote == 1) {
;
}
A₁ if (milk == 0) {
milk++;
}
A₂
A₃
} 
BlueNote = 0;

Jill
PinkNote = 1;
if (BlueNote == 0) {
B₁ if (milk == 0) {
milk++;
B₂
B₃
} 
PinkNote = 0;

Proof of Safety
Similar to previous case

Proof of Liveness
Jill will eventually set PinkNote = 0
(no loops)
Jack 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

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

2. Claim: If the following invariant holds when a thread enters the critical section, so does mutual exclusion:

\[
\begin{align*}
\neg & i_i \lor (i_j \land (\neg \text{turn}_i = 0)) \lor (\neg i_j) \\
\neg & (i_i \land (\neg \text{turn}_j = 0)) \land \neg (\neg i_j \lor \text{turn}_j)
\end{align*}
\]

3. How do we prove it?

Towards a solution

1. The problem then boils down to establishing the following:

2. How can we do that?

A first fix

1. Add assignment to \( i \) to establish second disjunct:

\[
\begin{align*}
& i_0 := \text{true} \\
& i_1 := \text{true} \\
& \text{ENTRY}_i := \text{false} \quad \text{while} (\text{turn} \neq i)
\end{align*}
\]

A dirty trick

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

\[
\begin{align*}
& i_0 \quad \text{while(false)} \\
& i_1 \quad \text{while(true)} \\
& \text{ENTRY}_0 := \text{false} \quad \text{while} (\text{turn} = 0) \\
& \text{ENTRY}_1 := \text{false} \quad \text{while} (\text{turn} = 1)
\end{align*}
\]
Is Peterson safe?

Thread T₀
while(!terminate)

{\(i₀\)}
\(α₀\)
while
\(i₀ \land (\neg i₁ \lor \neg \alpha₀ \lor \alpha₁)\)

If both in the critical section, then:
\(i₀ \land (\neg i₁ \lor \neg \alpha₀ \lor \alpha₁)\)

Thread T₁
while(!terminate)

{\(i₁\)}
\(α₁\)
while
\(i₁ \land (\neg i₀ \lor \alpha₀ \lor \neg \alpha₁)\)

Live: Non-blocking

while(!terminate)

\(\{R₀ : \neg i₀ \land (\alpha₁ \lor \neg \alpha₀ \lor \neg \alpha₁)\}\)
\(α₀\)

\(\{R₁ : \neg i₀ \land (\alpha₁ \lor \neg \alpha₀ \lor \neg \alpha₁)\}\)
\(α₁\)

\(\{S₁ : \neg i₁ \land (\alpha₀ \lor \alpha₁ \lor \neg \alpha₀)\}\)

Blocking Scenario: T₀ before NCS₀, T₁ stuck at while loop

Live: Deadlock-free

while(!terminate)

\(\{R₀ : \neg i₀ \land (\alpha₁ \lor \neg \alpha₀ \lor \neg \alpha₁)\}\)
\(α₀\)

\(\{R₁ : \neg i₀ \land (\alpha₁ \lor \neg \alpha₀ \lor \neg \alpha₁)\}\)
\(α₁\)

\(\{S₁ : \neg i₁ \land (\alpha₀ \lor \alpha₁ \lor \neg \alpha₀)\}\)

Blocking Scenario: T₀ and T₁ at the while loop, before entering critical section

A better way

5. 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

```c
Jack/Jill/even Dame Deb!
Kitchen::buyIfNeeded() {
    lock.acquire();
    if (milk == 0) {
        milk++;  
    }
    lock.release();
}
```

Use hardware to support atomic operations beyond load and store