Thread Synchronization: Foundations

Edsger's perspective

Testing can only prove the presence of bugs…

…not their absence!

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

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 threads are concurrently in the critical section (Safety)

Critical Section

- Mutual Exclusion: At most 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

- Mutual Exclusion: At most 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 is in its entry section, then there is a bound on the number of times that other threads are allowed to enter the critical section before 's request is granted (Safety)

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	Jill
Look in the fridge: out of milk	
Leave for store	Look in fridge: no milk
Arrive at store	Leave for store
Buy milk	Arrive at store
Arrive at home: put milk away	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
```

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

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

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

- 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!

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

- Safe? Live? Bounded waiting? Why?

Solution #1: Leave a note

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

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

- 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

```
  Jack/Jill
if (noMilk) {
   if (noNote) {
      leave Note;
      buy milk;
      remove Note
   }
}
```

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) {
```

switch

T2

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

switch

T1

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

Oh no!

Solution #1:
Leave a note

- If you find a note from your roommate don't buy!
  - Leave note ≈ lock
  - Remove note ≈ unlock
- Safe?
- This "solution" makes the problem worse!
  - sometime it works, sometime it doesn't

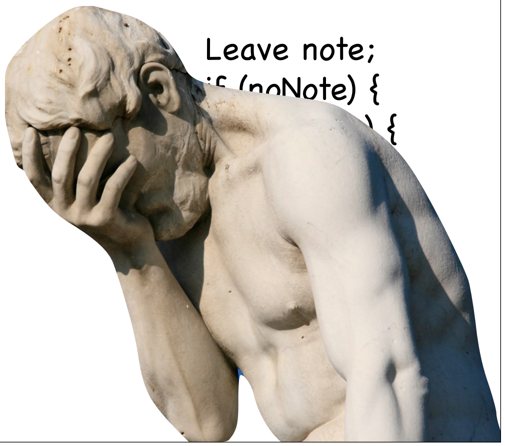
Jack/Jill

```
if (milk == 0) {
   if (note==0) {
      note = 1;
      milk++;
      note = 0;
   }
}
```

What if we leave the note first?

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

```
Leave note;
if (noNote) {
            ) {
```



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
```

Solution #2: Colored Notes

Jack

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

Jill

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

Proof of Safety

By contradiction:

Suppose Jack and Jill both buy milk

Consider state of variables (PinkNote,milk) at A_1

- 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 A_2 and never buy milk
- Case 3: PinkNote == 0, milk == 0
  - Impossible. Jill cannot be executing in B_1-B_3 (PinkNote is not 1!)
  - Since (BlueNote==1 or milk>0) is stable, then Jill will not pass B_1

Solution #2: Colored Notes

Jack

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

Jill

```
PinkNote = 1;
if (BlueNote == 0) {    B1
   if (milk == 0) {     B2
      milk++;           B3
   }                    B4
}                       B5
PinkNote = 0;
```

Proof of Liveness

Not Live!

Solution #3

Jack

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

Jill

```
PinkNote = 1;
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)

Jack will then reach line A_1

- 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:

  : id of thread allowed to enter CS under contention

  : thread is executing in CS, or trying to do so

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

How do we prove it?

in_i wants to enter CS

in_j does not desire to enter CS

in_j wants to enter CS, but it is in_i's turn

Towards a solution

- The problem then boils down to establishing the following:

- How can we do that?

A first fix

- Add assignment to to establish second disjunct

Thread T_0

while(!terminate)

$\{in_0\}$

while

$\{in_0 \wedge (\neg in_1 \vee turn = 0)\}$

Thread T_1

while(!terminate)

$\{in_1\}$

while

$\{in_1 \wedge (\neg in_0 \vee turn = 1)\}$

but these invariants do not hold!

A dirty trick

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

Thread T_0

while(!terminate)

$\{in_0\}$

α_0

while

$\{in_0 \wedge (\neg in_1 \vee turn = 0 \vee at(\alpha_1))\}$

Thread T_1

while(!terminate)

$\{in_1\}$

α_1

while

$\{in_1 \wedge (\neg in_0 \vee turn = 1 \vee at(\alpha_0))\}$

Is Peterson safe?

Thread T₀
while(!terminate)
$\{in_0\}$
α_0
while
$\{in_0 \land (\neg in_1 \lor turn = 0 \lor at(\alpha_1))\}$

Thread T₁
while(!terminate)
$\{in_1\}$
α_1
while
$\{in_1 \land (\neg in_0 \lor turn = 1 \lor at(\alpha_0))\}$

If both in the critical section, then:

$in_0 \land (\neg in_1 \lor turn = 0 \lor at(\alpha_1))$ $in_1 \land (\neg in_0 \lor turn = 1 \lor at(\alpha_0))$ $\neg at(\alpha_0)$ $\neg at(\alpha_1)$

Live: Non-blocking

while(!terminate)
$\{R_1 : \neg in_0 \land (turn = 1 \lor turn = 0)\}$
$\{R_2 : in_0 \land (turn = 1 \lor turn = 0)\}$
α_0
$\{R_2\}$
while
$\{R_3 : in_0 \land (\neg in_1 \lor turn = 0 \lor at(\alpha_1))\}$
$\{R_3\}$
$\{R_1\}$

while(!terminate)
$\{S_1 : \neg in_1 \land (turn = 1 \lor turn = 0)\}$
$\{S_2 : in_1 \land (turn = 1 \lor turn = 0)\}$
α_1
$\{S_2\}$
while
$\{S_3 : in_1 \land (\neg in_0 \lor turn = 1 \lor at(\alpha_0))\}$
$\{S_3\}$
$\{S_1\}$

Blocking Scenario: T_0 before NCS_0, T_1 stuck at while loop

Live: Deadlock-free

while(!terminate)
$\{R_1 : \neg in_0 \land (turn = 1 \lor turn = 0)\}$
$\{R_2 : in_0 \land (turn = 1 \lor turn = 0)\}$
α_0
$\{R_2\}$
while
$\{R_3 : in_0 \land (\neg in_1 \lor turn = 0 \lor at(\alpha_1))\}$
$\{R_3\}$
$\{R_1\}$

while(!terminate)
$\{S_1 : \neg in_1 \land (turn = 1 \lor turn = 0)\}$
$\{S_2 : in_1 \land (turn = 1 \lor turn = 0)\}$
α_1
$\{S_2\}$
while
$\{S_3 : in_1 \land (\neg in_0 \lor turn = 1 \lor at(\alpha_0))\}$
$\{S_3\}$
$\{S_1\}$

Blocking Scenario: T_0 and T_1 at the while loop, before entering critical section

---

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

Jack/Jill/even Dame Dob!

```
Kitchen::buyIfNeeded() {
    lock.acquire():
    if (milk == 0) {
            milk++;
        }
    lock.release();
}
```

Use hardware to support atomic operations beyond load and store