Lecture #6: Too much milk

****

Review -- 1 min

****

Independent v. cooperating threads

(1) must work with all possible interleavings
(2) not feasible to reason about all interleavings
a. mentally compile code down to assembly
b. think about every possible interleaving
c. [intuition is a poor guide…]

Atomic operations – a start

- but 3-line program still takes 200 work-minutes to analyze
- mentally disassemble code, compute all interleavings, …

****

Outline - 1 min

****

Issue: thread dilemma – want independence and cooperation
Basic correctness properties -- safety and liveness
Sample problem: too much milk
3 possible solutions

****

Preview - 1 min

****

today – ad-hoc solutions to illustrate issues
next week – more satisfactory abstractions

****

Lecture - 20 min

****

1. Motivation

Person 1 Person 2

3:00	Look in fridge; out of milk.	
3:05	Leave for store	
3:10	Arrive at store	Look in fridge; out of milk
3:15	Buy milk	Leave for store
3:20	Arrive home; put milk away.	Arrive at store
3:25		Buy milk
3:30		Arrive home; put milk away.
		Oh no!

1.1 Critical section problem

Shared state – state read or written by more than one thread

Synchronization: using atomic operations to ensure cooperation among threads accessing shared state

Lesson from last time – using "load" and "store" as our atomic operation is not tractable
→ critical section problem

Consider a collection of shared state and all code that reads or writes that shared state

Critical section – a set of code that accesses shared state
Critical section problem – ensure that all critical sections on a collection of shared state appear to execute atomically

> *i.e.,* thread A can never observe a state where thread B has partially executed a critical section

Rather than reasoning about atomic load/stores, reason about atomic critical sections
→ *fewer, coarser-grained interleavings*
→ *high-level invariants*

Solution to critical section problem must satisfy 3 rules:

1) **Mutual Exclusion:**
   roughly:"don't let more than one in at a time"
   *precisely***:** never more than one thread is executing in a critical section. One thread in CS *excludes* others.

2) **Progress:**
   roughly: "let someone in" (avoid trivial solution of let no one in)
   precisely: if no threads are executing in a critical section, and a thread wishes to enter a critical section, a thread must eventually be guaranteed to enter the critical section
3) **Bounded waiting:**
   roughly: "be fair" (fairness; avoid trivial solution of "let thread 1 in")
   precisely: if thread T wishes to enter a critical section, then there exists a bound on the number of other threads that may enter the critical section before T enters

Assumption: all threads are operating at non-zero speed (*over infinite time, each ready thread is scheduled an infinite number of times*), but you cannot make any assumption about the relative speed of the threads

The rules sound a bit strange. Basic idea is simple ("roughly" above). Specific wording of precise version is to couple the progress and bounded waiting in a way that works without making assumptions about thread behaviors, clocks, schedulers, etc.

Power tool:

safety -- the program never does anything bad
liveness -- the program eventually does something good

QUESTION: which properties above are safety and which are liveness?

Safety and liveness are key properties for reasoning about programs. Any definition of a correct program can be composed of a set of safety properties and a set of liveness properties.

Use in proofs: first prove safety, then liveness
Use in protocol design: design simple core protocol to be "safe" (regardless of scheduling, message order, etc.) Then, add constraints to ensure "liveness". (Related to "separation of mechanism from policy")

1.2 Basic idea

Programming model is restricted -- each shared variable is only accessed within a specified critical section → only one thread can read/write a shared variable at a time

1) *Entry section* **"**Lock" before entering critical section, before accessing shared data
   wait if locked
   Key idea – all synchronization involves waiting.

2) *Exit section* "unlock" when leaving, after done accessing shared data

1.3 Example: Object oriented programming model

PICTURE

- methods that access shared state are **critical sections**
- associate a **lock** with each shared object
- acquire/release the lock when entering/exiting a method that is a critical section

Warning: Many of the "classic" synchronization problems were formulated before OO programming. Many of the textbooks still present the "classic" (non-OO) answers. **Much** better to think of answers in OO format.

****

ADMIN

****

project -- challenging project; aggressive schedule
(1) "parse" is just "homework/exercise" to get ready for "thread create"; *not part of library*
(2) switch/yield (don't need (1) -- just getcontext(), setcontext()
(3) thread create (now need (1), allocate stack, stub...)
(4) thread destroy (now need zombie)

****

1.4 Too Much Milk: Solution #1

Suppose I write a program to model the too much milk problem. People act in parallel, so model each person as a thread. Model "look in fridge" and "put away milk" as reading/writing a variable in memory.

What are the correctness properties for the too much milk problem? QUESTION: what is the safety property? What is the liveness property?

- never more than one person buys
- someone buys if needed

Restriction: only use atomic load and store operations as building blocks.

Basic idea of solution #1

1) Leave a note (kind of like “lock”)
   {store "1" to location NOTE}
2) Remove node (kind of like “unlock”)
   {store "0" to location NOTE}
3) Don’t buy if note (wait)
   {load from NOTE, BEQ…}

Solution #1

```
if (milk == 0){
  if(note == 0){
    note = 1; // leave note
    milk++;   // buy milk
    note = 0; // remove note;
  }
}
```

Is this protocol safe?

Why doesn’t this work? Thread can get context switched after checking note but before leaving note.

Our "solution" makes problem worse – fails only occasionally. Makes it really hard to debug. Remember, constraint has to be satisfied, independent of what the dispatcher does – timer can go off and context switch can happen at any time.

****

Admin - 3 min

****

****

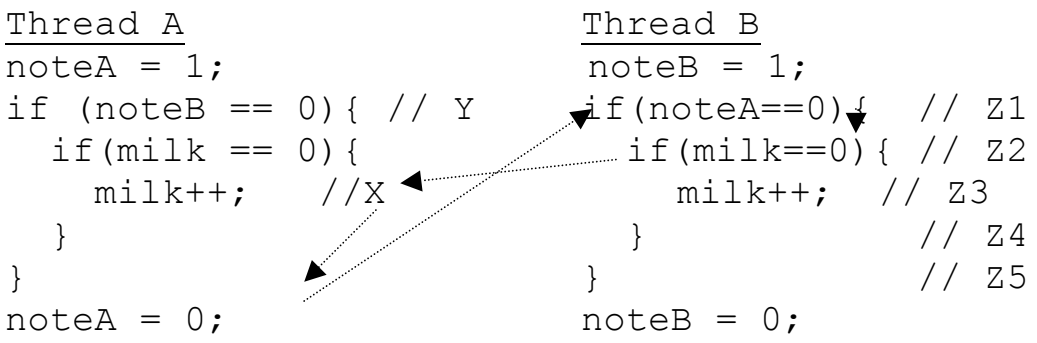
Lecture - 23 min

****

1.5 Too much milk solution #2

How about labeled notes? That way, we can leave the note before checking the milk or note.

Solution #2



Is this protocol **safe**? Proof sketch:

Lemma M: "Milk == true" is a **stable** property – once true, remains true forever

(Useful trick for reasoning about asynchronous protocols – reason about stable properties)

Part 1: A buys → B doesn't buy

Assume A reaches X

then there is no case in which B buys

when thread A was at Y consider state of "(noteB, milk)"

case 1: "(1,X)" – contradiction with assumption A reaches X (so this is not a case where B also buys)

case 1: "(0, 1)" → contradiction with assumption (by lemma M) (so this is not a case where B also buys)

case 2: "(0, 0)" → B is not in "Z" when A was at Y

if B below Z → B will not buy → safe

if B above Z

- assume B buys,
- then B read "noteA == 0"
  → "remove A" *happens before* Z1
- then B read "milk == 0"
  → Z2 *happens before* X
- .: we have removeA *happens before* Z1 *happens before* Z2 *happens before* X → removeA *happens before* X
- Contradiction → B will not buy → safe

- so this is not a case where B also buys
-

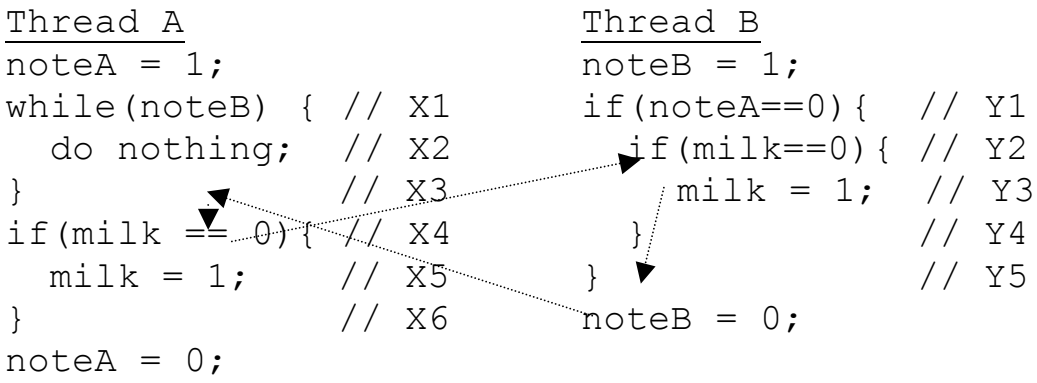
Part 2: B buys → A doesn't buy is similar

Is it **live**?
Possible for neither thread to buy milk; context switch at wrong time can lead to each thinking the other is going to buy

- Illustrates **starvation**: thread waits forever

Too much milk solution #3

Solution #3:



```
Thread A                          Thread B
noteA = 1;                        noteB = 1;
while(noteB) { // X1              if(noteA==0){  // Y1
  do nothing;  // X2                if(milk==0){ // Y2
}              // X3                  milk = 1;  // Y3
if(milk == 0){ // X4                }            // Y4
  milk = 1;    // X5              }              // Y5
}              // X6              noteB = 0;
noteA = 0;
```

QUESTION: does this work?

Yes. Can guarantee at X and Y that either
i) safe for me to buy
ii) other will buy; ok to quit

Is it **safe?**

Lemma M: (milk == true) is a *stable* property

Claim I. B buys → A doesn't buy

Suppose that B buys milk (reaches Y3), consider the instant that the load at Y1 completes,
consider states (noteA, milk) at that instant
case 1: (1,X) → contradiction (we assume B reaches Y3)
case 2: (0,1) → contradiction (lemma M; we assume B reaches Y3)
case 3: (0,0) → A is not in region X
    if below → A will not buy → safe
    otherwise, A must be above region X at that instant
        assume A buys
        → remove B *happens before* X3 (exit while loop)
        → X4 *HB* Y3 (A's check milk *HB* B's buy milk)
        Also, we know X3 *HB* X4 (program order)
        .: remove B *HB* X3 *HB* X4 *HB* Y3
        → Y3 *HB* remove B
        → contradiction (program order) → A cannot buy → safe

Claim II. A buys → B doesn't buy

Suppose that A buys milk (reaches X5), consider instant that the load at X1 completes and sees "0"

Consider state of (noteB, milk) at that instant
(1, X) → contradiction (assumed load at X1 sees "0")
(0, 1) → contradiction (lemma M; assume X5 reached)
(0, 0) → B is not in region Y
if B below region Y → B will not buy → safe
otherwise B above region Y
assume B buys (reaches Y3)
→ remove A *HB* Y1
→ Y2 *HB* X5
We also know Y1 *HB* Y2
→ removeA *HB* Y1 *HB* Y2 *HB* X5
→ remove A HB X5 → contradiction (program order)

is it **live**?
A must eventually reach "if(noMilk)"
Case 1: milk == 1 → milk bought → Live
Case 2: milk == 0 → A will buy → live

1.6 Too much milk summary

Solution #3 works, but it is really unsatisfactory:
1) Really complicated – even for this simple example, hard to convince yourself that it really works
   Every year when I teach this, I end up revising the proofs
   History is littered with published proofs of these types of algorithms followed 5 years later with published errors!
2) A's code different than B's – what if lots of threads. Code would have to be slightly different for each thread.
3) While A is waiting, it is consuming CPU time (busy-waiting)
4) doesn't work on modern hardware/compilers
   Modern HW and compilers reorder instructions. Loads/stores still atomic, but may not be executed in program order. (Can fix with "barrier" instructions, but even more complexity...)

There is a better way:
1) Have hardware provide better (higher-level) primitives than atomic load and store. Explain next lecture

2) Build even higher-level programming abstractions on this new hardware support. For example, why not use locks as an atomic building block (how we will do this in the next lecture)

    Lock::Acquire() – wait until lock is free, then grab it
    Lock::Release() – unlock, waking up a waiter, if any

These must be atomic operations – if two threads are waiting for the lock, and both see it is free, only one grabs it!

With locks, the too much milk problem is really easy!

Too much milk solution #4

```
Lock->Acquire();
if(milk == 0){
  milk++;
}
Lock->Release();
```

****

Summary - 1 min

****

Atomicity is key building block
Synchronization solutions will involve using low-level atomicity (load/store and others) to bootstrap higher-level atomicity (lock/unlock and others)

Use **safety** and **liveness** and **stable properties** to reason about programs