Lecture C3: Challenge: Independent and cooperating threads; too much milk

****
Review -- 1 min
****

multi-threaded process

User-level v. kernel threads
Pre-emptive v. non-pre-emptive threads
Thread control block
Dispatch

****
Outline - 1 min
****

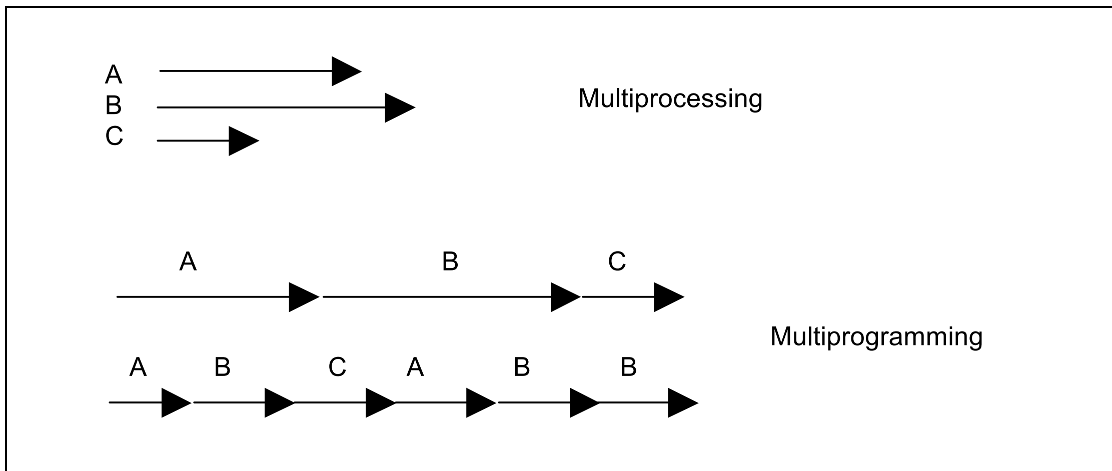
Independent v. cooperating threads
Atomic operations
Too much milk
- 3 solutions

****
Preview - 1 min
****

today – ad-hoc solutions to illustrate issues
next week – more satisfactory abstractions
Abstraction dilemma – want “independence” and “cooperation”

****
Lecture - 20 min
****

1. Multiprocessing v. Multiprogramming



Dispatcher can choose to run each thread to completion
or
time-slice in big chunks
or
time-slice so that each thread executes only one instruction at a time (simulating a multiprocessor where each CPU operates in lockstep)

If the dispatcher can do any of the above – programs must work under all cases, for all interleavings

So how can you know if your concurrent program works?
Whether **all** interleavings will work?

Option 1: enumerate and test all possibilities
Impossible!
Option 2: maintain *invariants* on program state; structure program carefully to maintain these invariants

2. Independent v. cooperating threads

2.1 Definitions

Independent threads – no shared state with other threads

- deterministic – input state determines result
- reproducible
- scheduling order doesn't matter

cooperating threads – share state

- non-deterministic
- non-reproducible

Non-reproducibility and non-determinism means that bugs can be intermittent. ***This makes debugging hard***.

3 problems
(1) combinatorial explosion in # possible interleavings
(2) program execution is non-deterministic
(3) compilers and architectures can reorder operations

2.2 Why allow cooperating threads?

People cooperate; computers model people's behavior, so at some level they have to cooperate

1. Share resources/information
   a) one computer, many users
   b) one bank balance many tellers
2. Speedup
   a) overlap I/O and computation
   b) multiprocessors – chop up program into little pieces and run them in parallel
3. Modularity
   Chop up large problem into simpler pieces

   for example – typesetting: `ref | grn | tbl | eqn | troff`

   This makes the system easier to extend; you can write eqn without changing troff

4. Fundamentally required – look at thread switch example above – different threads share ready queue, scheduling data structures, …

2.3 Some simple concurrent programs

Most of the time, threads are working on separate data, so scheduling order doesn't matter

Thread A	Thread B
x = 1;	y = 2;

What are the possible values for x:

x = 1;	x = 2;

What are the possible values for x:
initially: y = 12

x = y + 1;	y = y * 2;

What are the possible values for x:
initially x = 0

x = x + 1;	x = x + 2;

2.4 Atomic operations

atomic operation – always runs to completion or not at all; indivisible. Can't be stopped in the middle.

On most machines, memory reference and assignment (load and store) of **words** are atomic

Many instructions are not atomic. For example, on most 32-bit architectures, double precision floating point store is not atomic. It involves 2 separate memory operations.

2.5 A larger concurrent program example

Two threads, A and B, compete with each other. One tries to increment a shared counter, the other tries to decrement the counter.

For this example, assume that memory load and memory store are atomic, but incrementing and decrementing are **not** atomic

Thread A
```
I = 0;
while(I < 10){
   I = I + 1;
}
print "A wins"
```

Thread B
```
I = 0;
while(I > -10){
    I = I - 1;
}
print B wins
```

QUESTIONS
1) Who wins?
→ could be either

2) Is it guaranteed that someone wins? Why or why not?

3) What if both threads have their own CPU, running in parallel at exactly the same speed. Is it guaranteed that it goes on forever?

In fact, if they start at the same time, with A ½ an instruction ahead, B will win quickly

4) Could this happen on a uniprocessor?

Yes! Unlikely, but if you depend on it **not** happening, it will eventually happen, and your system will break and it will be very difficult to figure out why.

3. fundamental problem

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, …

fundamental problem
concurrency breaks modularity

key idea for reasoning about programs -- modularity. Structure system so that I only need to look at this code to understand what this code will do.

but with threads I need to reason about how different pieces of code interact -- I don'tjust get to step though (locally) this happens, then this happens...

Challenge -- need to restore modularity to our reasoning!

****
Lecture - 20 min
****

4. Too much milk

	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!

****
ADMIN
****

****

4.1 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
try to arrange so that only one thread is deciding/buying at a time
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.

4.2 Too much milk solution #2

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

Solution #2

```
Thread A                        Thread B
noteA = 1;                      noteB = 1;
if (noteB == 0){ // Y           if(noteA==0){   // Z1
  if(milk == 0){                  if(milk==0){  // Z2
    milk++;    //X                  milk++;   // Z3
  }                               }             // Z4
}                               }               // Z5
noteA = 0;                      noteB = 0;
```

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

Assume for the sake of contradiction that both A and B buy.

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

case 1: "(1,X)" – Impossible -- contradiction with assumption A buys milk and reaches X
case 1: "(0, 1)" → Impossible -- contradiction with assumption because milk != 0 is a **stable property (**once true, it remains true) in this simple program
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

- B is not in Z1-Z5
- noteA OR milk is a stable property
- --> B will not buy

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

4.3 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

****
Summary - 1 min
****

Thread programming – nondeterministic, irreproducible, intuition not always a good guide

   I repeat: it is **impossible** to enumerate and reason about all possible interleavings!

Key notions

Invariants – facts that must always hold true

atomic actions – the only thing you can trust

Next 2 weeks – learn how to structure program so that we can use atomic actions to build higher level programs that have invariants about which we can reason