Sept 2, 2010
Paul Hussman

Comp 409 - Concurrent Programming (Clark Verbrugge)

See Syllabus for Info → no office hours until sept 14. Class cancelled on tuesday the

This course
- lots of programming → Know Java. Can use C except for last assignment.
- no specific textbook → one main one (blue) → pretty good.
  - → Java concurrency... → good to read as well.
  - → a few other recommended texts. → see syllabus.
- can do assignments in PThreads. → there's a book on it too. Easy Read.
- In class midterm. Open book. Thursday Nov. 4, 2010

What is Concurrency vs Parallelism vs multithreading.

Parallelism ⇒ You, the user, the programmer has complete control over everything, and you give a complete specification as to what happens when and where.
- Kind of like being the coach of a sports team.

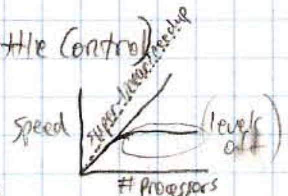
THIS COURSE { Concurrency ⇒ Things still happen at the same time, but...
- things *might* happen in parallel, but only based on initial conditions & constraints. once its set up, just start and let it go.
- Kind of like being the referee of a sports team. You set up & enforce the rules, but thats all you do. }

Multithreading ⇒ (or MultiProcessing)
- Process: includes: address space, code, data, IO handles, interrupt states (Big Package)
- Thread: Similar to a Process, but light weight. (Small Package).
  - shares part of the package of a process.
  - switches faster than Process.

- We will not cover much of distributed programming.
- We will discover few other languages than Java & PThreads. (It is assumed that we know some C too).
- What is a Thread?
  - An independent flow of control WITHIN a process, composed of a context (registers), and a sequence of instructions to execute.
- Inside a Thread

O.S.	USER
→ thread I.D. (very important) → scheduling policy (how it should be scheduled) → priority → signal mask	→ registers (includes P.C.) (local data) → stack (local data)
→ everything else is shared! (heap, static data, files handles)	

- O.S. May or may not schedule your threads at any particular point in time. (Little Control)
  - → we must program so our code works anyway!
- What are Threads good for?
  - speed things up (do half & half at once to half the time).



Amdahl's Law

total time = sequential part ("s") + parallel part ("p")

→ we have n threads

$$\text{(total time)} = \text{seq-part} + \frac{\text{parallel part}}{n}.$$

"s" (speedup) = $\frac{\text{old time}}{\text{new time}}$.

let total time be $1 \Rightarrow s + p = 1$. Then $s = 1 - p$

$$\frac{1}{(1-p) + \frac{p}{n}} = \text{actual speedup.}$$ (1 = old time)

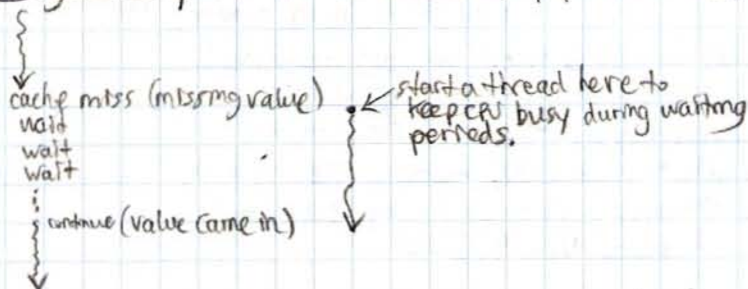
- assume program is 75% parallizable

then speedup $= \frac{1}{0.25 + \frac{0.75}{n}}$ (disregarding overhead)

n	speedup
1	1
3	2
15	~3.8
∞	4

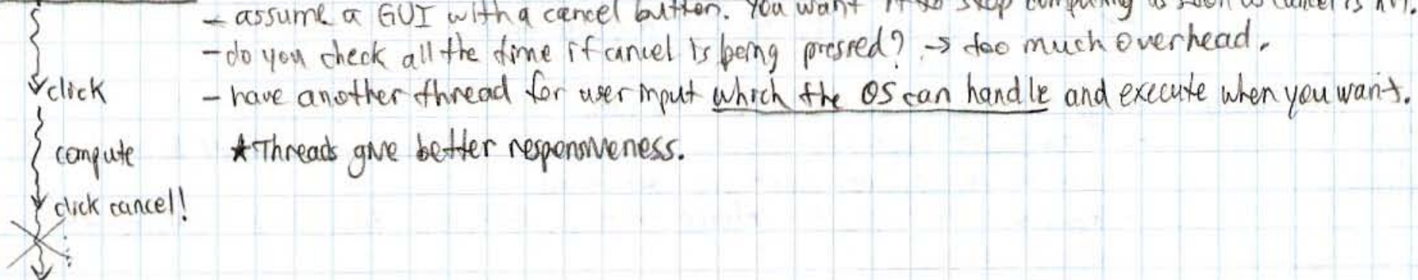
(limit to speedup)

- What else can Threads help with?
  - Hiding Latency. at cache misses, pipeline stalls, I/O, etc.



  - It is important to have a FAST context switch (ie LightWeight Threads).

  - Responsiveness



    - assume a GUI with a cancel button. You want it to stop computing as soon as cancel is hit.
    - do you check all the time if cancel is being pressed? → too much overhead.
    - have another thread for user input which the OS can handle and execute when you want.

    ★ Threads give better responsiveness.

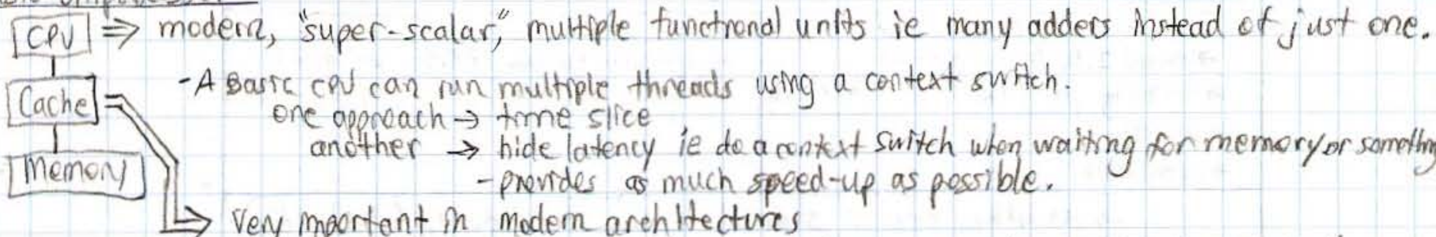
  - Appropriate Model (sometimes it makes more sense this way).
    → web server

- What are Threads NOT good for?
  - Some overhead exists for context switching etc.
  - Debugging is DIFFICULT! ⇒ get it right the first time & keep it simple!

| SEPT 9, 2010 | (CHRIS DRAGERT)

Hardware Basics → what kinds of machines support concurrency?
- new processors have more than one core → multicore.

Basic Uniprocessor



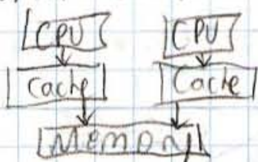
CPU ⇒ modern, "super-scalar," multiple functional units ie many adders instead of just one.

- A basic CPU can run multiple threads using a context switch.
  - one approach → time slice
  - another → hide latency ie do a context switch when waiting for memory/or something
    - provides as much speed-up as possible.

Very important in modern architectures
Why? ⇒ LOCALITY ie entire chunks of memory can be cached and sequentially accessed which is often very useful. You often might access the same instructions over & over ⇒ like a loop.

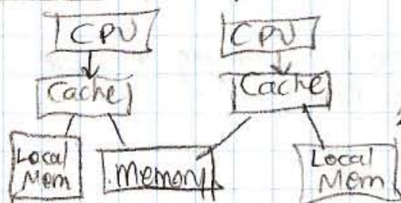
- Cache w/ Multiple Threading ⇒ lots of cache misses after context switch because processes get in the way of each others' locality
  - this isn't devastating but can be a concern.

Multiprocessor (several Types)
- UMA (Uniform Memory Access)



- no cache misses from locality, but there's still a cost to access memory

NUMA - (Non-Uniform Memory Access)



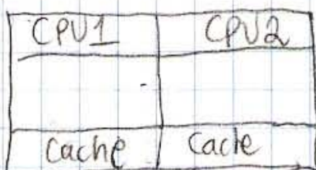
different Kinds of memory with different Access Costs.
- Numa Distributes Memory (ie distributed systems).

For this class we focus on UMA Architectures (most common these days)
- in practice, this distinction isn't hugely important.
there's multiple cache levels which make this distinction less useful.

SMP - (Symmetric Multiprocessor)
- All CPU's have the same status. Can have cache but not necessarily.

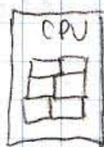
- Each CPU can be a seperate chip or we can put multiple CPUs on the same core.

CMP (On-Chip Multiprocessor)



- more efficient if you have multiple threads
- less efficient if you only have single threads.

CMT (Coarse-Grain Multithreading) (⇒ NOT USED MUCH ANYMORE)
→ less of a Multiprocessor.
→ Has hardware support for context switches!

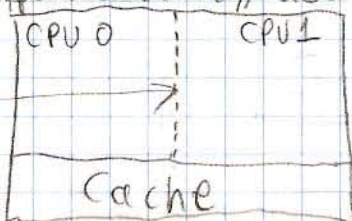


- CPU has slots that store register information for preventing memory access on each context switch.
- n-way CMT can support n threads in hardware ⇒ Fast context switch!

- Hardware can switch threads every so often like during a stall or every X cycles.
- No slow-down to run a single thread
- Basically a large Uniprocessor with hardware context switches.

FMT (Fine-Grain Multithreading)
- like a CMT, but switch every cycle.
A.K.A. Barrel Processing. ⇒ Rotate the barrel and one section executes per clock cycle, then it rotates to the next.
→ Deterministic Switching is Easier to Measure which is good.
→ For only one thread, you only get 1/n of the performance which sucks.
example ⇒ Tera-Cray has 70 slots ⇒ up to 70 cycles per operation. No data cache
- overall throughput is very good if there's lots of threads.

SMT Simultaneous MultiThreading a.k.a. HYPERTHREADING [Eggers et al 94/95]
- cross between CMP and CMT
- support concurrency, also single threading



→ Resources of each CPU are available to both CPUS.
So Threads can make use of all available functional units.
→ Running a single Threads lets it act like One Big CPU.
→ Running multiple threads lets it share resources in a flexible manner.
→ Why Better? Super-Scalar Architecture (multiple functional units like ALU's etc)

Functional Units of CPU

- ALU Arithmetic Logic Unit
- FPU Floating Point Unit
- LSU Load/Store Unit
- BPU Branch Prediction Unit
- (etc)

} Hyperthreading takes advantage of having multiple Functional Units on one CPU.

→ more than one instruction can be issued per cycle.

example 1: add R1, R2, R3 — ALU
2: add f1, f2, f3 — FPU
3: test r3, 0 — ALU

- this would take 3 cycles on basic cpu.
- with hyperthreading it takes 2 cycles.

↳

CPU0	CPU1
1	2
3	

Consider this program:

T1	T2	T3	T4
add	fadd	br	br
add	fadd		
add	fadd		
load	load		
add	add		

(dependency: load → add)

assume:
- can issue 3 instructions per cycle.
- assume we have 2 of each of the above functional units.
- load takes 2 cycles
- br takes 3 cycles
- add/fadd takes only one cycle.

With a Basic Uniprocessor:

	Issue Slot		
cycle	0	1	2
1 (T1)	T1 add	T1 add	—
2 (T1)	T1 add	T1 load	—
3 (T1)	—	—	—
4 (T1)	T1 add	—	—
: (T2)	T2 fadd	T2 fadd	—
: (T2)	T2 fadd	T2 load	—
: (T2)	—	—	—
(T2)	T2 fadd	—	—
(T3)	br	—	—
(T3)	—	—	—
(T3)	—	—	—
(T4)	br	—	—
(T4)	—	—	—
14 (T4)	—	—	—

} horizontal waste!
→ Bad → happens from dependencies or pipeline stalls.
→ Basic UP doesnt have a way around this.

↑ Vertical Waste!
→ This issue slot is never used because there's not enough functional units available in the cpu to do another instruction per cycle.

With other CPU Types

SMP:

P1			P2		
add	add	—	fadd	fadd	—
add	load	—	fadd	load	—
—	—	—	—	—	—
add	—	—	fadd	—	—
br	—	—	br		
—	—	—			
—	—	—			

(7 cycles)

CMT:

add	add	—
add	load	—
fadd	fadd	—
fadd	load	—
br	—	—
br	—	—
add	—	—
fadd	—	—

(8 cycles)

COMP 409
Sept 9 CONT'D

PAUL HUSSMAN

FMT

add	add	—
fadd	fadd	—
br	—	—
br	—	—
add	load	—
fadd	load	—
add	—	—
fadd	—	—

(8 cycles)

SMT (Hyperthreading)

add	add	add
load	fadd	fadd
fadd	fload	br
add	—	br
fadd	—	—
—	—	—

(6 cycles)

(Note: What if branches were earlier? It would be fewer cycles.)

Notice ⇒ Horizontal Waste → IPC/CPI (instructions per cycle / cycles per instruction)
- → We rarely get more than 2-3 IPC in practice
  wide issue → not really effective in practice.
- → Hyperthreading promises 6 CPI in theory! but not always the case.
- → can be useful in vector operations
- extracting / finding the concurrency at a low-level is hard.
- Hyperthreading is not quite as good as 2 CPUs in practice because there tends to be code conflicts
  - → threads fight over the cache
  - → you only get 1/2 the cache effectively
  - → instruction/thread mix.

Sept 14, 2010

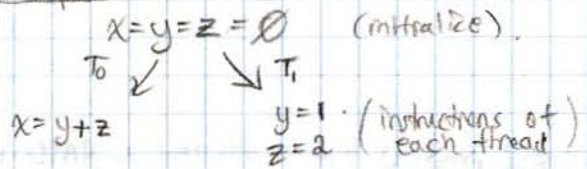
PAUL HUSSMAN

Last time → basic hardware designs → UMA, NUMA, SMP, CMP, CMT, FMT, SMT ← common these days.

Today: Atomicity → thread interactions → this is where things get interesting.
vs. embarrassingly parallel.

Definition: Thread Interaction
- The read set of a thread: set of all variables (data, mem locations etc) that a thread reads (not writes)
- The write set of a thread: set of all variables that a thread writes.
- Two threads are INDEPENDENT if the write set of each is disjoint from the read and write sets of the other thread.
  - → a R/W or W/W of the same variable implies NOT independent threads.
  - → R/R

Example



} what is the final value of x?
→ depends on order of execution in an absolute timeline.

3 Possible orders of execution: (interleavings)

	or:		or:	
T0 : x=y+z T1 : y=1 z=2		T1 : y=1 z=2 T0 : x=y+z		T1: y=1 T0: x=y+z T1 : z=2
x==0		x==3		x==1

Atomic statements are ones that can't be divided up smaller.
In concurrency we try to consider all interleavings of atomic operations.
eg: x=y+z is probably not atomic when considering the machine language:
In this case there are more possible interleavings!

```
load r1, [y]
load r2, [z]
add r1, r1, r2
store r1, [x]
```

More possibilities of interleaving:

```
T0: load r1, [y]        y=0
T1: y = 1
    z = 2
T0  load r2, [z]        z=2
    add r1, r2, r2
    store r1, [x]
```

} this possibility wasn't obvious from the actual code.

x == 2

- so we need to know what's atomic and what's not in order to program concurrently properly.
- we have to be very careful to do things correctly,
- If something is not atomic, we need to know how to break them down into atomic values.

Atomic operations

y = 1; (variable assignments of word size (32-bit value on a 32-bit machine) ⇒ usually atomic
variable assignment of long size (64 bit # in Java or long long in C) on a 32-bit machine requires a sequence of steps! (non atomic)

long x; x = 0; // x upper = 0, x lower = 0
x = -1; // x upper = ... lower = ...
} what happens when? Important to consider.

So: In Java, a 64-bit assignment is NOT necessarily atomic!
UNLESS: You declare them volatile.

volatile long x;
x = 0;
x = -1;
} this will happen as expected.

x = 1; likely atomic
x = y; ? not necessarily atomic
x = y + z; not likely atomic (probably not)

- sometimes you can use non-atomic operations, but assume that they are effectively atomic.
  eg: $x = y * z + q/w$
  - if all of y, z, q and w are all local (unseen by other threads), then this operation is effectively atomic. we need some way to prevent other threads from looking at our local operations.

Consider a statement: "x = e" where e is some expression.

Definitions

- e has a CRITICAL REFERENCE: if e uses a variable, that another thread writes.
- "x = e" satisfies the AT-MOST-ONCE property (AMO), ie: appears atomic, if at least 1 of 2 conditions are satisfied:
  1) e has exactly ONE critical reference (variable that another thread writes into) and x is NOT read by another thread.
  2) e has NO critical references in which case x may be read.

ex: x = y = 0
① x = y+1 ② y = y+1

① x = y+1 } AMO ✓ (x: not read, y: CR)

② y = y+1 } NO CR, another thread reads y but it's allowed ∴ AMO ✓

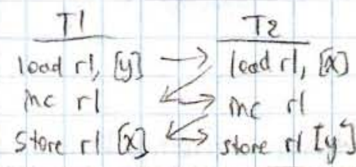
So now we only have to worry about what comes first between ① and ②, because they're effectively atomic.

ex: x = y = 0
Ⓐ x = y+1 Ⓑ y = x+1

Ⓐ NOT AMO!
Ⓑ NOT AMO!
} not effectively atomic.

If Ⓐ is first, x=2, y=1. if Ⓑ is first x=1, y=2. but more possibilities arise from interleavings of the machine code since they're not atomic!

T1	T2
load r1, [y]	load r1, [x]
inc r1	inc r1
store r1 [x]	store r1 [y]



x=1, y=1
different answer!

COMP 409
SEPT 14 CONT'D

PAUL HUSSMAN

→ why bother?
→ how many interleavings?
with n threads, each of which does m atomic operations, then
the # of interleavings is equal to: $\frac{(n \cdot m)!}{(m!)^n}$ ⇒ Grows FAST!

- so we can't consider all interleavings of big programs, so we try to make as large atomic pieces as possible.
(...to be continued...)

JAVA and PThreads

Java Threading

Threads are objects. You can make new thread objects. You can extend threads to make your own version of threads ie myThread.
Or the preferred way of doing it is to implement Runnable
eg:
```
class Foo implements Runnable {
    void run() {
        :
    }
}
new Thread (new Foo()).start(); // created & started.
```

- Inside of start, it looks for a Runnable, creates a new Thread, then calls run on it.

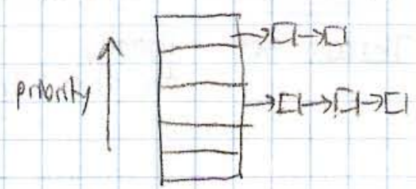
The other (less perferred way)
```
class MyThread extends Thread () {
    void run () {
    }
}
new MyThread().start();
```

- Both ways work.
- There are daemon threads and non-daemon threads. (→ default: non-daemon)
  - daemons are for run-time services. Things you want executing while the program is executing. (ready to work when needed)
  - non-daemons are default threads. (do all the work).
  - your code/program terminates when all non-daemon threads are finished.

- other methods of interest
  Thread class
  Thread.currentThread() // gets current thread. ⇒ used fairly often.
  isAlive(); // is this thread still alive? (not very useful. more like "wasAlive").
  enumerateThreads(); // by the time it returns, it's stale information ⇒ less useful.
  sleep(); // tells a thread to sleep for specified amnt of time.
    Time-out value is not guaranteed to be accurate.
    OS might still take a while after the time-out runs out to get back to it. ∴ time-out is a lower bound.
    Also granularity of timer isn't always accurate. High-res timers aren't really the norm.

SEPT 16 2010

Last time: Atomicity & Independence, At most Once, Java Threads, Runnable, daemon vs non-daemon, Thread Scheduling
Java → nominally priority pre-emptive scheduling model
  → time sliced within a priority class
  → higher priority ALWAYS runs in preference to lower priority.



- shouldn't really change thread priorities from the default because they could prevent other threads from being executed.
- normally there aren't any guarantees → priorities may not match what you think they should. (conceptualization)
  → preemptive is optional → "green threads", "run to completion" Threads run until completion unless they do a Yield() call to give up the CPU to some one else if anyone's there.
- Atomicity → 32-bit read/writes are atomic
  → 64-bit read/writes are NOT atomic unless declared volatile.

The 'volatile' keyword

- volatile means that the variable may change "asynchronously".
  → it's a way to tell the compiler that a variable can change at any time.

```
x = someclass.var;
  :        } → x doesn't change, someclass doesn't change, var doesn't change.
y = someclass.var;  // compiler optimization ⇒ y = x;
    \_____________/
          x
```

★ In concurrent code, the compiler doesn't know if maybe some other thread messes with something in that code, so it can't really do that compiler optimization, or it could sometimes be wrong.

```
static int x;
   ↙          ↘
println("waiting");     x = 1;
while (x==0);   // compiler would optimize this by removing the condition because compilers
println("off I go...");   are optimized for sequential code.
```

```
volatile static int x;  // compiler checks every time because you told it to be more careful.
```

- This is what VOLATILE means in Java and C, but in Java there's more meaning to it.

Basic Synchronization

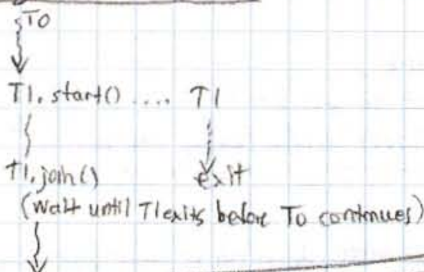
→ Every object has a lock associated with it. A lock can be owned by only one thread at a time.
If T1 is locked to an object, no other threads can access it until it is unlocked. They wait.

Java:
```
synchronized (obj) { lock(obj);
   :
} unlock (obj);
```

you can declare methods as synchronized. (2 ways)

```
synchronized int foo() {        ≡     int foo() {
    :                                     synchronized(this) {
}                                            =
                                          }
                                      }
```

The join() method



There is also:
Thread.stop(); ← doesn't work properly!
Thread.destroy(); ← really hard to get right ⇒ gave up.
both are deprecated

Termination → Usually when run() finishes, the thread finishes.

COMP 409
SEPT 16, CONT'D

(PAUL HUSSMAN)

PThreads

→ POSIX Standard 1003.1C is PThreads
→ It's basically a library that provides threading & synchronization
→ there's also 1003.1b → realtime extentions.
→ for the c compiler you do -lpthread to link to the pthread library.
  → also need to link to the C Rentrant realtime libraries. (thread safe).
    eg: strtok('') called repeatedly with null to keep getting bits of the string.
      if this is called by more than one thread at a time, there could be a problem
  → nowadays -pThread takes care of the re-entrant cases.
→ creating a thread.
  pthread_create(handle, attributes, startRoutine, args); // creates new thread

Handle:
  - handle is a pointer (identity of your thread), similar to "thread object" in Java.
  - handles might be structs, so you can't just use '=' to compare them.
    - instead use pthread_equal(,);
→ pthread_self(); // returns your handle.
→ exiting via pthread_exit(); // optional way of exiting a current thread. You can pass retVals.

Attributes: "type" → scheduling params, priority, stacksize, etc.
  - allocate an attribute structure
  - there's a routine to fill out the defaults.
  - change the ones you want
  - pass to the pthread_create function
  - note) you can re-use attributes.
  - don't forget to free the memory of it later.
  - you can just pass null as the attribute and it takes a default.

The detached state
  → joinable thread (default threads) → You ~~can~~ MUST call (pthread_join()) to join it. (but ONCE only!) [gives a return value.]
  → detached thread → doesn't leave space for a return value
    → do NOT call pthread_join().

- There are lots of options and flexibility in PThreads.
  → you can check what is available at compile time or run-time.
    eg: _POSIX_THREAD_ATTR_STACKSIZE
  compile: #ifdef... // check if it's there
  runtime: sysconf(...) can pass _SC_THREAD_ATTR_STACKSIZE

Scheduling
  → SCHED_RR → Round Robin Time slice like Java's priority pre-emptive model.
  → SCHED_FIFO → First In First Out like "green threads" → run to completion
  } check if exists before using it.
  → SCHED_OTHER → Whatever the OS does.
    there are priorities but don't mess with them

-Contention Scope
  → what level/scope do your threads compete at
  → either within your process PTHREAD_SCOPE_PROCESS
  → or globally PTHREAD_SCOPE_SYSTEM
Note: these do not always mix well with other attributes ⇒ may need superuser privilages.

→ Fork
  → copies/duplicates a process, but doesn't copy threads
  → with a multithreaded program, only the calling thread is duplicated (other threads aren't copied).

→ Locking
  → pthread_mutex_lock() or pthread_mutex_unlock().
  → also attribute based.
  ⇒ only the thread that locks can unlock, and no recursive locking.

- you can describe locks as NORMAL (as described)
RECURSIVE (more like Java)
ERROR CHECK (gives an error if you do something wrong).

Sept 21, 2010 | PAUL HUSSMAN

Last Time → pthreads in C & Java. Pthreads is more error prone ⇒ be Very Careful!

We discussed ATOMICITY already.

→ we look for the concept of Mutual Exclusion. Only one thread at a time can access any particular chunk of code. The chunk of code is called a CRITICAL SECTION.

How do we do this? We have various techniques.

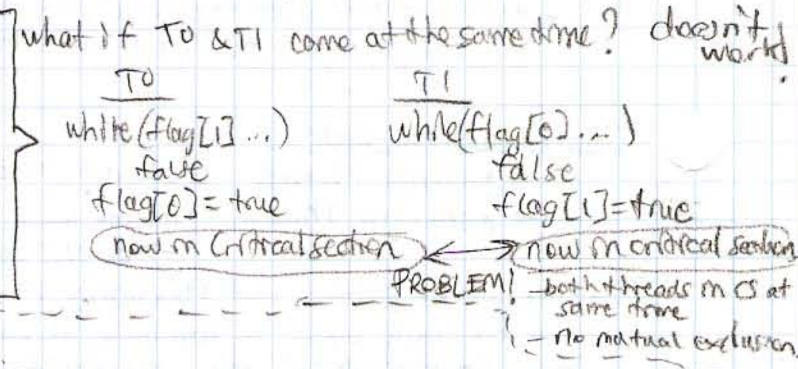
Basic Technique: Put entry & exit sections around critical sections using a busy-wait model (aka spinning)

→ 2-thread Solution

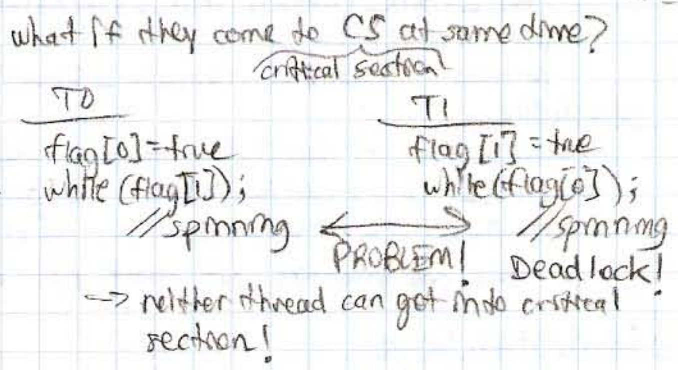
1)
```
turn = 0;   //initialize
// assume thread i.d.'s are 0 and 1.
entry (id) {
    while (turn != id);   // loop with no body
}
exit() {
    turn = 1-turn;
}
```

→ it works!
→ gives mutual exclusion.
→ BUT: enforces strict alternation beginning with thread zero.
∴ sub-optimal.
if thread one shows up first, too bad → it must wait for thread 0.

2) Initialization: flag[0] = flag[1] = false;
- flags indicate if something is actually in the critical section.
```
entry (id) {
    while (flag[1-id]);  //spin
        flag [id] = true;
}
exit (id) {
    flag [id] = false;
}
```



3) flags declare our INTENTION. (similar)
```
flag[0] = flag[1] = false;  //initialize
entry (id) {
    flag[id] = true;
    while (flag[1-id]);  //spin
}
exit (id) {
    flag(id) = false;
}
```



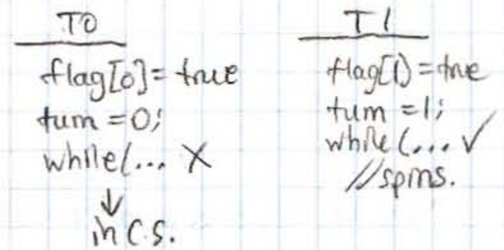
4) add some randomness
```
flag[0] = flag[1] = false;  //initialize
enter (id) {
    flag[id] = true;
    while (flag [1-id] {
        flag[id] = false;
        → random delay ←  // breaks symmetry of previous scenario
        flag[id] = true
    }
exit (id) {
    flag[id] = false;
}
```

→ this really does work (eventually).
→ enforces mutual exclusion
→ prevents deadlock
→ problem → Random delay affects efficiency!

PAUL NUSSMAN

5) This one works nicely...
Dekker's Algorithm (similar to Peterson's 2-process Tie-breaker) [1981]

```
flag[0] = flag[1] = false;   } // initialization.
turn = 0;   // tie-breaker   }
entry (id) {
    flag[id] = true;
    turn = id;
    while (turn == id && flag[1-id]); // spin
}
exit (id) {
    flag(id) = false;
}
```



Note: assigning an id to turn is an atomic operation, so it's not a problem.
If the turns get switched it still works but the other one goes first.

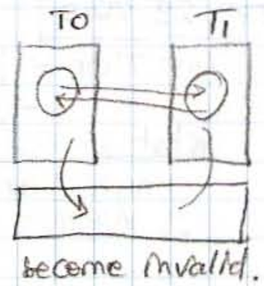
[The LAST thread to set turn goes first.]

---

n.b.: Some syntax (await())

while(turn == id && flag[1-id]); // spin.
↓ can be written as:
await(turn != id || !flag[1-id]); // await the inverse condition.

ie: while(B) ⟺ await(!B)



→ in terms of efficiency, if 2 threads access the same (shared) data, writes from one must be made visible to the other.
- CPU's communicate with each other with special hardware.
- flags get set telling the other cpu to ignore certain cache lines if they become invalid.

→ accessing shared data is expensive!
- → we should minimize shared data.
- → we should also minimize WRITES to shared data!
- → there's a nice algorithm for this: Kessel's Algorithm;

Kessel's Algorithm [1982]
- → minimize the writes to shared data. (slightly more efficient).
- → uses 4 shared 'registers': → multiple writers, readers, single writer, single reader, or combo.
- → uses 4 single writer registers. (only one thread can write into each).

6) flag[0] = flag[1] = false;
turn[0];
turn[1];

compare the 2 halves of "turn".

[So Peterson's turn==0 ⟺ Kessel's turn[0] == turn[1]
& Peterson's turn==1 ⟺ Kessel's turn[0] != turn[1]]

→ need 2 other UNSHARED variables
local[0];
local[1];

```
enter (0) {   // just for T0.
    flag[0] = true;
    local[0] = turn[1];
    turn[0] = local[0]
    await(flag[1] == false || local[0] != turn[1]);
}
```

```
enter(1) {
    flag[1] = true;
    local[1] = !turn[0];
    turn[1] = local[1];
    await(flag[0] == false || local[1] == turn[0]);
}
```

```
exit(id) {
    flag[id] = false;
}
```

so this time: →

T0	T1
flag[0] = true	flag[1] = true
local[0] = turn[1] // false	local[1] = ! turn[0] // true
turn[0] = local[0] // false	turn[1] = local[1] // true
await(flag[1] == false ✓	await(flag[0] == false ✗
	|| local[1] == turn[0]) ✗ // SPIN
in C.S.	

What if they show up at the same time?

TRUE {

T0	T1
flag[0] = true;	flag[1] = true;
local[0] = turn[1] // false	local[1] = !turn[0] // true
turn[0] = local[0] // false	turn[1] = local[1] // true
await (flag[1] == false ✗	await (flag[0] == false ✗
|| local[0] != turn[1]) ✓ (false, true)	|| local[1] == turn[0]) ✗ (true, false) // SPIN
in CS	

COMP 409
SEPT 23, 2010

PAUL HUSSMAN.

Last Time → basic mutual exclusion → turn-based, alternation-based methods, Peterson, Kessel,
→ 2-process solutions. What about n-process solutions? → look up algorithms if interested.

What are the properties we want?

1) mutual exclusion → control over critical section access.
2) Absence of deadlock → if 2 or more threads are trying to get into the C.S. & nothing is in the C.S. then only one should succeed.
3) No unnecessary delay → threads should not have to wait to enter the C.S. if no one is in there.
4) Eventual entry → if many threads are trying to get through a C.S., all of them should eventually get in (assuming all threads that enter do eventually exit.

- Properties 1, 2 & 3 are known as SAFETY PROPERTIES. → guarantee that "nothing bad happens". We could usually prove these properties → show invariants $!CS_1 \,||\, !CS_2$ or look at all interleavings.
- Property 4 is a LIVENESS PROPERTY → guarantee that "something good happens eventually". This depends on the scheduler → some algorithms help, but HARD TO PROVE.

Why is it hard to prove?

Eg: boolean go = false;

T0	T1
while (!go) { //spin	go = true;
killTime();	someCalculation();
}	
doSomethingUseful();	

→ single CPU system
→ start running Thread 0
→ keep CPU busy, so take time spinning
→ CPU is at 100% even if you never schedule Thread 1.
→ need FAIRNESS for this to actually work.

Fairness Properties (background knowledge).

1) Unconditional Fairness → A scheduling policy is unconditionally fair, if every unconditional atomic action is eventually executed.

eg:

T1	T2
while (!listReady); //spin	listReady = false;
item.next = head(list);	do some list operations;
head(list) = item;	listReady = true;

PAUL HUSMAN

2) Weak Fairness → A scheduling policy is weakly fair if
   1. Unconditionally Fair
   2. Every conditional atomic action that is eligible is eventually executed assuming the condition on which it depends becomes true and remains true forever (until our condition action is executed).

   - In previous example it is simple to see that it would work

3) Strong Fairness → A scheduling policy is strongly fair if
   1. Unconditionally Fair
   2. every conditional action that is eligible is executed eventually assuming the condition on which it depends is true infinitely often. // assume infinite program.

   → In Practice you (almost) never get strong scheduling guarantees, so CODE DEFENSIVELY!

   → In general, people will assume a strongly fair scheduling system. It's only true statistically speaking, so there's no 100% guarantee, but usually true over time.

   → If you're writing code for airplane systems or something, you need 100% guarantee.

Algorithms with some fairness.

Ticket Algorithm → take a ticket as you walk into a store.
- → tickets have numbers
- → wait for your number to be called.
- → then you're in the CS until you're done,
- → then they call the next number.

"make this part atomic" → `<turn[id] = number; number += 1;>`

```
enter(id){
   <turn[id] = number;
    number += 1;   >
    await(turn[id] == next);
}  // now in C.S.

exit(id){
    next += 1;
}
```

→ needs some atomicity & mutual exclusion.

this part is not atomic, so there must be a way around it...
eg: FA(number, 1); instead of number++;

(next += 1;) does this need to be atomic? NO since only one thread does this at once.

Fixed algorithm with low-level algorithms eg: Petersons Alg.
- → Hardware support also helps.
- → Fetch-and-add is an instruction that adds atomically.
- x = FA(y,1) → returns value of y and adds a 1 to it at the same time, atomically!

Bakery Algorithm → variation of the ticket algorithm.
→ when you walk into the store, your priority gets set to one more than everyone else in the store, by counting the people ahead of you.

```
enter(id){
  <turn(id) = max(turn[i])+1;>     // Atomicity problem. → (hard to solve simply)
    for(j=1 to N){
        if (j == id) continue;
    await(turn[j]==0 || turn[id] < turn[j]);
}
```

Hardware Support
→ Test-and-set TS(x, v) (x: variable, v: value) atomically does ; { temp = x; x = v; return temp; }
- this can help with mutual exclusions.

eg: int lock = 0;
in order to lock:
entry: while (TS(lock, 1) == 1); //spin.
exit: lock = 0; // relies on atomicity of 32-bit assignment ⇒ no problem.

→ Test-&-Test-&-Set (software method using test & set).
→ Test & set involves a lot of writing to a shared variable (lock). So above method is inefficient.
→ This is a CHEAPER way of doing Test&Set.
→ uses a read spin with a normal read, and only if it is ok, do you use test & set to write.

```
entry: do {
          while (lock == 1); // spin (repetative normal read of a shared variable).
       } while (TS (lock, 1 == 1));
```

→ Fetch & Add.
→ older (old Power PC CPUS)
→ not used much anymore

→ Compare & Swap.
→ CAS(x, v_1, v_2) ATOMICALLY sets $x = v_2$ if x is already equal to v_1 and return old value no matter what.
eg: CAS(x, v, x+5);
→ on intel there are for sure 32/64 bit versions. maybe even 8/16-bits.
→ there's a nasty problem under the covers.