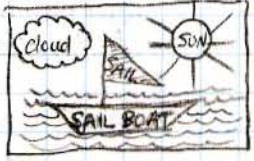
→ divergence

eg: `while(true);` // infinite computation.

→ looks stuck, but it's not waiting for anything/anyone.

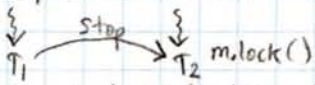
→ FLAW in a sequential program

⇒ could be valid in a concurrent program → but don't use this! wastes CPU for no reason.



THREAD TERMINATION.

→ Safety issues exist with ASYNCHRONOUS termination.



→ we don't really know the state of T2

→ what locks it holds?

→ what function/private state is in?

eg: In early versions of Java, 64-bit integer operations were done by function calls.

→ In general, it's too dangerous since you can't be too sure exactly what's going on.

→ instead set a flag,

eg: pThreads → Cancellation (polling a flag).

→ POSIX lets you call stop which sets a flag. SYSCALLS check the flag.

→ this is called CANCELLATION.

→ when you cancel a thread it sets the flag

→ flag is checked frequently, How frequently? ⇒ Nearly every BLOCKING syscall.

- Error interrupt code EINTR ⇒ every syscall that could return this checks the flag.
- Doesn't check on mutex_locks. (purely for performance) but there's an API to check yourself.

COMP 409
OCT 21, 2010

PAUL HUSSMAN

LAST TIME ⇒ finished looking at most of the core synchronization primitives

⇒ Deadlock & Termination ⇒ Coffman's Conditions → Must satisfy all 4 properties for resource deadlock. Avoid deadlock ⇒ other recovery

⇒ divergence as a possibility.

⇒ Termination → dangerous to stop/destroy for subtle reasons

⇒ Cancellation → polling method in pThreads.

Cancellation Cont'd

`pthread_cancel(&t);`

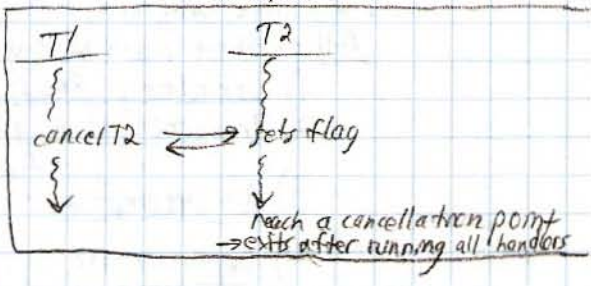
→ sets the flag TRUE

→ flag will be checked in various places, but there's also a way to check at a specific time.

→ CANCELLATION HANDLERS.

→ it's a stack → you can push/pop handlers.

⇒ handlers get run on exit.



Eg: `m.lock();`

→ push new handler to unlock m if it gets cancelled.

⋮

→ pop handler

`m.unlock();`

→ Very much like atexit, except for Threads (from C.)

→ In pThreads, you can turn cancellation on or off at any point, WITHOUT affecting flag states.

→ PTHREAD_CANCEL_DISABLE, PTHREAD_CANCEL_ENABLE

→ Cancellation is not reccomended, but if you want → PTHREAD_CANCEL_ASYNCHRONOUS
literally kills the thread even if it's in the middle of something.
by default → PTHREAD_CANCEL_DEFERRED : safely terminates a thread.

JAVA → doesn't have cancellation, but DOES have interrupts.

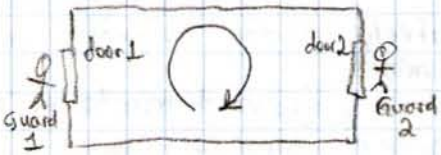
→ try { ... } catch (InterruptedException ie) { __ }

↑ eg: Thread.sleep();

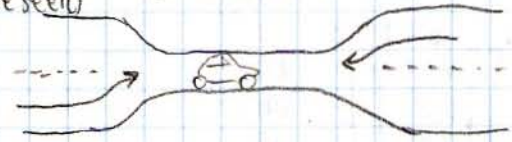
Thread.interrupt sets a flag in the target thread which gets periodically checked, and if it's interrupted, it puts you into the catch clause ⇒ do whatever you want.

NOTE: NOT EVERY BLOCKING CALL CAN BE INTERRUPTED/CANCELLED
→ there's no guarantee with this method that it stops right away (or at all).
→ InterruptedIOException

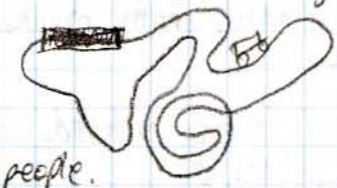
Concurrency Problems (the classic ones).
→ Producer Consumer (as we've seen) → used very often eg: buffers etc.
→ Dining Philosophers (as we've seen) → deadlocks
→ Oriental Gardens (assignment 1).
→ great-big garden



→ Readers & Writers (as we've seen)
→ One-lane bridge
- similar to reader/writer
- can allow multiple east-bound at once, and can have multiple west-bound at once. → "Readers & Other Readers"



→ Unisex bathroom
→ one door, can't have men + women together
→ fairness → don't want to keep people waiting too long.
→ Rollercoaster Problem.
→ let threads be the people riding the rollercoaster.



→ say car holds 4 people.
→ wait for 4 people to sit
→ only then can it START
→ once moving no rider does anything ⇒ threads SLEEP
→ at the end, they must be woken up
→ wait for them all to leave
→ then repeat.

→ Cigarette smoker's Problem
→ 3 smokers sitting around some sort of table.
→ they sit there making or smoking cigarettes.
→ ingredients: tobacco, papers, match
→ each smoker only has ONE of the 3 ingredients (but an infinite supply).
→ a mysterious agent deposits TWO ingredients on the table.
→ originally this was proposed as something you needed condition synchronization for (to avoid an inherent deadlock situation), but this was not true.

→ Sleeping Barber (most complex).
→ Multiple client-server interactions
→ considers rendezvous situations → eventually two threads end up at the same place to communicate.
→ Barber shop has an entrance & an exit.
→ customers show up, take a chair when available, notifies a barber, sleeps until done, he wakes us up, then sleeps himself until someone else notifies/wakes.
→ sequence of steps
→ barber sleeps
→ customer comes in
→ wakes up barber
→ sits in chair (sleeps)
→ barber cuts hair
→ wakes up customer when done
→ customer leaves
→ barber goes back to sleep.

→ if someone is already in chair, customer sits in a wait chair until barber notifies the customer that a chair is free.

→ CHECKING ATOMICITY → "should be fairly important to a lot of us".

→ BE VERY CAREFUL TO AVOID RACE CONDITIONS!!!

- we can have failures of atomicity with OR without race conditions. eg: [T1: x++ | T2: x++]
- there is a technique to CHECK ATOMICITY.
  - → starting point for checking atomicity in general, but these processes are VERY EXPENSIVE.
  - → Lipton 1975 → "left/right movers"
  - → reduction technique → makes larger atomic chunks, & thus have a simple program.
  - → take a statement S. We want to make it atomic. S → [[S]].
    - → we can't naively do this everywhere or it could change the semantics/meaning of the program.
    - → Places to Consider.
      - → whether the program terminates or not.
      - → equivalence in terms of halting.

eg: a = b = 1

P(a)	P(b)
P(b)	P(a)
V(a)	V(b)
V(b)	V(a)

↕ making this atomic changes behavior (deadlock).

- → with no atomicity, deadlock is possible in different ways, so we can halt.
- → if its atomic, it is infinite → doesn't halt.
- → So: can't arbitrarily make chunks of code atomic.

→ So when can we make stuff atomic.

→ Set of rules:

Rule 1: If statement S is ever entered, it must be possible to leave (eventually)

- this alone is not yet enough

eg: x = y = 0.

x=0 → [x = 1 ; y = x ; P(y)] ← is this valid as an atomic statement? y would always be 1 at this point. → NOW IT NEVER HALTS

(if y == 0, this halts) ↳ changed behavior. (BAD)

COMP 409
OCT 26, 2010

✱ Assignment #2 (Problem) ⇒ as written → no dependencies

fix: loop body

```
a[i] = f[i]
if (i > 1)
    b[i] = b[i-1] * c[i-2]
else
    b[i] = g[i]
c[i] = a[i+3] + b[i]
```

Last time: atomicity checking S ⇒ [[S]], behavior differences, different data output, halting behavior

- continuing set of rules. (R1 is not enough).

Rule 2: The effect of statement [[S]] must be the same as S.

- Let f, g & h be statements inside a program, and let α be some sequence of statements from start.
- f is a RIGHT MOVER if for any computable α f h, when f, h are executed in different threads, then α h f is also a computation, and the result (re effect on all vars) of α f h and α h f are the same.
- g is a LEFT MOVER if for any computation α h g where g & h are executed in different threads, the α g h is a computation and the result of α g h and α h g is the same.

COMP 409
OCT 26 CONT'D

PAUL HUSSMANN

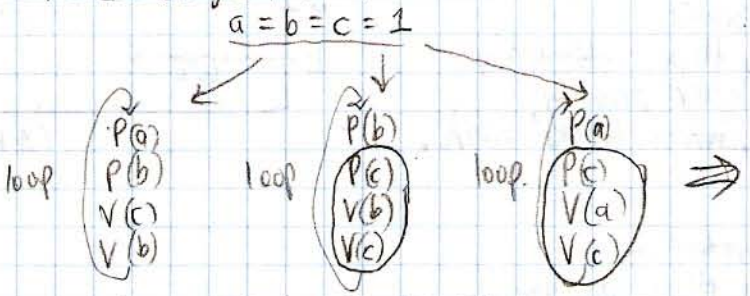
So we will use this to decide what is atomic and what is not.
left movers: release, unlock, V() operations → $\alpha P() h \Leftrightarrow \alpha h P()$
right movers: acquire, lock, P() operations → $\alpha h V() \Leftrightarrow \alpha V() h$

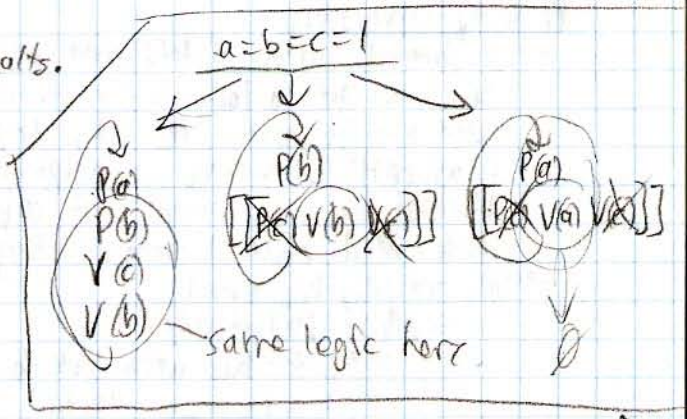
D-Reduction

$S_1, S_2, S_3, \ldots, S_n \Rightarrow [[S_1, S_2, S_3, \ldots, S_n]]$
sequence of statements ⇒ atomic version of same sequence.
This can be made with D-reduction.
So for some i we have
$S_1 \ldots S_{i-1}$ as all right-movers
$S_{i+1} \ldots S_n$ as all left-movers
(nb; S_i is unconstrained)

* → and each of $S_2 \ldots S_n$ is executed
Then: [[S]] is a D-reduction of S.
- If S' is a D-Reduction of S, then S halts IFFS' halts.

Example: Launching 3 threads:





we consider particular sequences:

P(c) | V(b) V(c) and decide that this sequence can be atomic
(right) (left) (left)

[[P(c) V(b) V(c)]] which makes the problem much smaller.

- we can do this again, and again...
we notice that with [[P(c) V(b) V(c)]] the P(c) and V(c) happens no matter what and since it's atomic, it's similar/equivalent to V(b) on its own.
- Then after, we see this; P(a) V(a) P(b) V(b) P(c) V(c) ⇒ this program does nothing.
- This is actually used in practice (research practice) as an atomicity checker, but it slows down the code by a factor of 10.
- This was done in (2003 Flanagan, Qadeer).
- This is NOT EASY!
- And they actually did find several atomicity bugs in JAVA Runtime Library, which were later corrected.

Scheduling & Priority

- there are different mappings
→ The OS used to schedule PROCESSES. Not threads.
→ Now the OS schedules KERNEL THREADS, which can then create USER THREADS.
→ A LIGHT-WEIGHT-PROCESS (LWP) is a KERNEL thread partitioned differently.

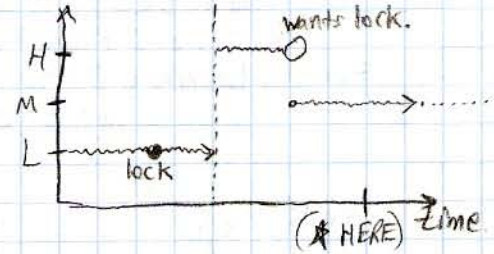
Models: One-to-One
- User Threads ARE Kernel threads (something).
- Advantage → Real Concurrency with multiple CPUs.
- Disadvantage → everytime we make a thread, there's MORE OVERHEAD through the OS.

Many-to-One (like green threads)
- many user threads on ONE KERNEL thread.
- Advantage → Requires no limited OS support, Less overhead (NB: avoid getting stuck in blocking calls)
- Disadvantage → No Real Concurrency with several CPUs.

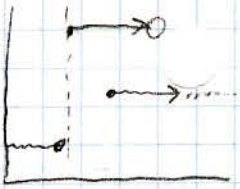
Many-to-Many (arbitrary mapping, most flexible model → most general, gives OS more freedom)
→ most flexible,
→ processor affinity ⇒ assigning certain things to specific CPUs.

PRIORITIES
→ DON'T USE THESE! (most often not necessary).
→ With RealTime Priority → easy to lock yourself out of the system.
→ there's different priority in different contexts.
Java has 1–10. where 10 is high priority.
→ windows NT had 7 priority levels, so Java's 1–10 gets squished somewhere in the 1–7.
→ Linux: 128 levels → is priority 10 in Java = 128 in Linux ? = 10? we don't know.
→ pThreads: ≥32 levels of priority. → sometimes it gets mapped down to fit, or mapped up & spread.
→ in pThreads priority levels are seperate for SCHED_RR and SCHED_FIFO.
→ how these mix? → SOMEHOW. (we don't know).

Priority Inversion
→ 3 threads with 3 different priorities: Low, Med, High.
If Low gets a lock befor the others start
and a High wants the lock, it's blocked,
So at (*HERE) in time (see graph →), High is effectively blocked by both threads that are lower priority!
Note: Med blocks L from unlocking, which blocks High.

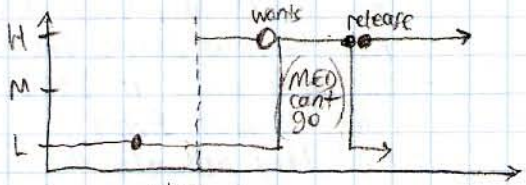


→ This actually happened
→ Mars Pathfinder
→ info-bus with lock to protect it.
→ bus manager thread is HIGH priority
→ meteriological Data is LOW priority.
→ communication thread is MED priority
→ Problem: Priority Inversion.
Scenario: Meteriological Thread (Low) gets lock, bus manager makes up, starts executing but can't get the lock. then communication starts running (takes a LONG time)

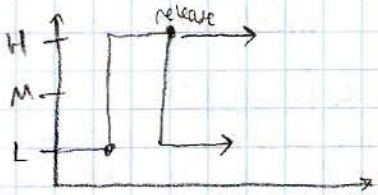


There's a seperate "watch-dog" process.
- if anything looks weird, it reboots the system.
- it spontaneously reboots, and the reason was hard to find, but this is why

TWO SOLUTIONS TO PRIORITY INVERSION
1) Inheritance
- Holder of a lock gets HIGHEST priority of anyone else trying to acquire it.



2) Ceilings
→ give the lock a priority
- a thread holding a lock has the PRIORITY OF THE LOCK.



PAUL HUSSMAN

COMP 409
OCT 28 2010

Last time ⇒ checking atomicity using Left-movers & Right-movers. (basis of a realistic technique).
- scheduling
- priorities → problems: priority inversion

Miscellaneous Java/Pthreads
→ ONE-TIME Initializers (for convenience):
- often we need to do something once after we are already multi-threaded.
- global variable of type pthread_once_t initialized to PTHREAD_ONCE_INIT
- now threads can call pthread_once(&once, &function). ⇒ many threads call it, but it only happens once.

THREAD SPECIFIC DATA

- errno (global) records the error value whenever you make a system call.

```
T1                          T2
x = read()                  x = read()
(errno = BADF) ----------> (errno = 0)   ← Problem: errno could be wrong
if (x<0)                                   depending on order of execution.
  printf("%d", errno);
```

- Solution: T.S.D. (Thread-specific-Data) in PThreads, or Thread-Local-Storage in Java.
  - → Idea: we have a single global variable name, but every thread has its own value.

In Java → allocate an object

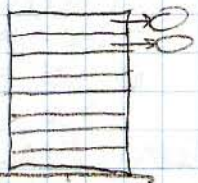
```
ThreadLocalStorage tls = new(...

T1                          T2
tls.set(__)                 tls.set(__)
p = tls.get()               q = tls.get()
```

} distinct values stored in the same place for each thread,

→ How to do this

Java
- have a hash table mapping thread ids to their corresponding values.
- (easy conceptually, but hard for efficiency). ⇒ was really slow!
- there are some clever schemes, but we won't go into that.



PThreads
- works differently,
- key, value pairs
  - → allocate a key (ie global handle for each thread).
  - → PThreads maintains a table
    - allocating a thread adds a row
    - allocating a key adds a column.

	key1	key2	key3
Thread 1			
Thread 2			
⋮			

  - → pthread_create_key(&key, &fn) ↳ destructor
  - when a thread terminates
    - calls destructor function on each key.
    - you recieve the actual TSD value → can do anything with it.
  - in the destructor, what if you create a new key?
  - once all the destructors are called for a thread, PThreads will do another pass to call destructor for any new keys to some limit.
    - limit = PTHREAD_DESTRUCTOR_TERMINATION
    - → can delete keys explicitly that doesn't call destructor
      - pthread_set_specific(key, value)
      - pthread_get_specific(key)
  - → nb! for errno as a special case, all the syntax for this is hidden.

MEMORY MODELS & CONSISTENCY

example

```
T1          T2
x = 1       y = 2
b = y       a = x
```

- possible final values of (a, b)
  - → interleavings.

x=1	y=2	x=1
b=y	a=x	y=2
y=2	x=1	b=y
a=x	b=y	a=x
(1,0)	(0,2)	(1,2)

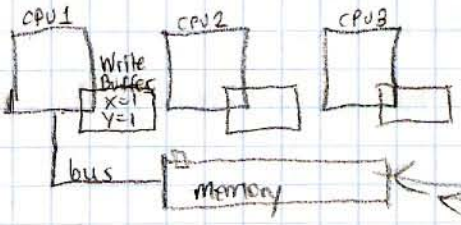
... (etc),

  - → notice (0,0) is not possible to achieve.
- Sequential Consistency ⇒ a multithreaded program is S.C. (Sequentially Consistent) if any execution is the same as if the ops of all processes were executed in some sequential order, and ops of each processor respect program order.

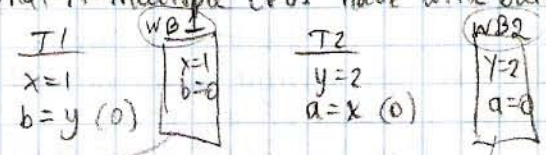
  nb: as if means it doesn't actually need to execute in that order but it must appear to.
- Strict Consistency ⇒ it MUST actually execute like S.C.

ex: T1 (x=1, y=1) T2 (z=3)
- we couldn't really tell if T1's instruction order is changed ⇒ same result no matter what.
- Sequential Consistency gives you the ability to switch things up, as long as the result doesn't change no matter what

ex: Write buffer ⇒ allows for multiple memory writes at a time as a batch process over a single bus.



- what if multiple CPUs have write buffers?



note: this time both end up as 0 ⇒ (a,b) = (0,0).
- So it's not common to have S.C. because of problems like this.
- No modern hardware guarantees SC for us,
- Intel gets very close to S.C. ⇒ strong model
- PowerPC is more weakly ordered → less guarantees than Intel.
- Dec Alpha was a very weakly ordered model.

nb: estimate: 20-80% of performance can arise from weak ordering.
→ so we need memory consistency models to tell us what we can actually rely on.

Memory Models
- spec/model of what is actually allowed, and what is not.
- coherence (one of the most common).
  - weaker than S.C. but quite popular (as weak as you probably want to go for human beings)
  - in S.C. we produce one global ordering for all reads & writes.
  - in coherence we have an ordering for each variable
    - each variable in isolation behaves in a S.C. fashion.

example:

P1	P2
x=0	=y → can be 2
x=1	=x → can be 0 ⇒ doesn't assume x must be 1 eventhough y is already 2.
y=2	=x → can be 1, but after that it can't be 0.

coherent, but not SC.

- PRAM - Pipelined Ram model
  - writes of a single processor will be seen in the same order by every other processor
  - still have multiple orderings → one per processor.

P	Q
x=0	y=0
x=1	y=1
=y (0)	=x (0)
=y (1)	=x (1)

→ as long as x is 0's it can see 0's again, or 1's but once it's a 1, it can't go back to 0. (follows order that x is written in P).

- this is PRAM but not SC.

nb: PRAM & Coherence are NOT comparable. ⇒ one doesn't imply the other.
look at above example ⇒ in coherent, but NOT PRAM.

ex:

P	Q
x=0	x=1
=x (1)	=x (0)
y=1	y=2
=y (2)	=y (1)

in PRAM, but not coherent.

COMP 409
OCT 28 CONT'D

PAUL HUSSMAN

- Release Consistency
  - → acquire & release (lock & unlock)

```
a =
b =
acquire   ↑ acquire can move up, but not down
x =
y =
z =
release   ↓ release can move down but not up
c =
d =
```

This is called "ROACH MOTEL" symantics (can check in but not check out)

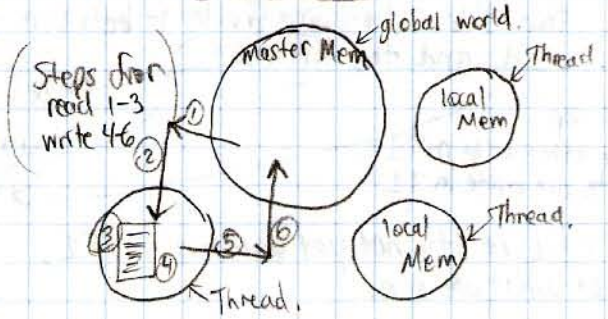
⇒ MODERN HARDWARE GIVES YOU NONE OF THESE!

COMP 409
NOV 2, 2010

Midterm next class. OPEN BOOK, no electronics.
Covers up to but NOT including memory consistency,
any programming question (if any) will be pseudocode and short.

Last Time
- → Memory Consistency (Models). simple, basic model (usually assumed): Sequential Consistency.
  - → In practice, you never really get Sequential Consistency.
  - → There are many → one of the most important: Coherence. (kind of like a lower bound)
    - → another interesting one: PRAM
  - → Acquire/Release semantics ⇒ roach motel.
  - → Location Consistency
  - → casual consistency → etc.
- → Hardware has its own models ⇒ whatever is in the CPU.
  - → Intel & AMD aren't as bad, you just have to look at the manual for corner cases otherwise is pretty-much close to S.C.
- → all Multiprocessor hardware needs memory models.
- → most languages DON'T. (unless they have explicit concurrency).
  - → Java → interesting example ⇒ has concurrency built in. & runs on a Virtual Machine (→ multi-thread).
    - Java HAS a memory model
    - In fact it's on it's second version

OLD MODEL



- read - load - process ← local Memory
- process - export - write.
- lots of little things to worry about
- old model implied coherence. (serious problem!)
  - example by (Bill Pugh)

```
→ int i = p.x;
  int j = q.x;
  int k = p.x;      } compiler optimization problem
⇒ could say int k = i; // FASTER
```

- what if at runtime another thread writes to p or q and p & q are both pointing to the same thing. Say it writes a 1, and it used to be 0.
- You would see the OLD value in k ⇒ Violates Coherence! ⇒ The optimization would be wrong.

so it turns out that you can't really optimize much this way!
(most compiler optimizations are invalid).
Everyone has always tried to improve Java performance, so this is a problem! (most).
A NEW MEMORY MODEL was introduced ⇒ JSR 133 - memory model allowing for compiler optimizations!
Also thinking of the programmer to make things easier without having to follow so many rules
The easiest would be Sequencial Consistency, but that's very hard to model.
They used something in between.

→ Divide Programs into two classes.

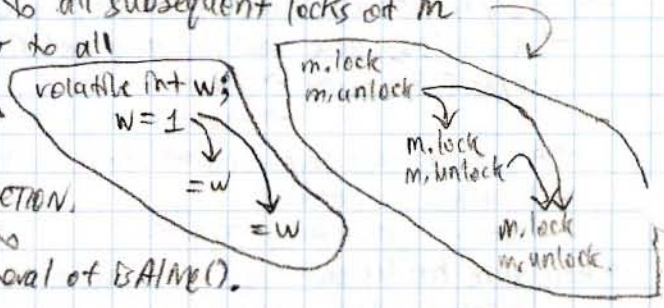
(1) Correctly Synchronized Programs.
- → CONTAIN NO DATA RACES.
- → In this case Sequential consistency is guaranteed. (good for the programmer).

Could stop here (as in C++) ⇒ Incorrectly synchronized programs behave in an undetermined way.

(2) Incorrectly Synchronized Programs.
- → There are many semantics for this
- → We want defined semantics in this case (unlike C++).

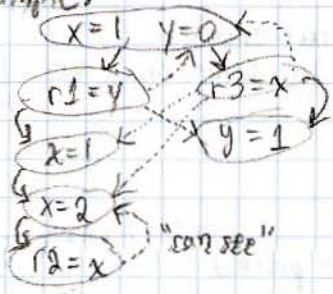
BASIC MODEL → HB "Happens-Before" Consistency
- → Build a graph from the actions found in execution traces.
- → nodes are runtime actions.
- → edges represent "Happens-Before" relations

HB Edges
- → between each action of a thread t to the next action in t. (intra-thread)
- → From an unlock of monitor M to all subsequent locks of m
- → From each write of a variable v to all subsequent reads
- → From the initialization of a thread to its FIRST ACTION.
- → From Thread.start() to the FIRST ACTION.
- → From the final action in a thread to any subsequent join() or false eval of isAlive().
- → etc.

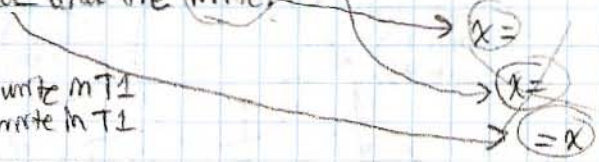


So we build this graph

Example:



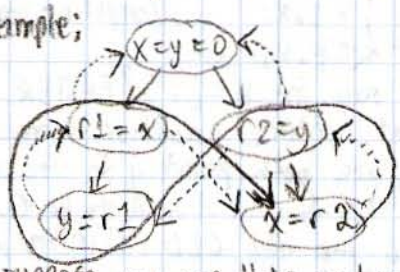
- ⇒ use the HB edges as a PARTIAL-ORDER.
  ie: x < y iff there is a HB-path from x to y.
- → a read r of a variable x is allowed to see a write of x if:
  1) r is NOT ORDERED BEFORE the write
  2) THERE IS NO INTERVENING write to x, such that this intervening write is between the read and the write.



Notice r3 = x can either be x = 1 from the top
x = 1 from the other write in T1
x = 2 from the 2nd write in T1

- so r3 can be = 2 in T1 (due to an optimization) even if it's not yet written in T1.
- so the output can be verified in terms of what we see.

Example:

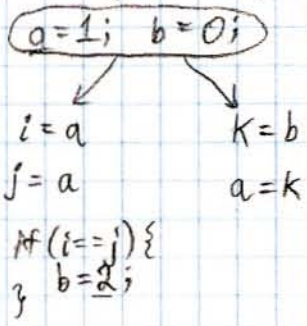


⇒ any value printed out here is legal according to the memory model. "out-of-thin-air" values. ⇒ considered bad! (for programs)
(come from leftover stack/memory data).

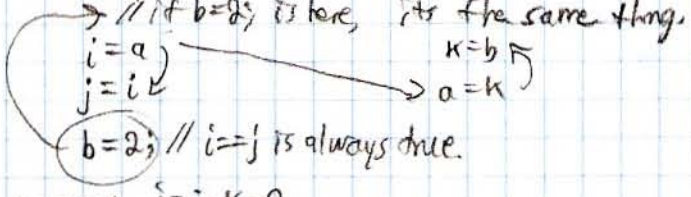
→ suppose you ran this and got x = y = 42. Would you be able to complain to Sun? No!

COMP 409
NOV 2 CONT'D

- Maybe we should dis-allow these "causal cycles"
  But ⇒ are all cycles bad?
  Example:



- can it result in i=j=K=2?
- How can we optimize T1's code?



- so now i=j=K=2