COMP 409
NOV 2 CONT'D

(PAUL HUSSMAN)

- Maybe we should dis-allow these "causal cycles"
  But ⇒ are all cycles bad?
  Example: (a=1; b=0;)

```
      i = a          k = b
      j = a          a = k
      if (i==j){
        b=2;
      }
```

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

```
   → // if b=2; is here, its the same thing.
   i = a                    k = b
   j = i                    a = k
   b=2; // i==j is always true.
```

- so now i=j=k=2

COMP 409
NOV 9 2010

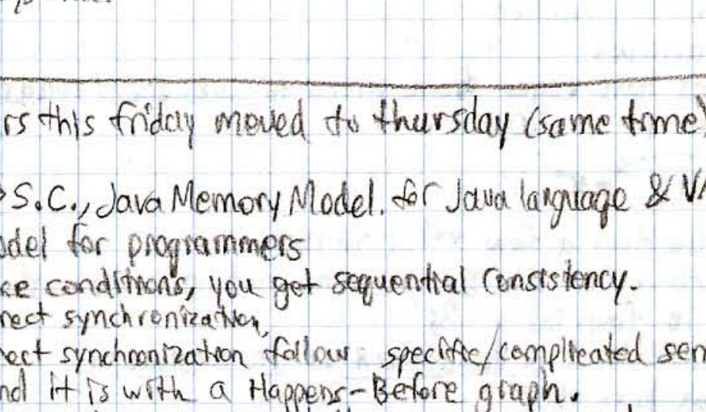
Note: Clark's office hours this friday moved to thursday (same time)

(PAUL HUSSMAN)

Last Time → memory consistency ⇒ S.C., Java Memory Model. for Java language & VM.

JMM → gives a nice, simple model for programmers
- → If you can avoid race conditions, you get sequential consistency. ie; must have correct synchronization.
- → programs without correct synchronization follow specific/complicated semantics.
- → one way to understand it is with a Happens-Before graph.
  - → can use this HB-Graph to justify whether a value seen during a read is allowed. (HB-Graph gives a partial order on runtime actions.)
  - → you can see writes that are unordered with the read (cannot skip a write) (can't see a write in the future), everything else is fair-game.
- nb: This can introduce CYCLES (as we saw last class). (unintuitive!)
  - → we may think of disallowing cycles, but we can't! (some are good).

Nice & Complicated Figure: (just for our interest)



- Hard to allow all these things.
- The JMM is complicated!

- we end up with a justification process.
  - → we won't go through it because it's HUGE!
  - → we start with a trace
    - → incrementally build & commit trace.
    - → commit actions
    - → restart the trace each time.
- → nb: it is still broken (there are flaws)
  - → some flaws have been repaired

nb: Orderings in the Happens-Before Graph.

→ intra-thread ordering.
→ edges between threads are the more interesting ones.
→ HB-ordering is actually the merge of intra-thread order and synchronization order.
→ synchronization order
  → an order which locks are acquired & released.
  → (there's an order of locks & unlocks that happened).

lock / unlock ? — lock / unlock

eg:

T1	T2
x = 1	lock m
lock m	r1 = y
y = 1	unlock m
unlock m	r2 = x

(if T2 locked 1st) (if T1 locked first)

- one possibility is if T1 got the lock first.
- the other possibility is if T2 got the lock first.
- so there's 2 HB Graphs depending on ordering.
- if its not an actual trace, we may need to think about all the different possibilities!

- Race Condition
  → unordered access
    - lock m / x = / unlock m — lock m / = x / unlock — } no race condition here
  → volatile variables
    - volatiles have edges from writes to subsequent reads
    - volatile int x = 0;

      x → = x

→ The JMM also does a few other things
  → it affects a few things you may not have guessed.
  FINALS ie final int x = 3;
  - can be initialized at declaration or in constructor.
  what if we have this:

```
class foo extends Thread {
    final int x;
    foo {              //CONSTRUCTOR.
        synchronized (this) {
            start();   //Note: don't start threads in the middle of the constructor
                       and initialize at declaration when possible.
            sleep(10); // no guarantee. ⇒ means nothing.
            x = 42;
        }
    }
    public void run() {
        int i = x;     // might read 0. (default value) ⇐ Problem. ⇒ It's not actually a constant.
        synchronized (this) {
            i = x;  // x = 42
        }
    }
```

→ Finalization (hard to use well).

```
class foo {
    :
    finalize() { // executes after garbage collection (object is no longer pointed to)
        ≡        but BEFORE the memory is given to anything else.
    }
}
```

COMP 409
NOV. 9 CONT'D

PAUL HUSSMAN.

```
class Foo {
    private File f;
    Foo() {
        f = new File(___);
    }
    public void write (int x) {
        f.write(x);
    }
    public void finalize() {
        f.close();
    }
}
```

Example:

```
y = new Foo();
y.write(17)
y = null;  // don't need it anymore,
           // garbage collector collects & runs finalizer.
```

Code Optimization

```
y = new Foo(); => { y = allocate space          <- becomes collectable HERE due to optimization.
                    ~~constructor(y).~~
                    r1.f = new File(___);
y.write(17)    =>   r1.f.write(17);
```

-> so at point of collection, it can run the finalizer!
  -> the file **closes before** file gets written to!!! BAD!
-> So we realize that finalizers should be treated as concurrent methods.
  => USE SYNCHRONIZATION (or don't use finalization).
  -> finalizers aren't even guaranteed to run in Java.

sequential consistency for (race-free) program

-> this is a very strong guarantee.
-> being race-free is a runtime/dynamic property
ie: $x = y = 0$

```
T1                          T2
do {                        do {
    r1 = x;    } ∞ loop         r2 = y      } ∞ loop
} while (r1 == 0);          } while (r2 == 0)
y = 42;  ↓ never gets here  x = 42;  ↓ never gets here
```

(What if x=42; was moved to the top? it would introduce a race condition!)

- infinite iterations of T1 interleave infinitely with infinite iterations of T2.
- This program is RACE-FREE due to divergence.
- the race condition is statically present, but dynamically absent.

COMP 409
NOV. 11, 2010

Clark will not be here next week => Chris D, Chris P.
-> conference in the country -> limited email.

PAUL HUSSMAN.

Last Time

Java Memory Model -> HB consistency, HB graph -> intra thread + sync -> allows some bad cycles
justification process -> causality -> no values out of thin air -> Race free = runtime property

-> Programs can be race-free due to unexercised code
  - we can no longer assume FORWARD PROGRESS (issue for compiler developers)
  -> This affects all control flow.

```
while () {                  while () {
    if (b) {                    if (c) {
        r1 = x                      r2 = y
    } else {                    } else {
        y = 42                      x = 42
    }                           }
}                           }
```

race condition only happen if an else clause gets executed.

Double-checked Locking

```
class Foo {
    private Helper h = null;
    public Helper getH() {
        if (h == null) {
            h = new Helper();
        }
        return h;
    }
}
```

- In a multithreaded environment, this needs to be fixed:
  - → easy way is to make getH() a synchronized method.
    - this involves overhead → might want to improve performance.
  - → Double-check Locking:

```
class Foo {
    private Helper h = null;
    public Helper getH() {
        if (h == null) {
            synchronized (this) {
                if (h == null) {   // double check
                    h = new Helper();
                }
            }
        }
        return h;
    }
}
```

Problem:

h = allocate space ← not null here!
call constructor } 2 steps.

Some other thread can then see our partly constructed h.
⇒ PROBLEM.

- → THIS DOESN'T WORK! (subtle reason)
- → BUT it does work for PRIMITIVE values ⇒ if h was an int.
- → IT ACTUALLY CAN BE MADE TO WORK for objects. (must avoid race conditions.
  - can make h volatile ⇒ but synchronized vs volatile are similar in cost
  - so double checking is not always a good idea

JSR-166 (finished)

JAVA UTIL. CONCURRENT.*

- Rich set of things for concurrent programming at a slightly higher level.
- Executors → flexible way of defining how a thread will be executed.
- Locks →
- Barriers →
- Future →
- Queues → blocking & non-blocking queues.
- Timing. → better timing facilities.

java.util.concurrent.lock.

- like in pthreads, you can allocate a lock.

```
Lock l = new Lock();
    l.lock()      } watch for exceptions! ⇒ ( try (lock)
    l.unlock()    }                           finally
                                                (unlock) )
```

- makes it easy to convert pthreads to Java!
- locks do not interrupt. (no interrupted exceptions).
  - If you want to interrupt: lock.interuptable().
- tryLock primitive: another non-blocking attempt to acquire a lock based on a small time-out.
  - eg: tryLock(100, TimeUnit.NANOSECONDS)
- reentrantLock
- reentrantReadWriteLock ⇒ readLock, writeLock

COMP 409
NOV. 11, CONT'D

PAUL HUSSMANN

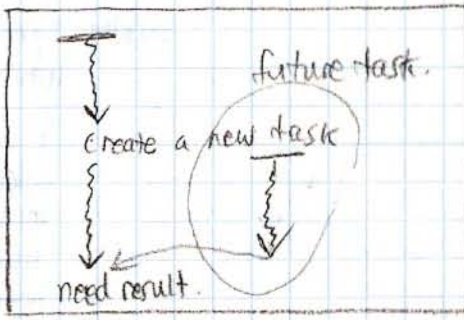
java util.concurrent.atomic
- → Atomic Data: AtomicBoolean, AtomicInteger, AtomicLong, AtomicReference.
- → getAndSet
- → addAndGet. → similar to Fetch & Add. (FA)
- → decrementAndGet
- → compareAndSet → similar to Compare-and-Swap (CAS)
- → weakCompareAndSet ⇒ sometimes fails spuriously but higher performance.

→ These DO NOT replace regular booleans, integers, etc...
→ StampedReference → pointer to an object w/ a version/time stamp. (time, ptr)
→ MarkedReference → boolean value associated with the reference.
→ Semaphores → can use existing semaphores. → can be fair ⇒ FIFO
→ Explicit condition variables → from a lock you can allocate a new C.V.
→ keep in mind: SPURIOUS WAKEUPS ARE STILL POSSIBLE.
→ waitUninteruptably() → don't have to catch InterruptedException. Ignores interrupt calls
→ spurious wake-ups are still possible.

Barriers
- → threads can't continue until they all get to a particular point.
- → can be made with a semaphore.
- → making it work once is easy
- → making it work in the case that threads can retry to get past the barrier is harder.
  - → make sure everyone arrives, and make sure everyone's gone (cyclic barrier).
- → CyclicBarrier → waits for specified threads, and then allows them to continue executing.
  - → can also specify a method to be executed before threads are released from the barrier, but after they have all arrived.
- → CountDownLatch → One-shot barrier → initialize to a count.
  - → every thread counts down and then waits for the barrier.
  - → If you want to reuse it, it must be re-allocated. ⇒ One-shot.

Futures (made for functional languages → useful for java).
- Start a task to compute something we'll need in the future.
- f = new Future(__); //create new task.

  f.get(); // waits for result. (hopefully it's already done).



Executors
- → replace "new Thread"
- → allow code to be executed in different ways depending on the executor.
  Executor ex = new Executor();
  ex.execute(Runnable Object);
- → can create different kinds of executors:
  ThreadPoolExecutor
  - FixedThreadPoolExecutor → fixed # of threads: n
    → feed them tasks. & executes them.
    → pass more than n threads → they get queued
  - Unbound # of threads → similar to just adding new threads.
  - ScheduledThreadPool → to execute code at a specified time in the future / repeatedly.. (every x ms)

COMP 409
NOV 16 2010 | CHRIS DRAGERT | PAUL NUSSMAN

Message Passing.

- we have mostly been using shared memory to communicate between threads
- sometimes its easier, as with Producer/Consumer, to think of threads as sending & recieving messages ⇒ Message Passing.
- IDEA: threads/process are independent entities
  - have shared, global channels for communication

2 main forms of communication

① Asynchronous Message Passing
  → send operation returns immediately.
  → recieve BLOCKS until a message is ready. (future info might depend on msg content)
  → since send returns immediately, we don't know what happens with the message & when it happens.
  Therefore message delivery time is unbounded. eg: physical mail.
  → Variants: bounded channels, FIFO

② Synchronous Message Passing
  → send BLOCKS until recieve
  → recieve BLOCKS until message is ready. (as before).

Code Sample: (Java Synchronous Channel)

```
public class SynchChannel {
    private Object msg;
    public synchronized void send (Object o) {
        msg = O;
        notify(); //notify reciever.
        while (msg != null) {
            wait();
        }
    }
    public synchronized Object recieve() {
        while (msg == null) {
            wait();
        }
        object ret = msg;
        msg = null;
        notify(); // notify potential sender
        return ret;
    }
}
```

Which is better?
- asynchronous is more flexible ⇒ send messages then move on... (realistic)
- we actually get a lot of value out of synchronous message passing. It turns out to be MORE EXPRESSIVE. (Really? YES.).

Consider the following example:

P	Q
send(Q, 23);	x = 0;
	x = recieve(P);

→ what does P know about Q?
  asynchronous; could be x=0 or x=23
  synchronous: x = 23. ⇒ synchronous gives more knowledge!

PAUL HUSSMAN.

Common Knowledge

in an asynchronous system, if A sends to B, how does A know that B recieved it?

→ acknowledgements? B → A

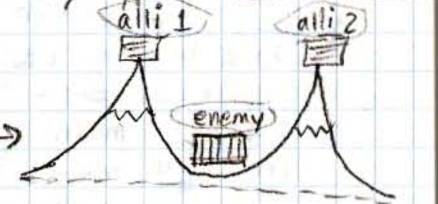
- how does B know if A got the acknowledgement?
- so A has to acknowledge the acknowledgement... etc.
- Here B knows m but A doesn't know
- then A knows but B doesn't know ..., etc.

→ So How do we resolve this?

- in asynchronous communication, we can never gain common knowledge!
- in synchronous communication, common knowledge is TRIVIAL!

Therefore SYNCHRONOUS communication is required for any type of coordination.

→ 2 Army Problem.



- Two armies occupy neighboring mountaintops and the third army (enemy to both) is in the valley between →
- enemy is bigger than each other army alone but not combined.

J. Gary - 1978

- If both allies attack at the same time, they win
- if only one alli attacks, the enemy wins.
- Therefore neither alli wants to attack alone → must COORDINATE

Idea: Send messages via messenger

→ unfortunately, the messenger must pass through the valley → but he might not make it.

→ alli 1: "attack at noon"
alli 2: "ok"
alli 1: "ok, its on!"
etc } doesn't work.

⇒ no sequence of messages/acknoledgements can guarantee coordination!

⇒ in fact in an asynchronous message passing model, COMMON KNOWLEDGE can be neither gained, nor lost.

→ how do we lose knowledge?

$P_1 \xrightarrow{synch} P_2$

P_1 has lost the knowledge that P2 never recieved a message from P1 etc...

→ Muddy Children Problem

[J. Burwise Scenes & other situations]

- n children are sent out to play and told not to get dirty.
- $K < n$ children get dirty.
- parent announces: "at least one of you has mud on your forehead".
- parent repeatedly asks "do any of you know if you have dirt on your head?"
- children are truthful and clear.
- children all answer at the same time (in unison).
- what responses do they give?

Answer: K-1 times, all children say no.
K^{th} time all children with mud say yes.

Proof: (by Induction)

for $K=1$, lone dirty child sees no one else with dirt ⇒ says "Yes".

for $K=2$, first time c_1 sees c_2, c_2 sees c_1 ⇒ both conclude its not them

second time: c_1 notes that c_2 said "no" before, so c_1 concludes there must be at least one other with dirt and no other dirty children ⇒ "Yes" (same for c_2)

assume true for $K-1$ ⇒ K^{th} time → at the $(K-1)^{th}$ question, c_1 sees $(K-1)$ dirty children ... etc.. ⇒ "Yes".

k^{th} time: - at the $(k-1)^{th}$ question, c_1 sees $(k-1)$ dirty children
- if this was all of them they would have said yes, to the $(k-1)^{th}$ question but they didn't
- they conclude there must be one more ie themselves → same for $c_2 \ldots c_k$.

→ Note: this only works because the parent introduced COMMON KNOWLEDGE to the children: that at least one was dirty.
parent announces something that all the children already know (if $k>1$)
→ everyone sees a child with muddy forehead.

→ oddly, had the parent not announced this, it wouldn't have worked.
→ each time the parent asks, everyone will say "no".
→ clean child always says no
→ dirty child $(k>1)$ will have exactly the same info as the clean child → must answer "NO".

- We have to think about knowledge in a different way.
- [Halpern & Moses - 1990, Knowledge and common knowledge in a distributed environment]* (LOOK AT PAPER FOR MORE INFO ↑ (google scholar))

Let $K_i\psi$ mean agent i knows ψ

Let $D_G\psi$ mean group (G) has distributed knowledge of ψ
eg: agent i knows δ and agent j knows that δ implies ψ, then $D_G\psi$.

Let $S_G\psi$ mean someone in G knows ψ.
eg: $S_G\psi = \bigvee_{i=G} K_i\psi$

Let $E_G\psi$ mean everyone in G knows ψ.
eg: $E_G\psi = \bigwedge_{i=G} K_i\psi$.

Let $E_G^k\psi$, $k\geq 1$ mean that $E_G^1\psi = E_G\psi$
$E_G^{k+1}\psi = E_G E_G^k\psi$
(etc)

ie: "everyone knows that" repeated k-1 times + "everyone knows ψ."

Let $C_G\psi$ mean that ψ is common knowledge.
ie: $C_G\psi = E_G^1\psi \wedge E_G^2\psi \wedge \ldots$

We can make a hierarchy of these knowledges
$C\psi$ implies $E^{k+1}\psi$ implies ... implies $E\psi$ implies $S\psi$ implies $D\psi$ implies ψ.
(not always distinct) → shared memory - all are equal
→ distributed - all are strict

so in the muddy children problem,
m: "at least one is muddy"
before the parent speaks, the children have $E^{k-1}m$ but NOT $E^k m$
→ see article for the rest.

In the 2-army problem its harder.
→ K muddy children require $E^k m$
- 2-army problem requires Cm (common knowledge).

A2 is almost done being marked.
→ has been fairly well done...
→ people are over-synchronizing stuff!
→ people synchronized the move() operation which isn't best since its a collection of smaller procedures.
→ make code to calculate new position
then synchronize (this) just for the actual move.
(we wouldn't have lost marks, but this way is much cleaner).

→ for loop questions the semaphores should tighten-up synchronization around critical sections

comp 409
NOV 18 2010

[PAUL NUSSMAN]

→ Clark's PhD student.

CHRIS PICKETT chris.pickett@mail.mcgill.ca sable.mcgill.ca/~cpicke

"Speculative Multithreading" a.k.a "Thread-level Speculation" (making uni-thread stuff to multi threaded).

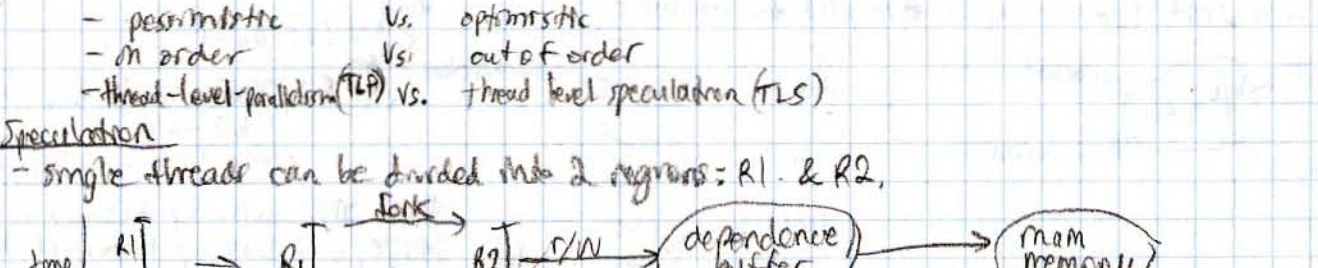
- we've been doing manual parallization ⇒ locks, critical sections, threads.
- there are special languages for concurrency (not C or Java).
  - eg: Linda, see comp 623: concurrent programming languages (taught by clark), CSP, OpenMP
    for OpenMP, the programmer does the parallization explicitally (convert single T → multi T or do it from scratch).
    - there are pros & cons.
  - automatic parallization: write your program as a single thread, and the compiler or runtime system parallizes.
    Issues: - less control over what to actual parallize.
    - might be over cautious → you could manually parallize some parts that the compiler wouldn't see.
    - what if you want to create a "background" process? → would have to do it manually.
    so certain behaviors & symantics are not feasible for an automatic compiler to detect.
  doing things manually is different though.
    Issues: can have race conditions, can have deadlocks, can have weird bugs/behavior,
    ~2% of programmers know about concurrency, and 2% of them do it right,
    so auto-parallelization can be really useful.
- Auto parallelization is a work in progress since the 80s, but it is so complicated for everything other than loops. Not a lot of programmers rely on it.

Kinds of Auto parallization

- static AOT compiler
- dynamic JIT (just in time) routine (eg: VM).
- non-speculative vs. speculative
- conservative vs. non-conservative
- pessimistic vs. optimistic
- in order vs. out of order
- thread-level-parallelism (TLP) vs. thread level speculation (TLS)

Speculation

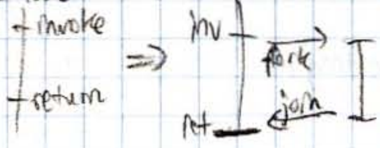
- single thread can be divided into 2 regions: R1 & R2,



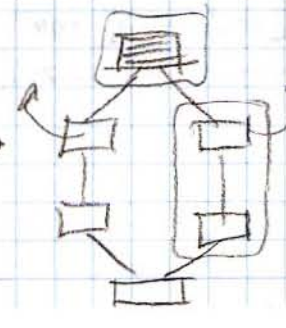
executes R2 in parallel with R1
R1 deals w/ local mem
R2 deals w/ dependence buffer.
at join(), reads & writes are in buffer.
speculative region gets the earliest read from the buffer.

Kinds of Speculation

loop-level
  i, i+1, i+2, i+3, ... (one iteration to different CPUs).
method-level



basic-block-level



divides program into threads according to control flow, and executes parts of the program in parallel.

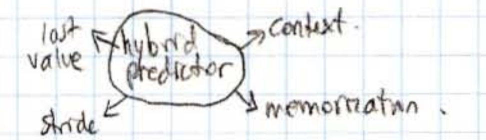
transactional-memory (4th kind of speculation).
- atomic {
  ≡ ← guarantees atomic parts. uses an undo-log if it doesn't finish an atomic section.
  }
- In practice they haven't ended up using this yet.

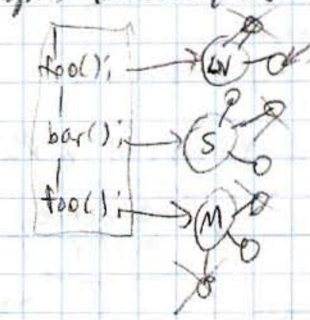
Chris Pickett's Thesis.

- existing thread-level speculation (TLS) / SpMT is reserved:
  - done in: { loop-based, hardware (doesn't exist), C, C++, Fortran (popular).
  - Chris's Thesis: { method based, software → pros: HW exists, easy to modify; cons: correctness, overheads; Java VM }
- Initial Java VM
  - added stuff: - dependence buffer
    - TLS for Java language: gc, synchronization, exceptions, native methods, class loading etc.
    - bytecode interpreter
    - basic overall VM design
    - MLS
    - RVP - return value prediction.
    - fork heuristics
    - profiling: finds what's slow/bad.
      1) RVP is expensive → tried to make it cheaper.
      2) extra CPUs were often idle. → tried to support nested speculation
      3) threads created were short. → tried having a better fork heuristics.

```
r = foo(a,b,c)
if (r>10) {
    l1 = x
    y = l2      → dependence buffer.
}
```

- Was done initially in sable VM.
- then refactored it to a library (VM independent)
- adapted it for IBM JIT
- used soot (Java compiler at mcgill) for analysis.





constant space { LV: $v_n - 1$; S: $v_n - v_{n-1} + (v_{n-1} - v_{n-2})$; 1, 2, 3 → 4. (after 1,2,3 it predicts 4).

grow in size { M: looks up f(args) in table; C: look up history of length c in table.

- free memory & execution time of un-used predictors.
- optimizes RVP.
- tested with SPECjvm98 (one of the first benchmark application for Java).
  - compress, database, javac, jess expert, jack parser, mpeg audio, raytracer (mtrt).

gave threads scores
  expected length × success rate.
  determines if it's a good idea to make a thread (fork heuristics).

nesting

```
call stack   ↑ c []          e[]
(out of      | b [] → [] b'  d[] → [] d'
order        | a [] main → [] main' → [] main''
nesting)     ──────────────────────────→ (in-order nesting)
```

- chained threads.