

Concurrent Java

Java Programming in a Multicore World

Angelika Langer

Trainer/Consultant

<http://www.AngelikaLanger.com/>

objective

- take look at current trends in concurrent programming
- explain the Java Memory Model
- discuss future trends such as lock-free programming and transactional memory

speaker's relationship to topic

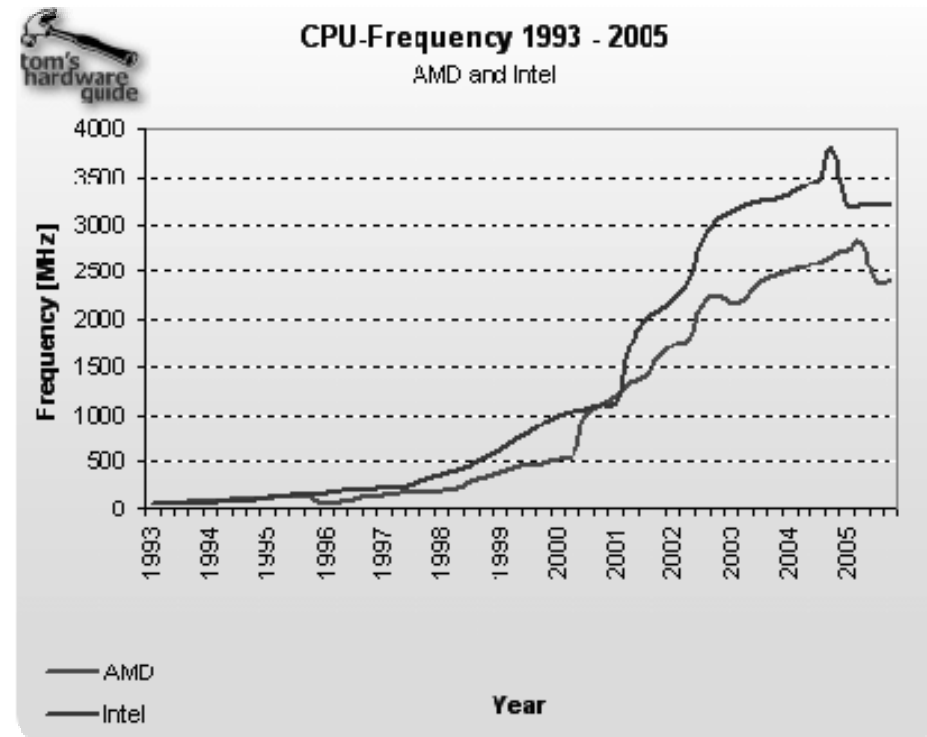
- independent trainer / consultant / author
 - teaching C++ and Java for 10+ years
 - curriculum of a dozen challenging courses
 - co-author of "Effective Java" column
 - author of Java Generics FAQ online
 - Java champion since 2005

agenda

- **history of concurrency & concurrency trends**
- synchronization and memory model
- fight the serialization – improve scalability
- future trends

CPU development

- Moore's law:
 - number of transistors doubles every two years
 - since 2004: more cores
 - until 2004: faster ones
 - main reason: heat
- 2 cores became standard 2007
 - 6-12 in 2009 (AMD)
- more complex caches
 - hierarchy



CPU development implies need for JMM

- new CPU does not solve our performance problems
 - if our program does not scale (well) to multiple cores
 - i.e.: find (and fight) the serialization
- existing programs
 - undetected errors might pop up
 - multi-core + caching uncovers synchronization problems

Java history – initial MT support

- low level functionality built into the language
 - synchronized block/method
 - `Object.wait()`, `Object.notify()`
 - no thread pool, no blocking queue, ...
- an incomplete and hard to understand memory model

Java 5.0 MT support

- explicit locks and conditions
- high-level abstractions (thread pool, synchronizers, ...)
- lock free programming (atomics + concurrent collections)
- reworked memory model
 - clarifies what `volatile` and `final` mean in a MT context
 - defines requirements regarding atomicity, visibility and ordering

Java keeps up to ...

- ... the needs and requirements of the changing MT uses
- more people build MT programs
 - MT patterns and idioms become common knowledge
 - need for high-level abstractions
- more people build Java MT programs
 - for multiprocessor platforms
 - need for clear and exact memory model
 - wish for better scaling MT abstractions
 - need for lock free programming
- former niche becomes main stream with multi core CPUs

architecture history – mid 90ies

- no or insufficient support of threads
 - proprietary UNIX thread implementations in the user space:
 - blocking `read()` in one thread blocked all threads of the process
 - Java started with a similar model: green threads
 - non-preemptive, sometimes obscure behavior
- threads used to structure programs –
not to achieve more through put
 - scalability was not an issue, multi-processor systems were rare
 - multi-core unknown
 - user space threads scalability limited with multi-processors

architecture history – since then

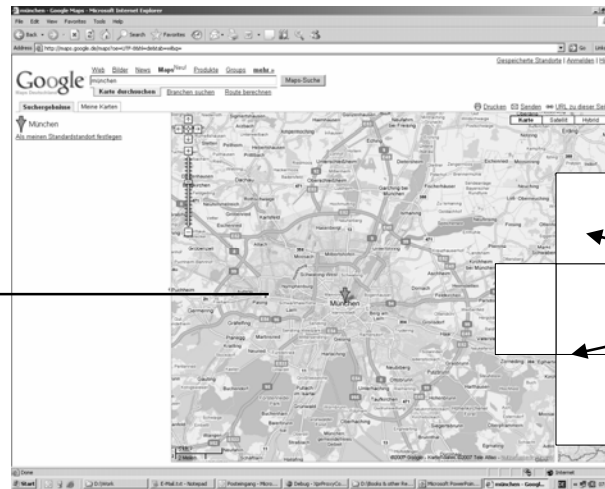
- trend to asynchronous and parallel computing
to increase throughput
- Java examples
 - asynchronous I/O
 - 1.4 socket, 5.0 sockets + SSL, 7.0 sockets + file system
 - essential: frees you from one thread per socket / file
 - but: program structure gets more complex and technical
 - JMS introduced 2001
 - much later than RMI which was part of Java from the beginning
 - effect: EJB became *message-driven beans*

(cont.)

- general example:
 - AJAX (Asynchronous JavaScript and XML)
 - means: user interaction decoupled from HTTP requests
 - traditionally
 - you select a link / push a button / etc. , and
 - a new page gets loaded into your browser
 - AJAX example: Google Maps

user interaction

**e.g. pull the map
to the left**



**map elements are
asynchronously
pulled from the server
via Javascript**

more AJAX

- Google Maps
 - user interaction decoupled from HTTP requests
- more asynchronicity
 - HTTP push via Ajax
 - signal an asynchronous event in the browser
 - e.g. incoming telephone call
- solutions boil down to ‘long-lived’ HTTP requests
 - long polling
 - HTTP streaming

'long-lived' HTTP requests

- persistent communication / long polling / hybrid polling:
 - request lives, until the event occurs
 - or a 'long' timeout occurs (5-10 minutes)
 - event is signaled in the response
 - or the timeout
 - new request to poll the next event

- comet style / HTTP streaming:
 - request lives, until the client goes away
 - all data is send from the server to the client in the same response

problem in traditional web servers

- traditional servlet programming
 - one thread:
 - receives HTTP request
 - determines what has to be done
 - gathers the data (and renders the new page)
 - sends all this back to the client in a response
- what about an long-lived open HTTP request ?
 - that waits for an external event
 - e.g. the incoming telephone call
- allocates a thread until
the event occurs / client goes away !
 - with 50000 users on the server !

asynchronous web servers

- decouple the request from the response
- Jetty 6 Continuation
 - `Continuation.suspend()`, `Continuation.resume()`
 - <http://docs.codehaus.org/display/JETTY/Continuations>
- Tomcat 6.0
 - Comet module allows to process I/O asynchronously
 - <http://tomcat.apache.org/tomcat-6.0-doc/aio.html>
- Java standard for asynchronous web server
 - JSR 315 = Servlet 3.0 specification
- underlying concept: asynchronous I/O

more and more asynchronicity

- not only web server – other servers too
- SOA (service oriented architecture)
 - service -> service -> service ...
 - you don't want to have a waiting thread in each of the server
 - i.e.
 - asynchronous handling of the request
 - MOM (message oriented middleware), means often JMS in Java
- all this means:
 - you need multiple threads and some synchronization of these to tie the external asynchronous channels to your program

agenda

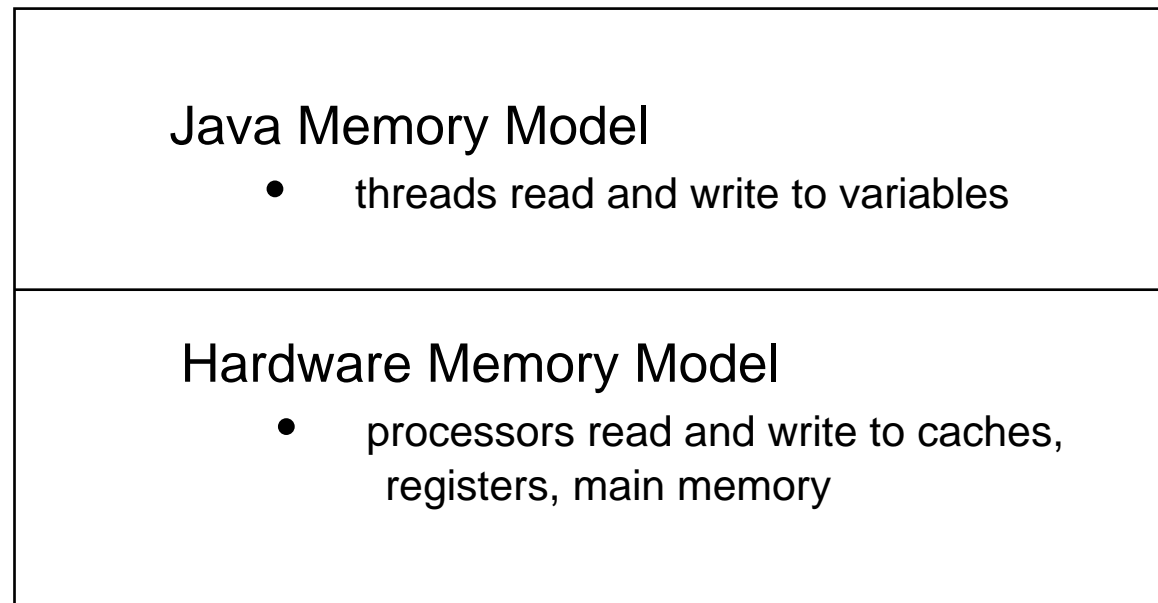
- history of concurrency & concurrency trends
- synchronization and memory model
- fight the serialization – improve scalability
- future trends

motivation - why does JMM matter?

- **JMM = Java Memory Model**
- understanding JMM reveals errors in existing programs
 - undetected errors might pop up
 - multi-core + caching uncovers synchronization problems

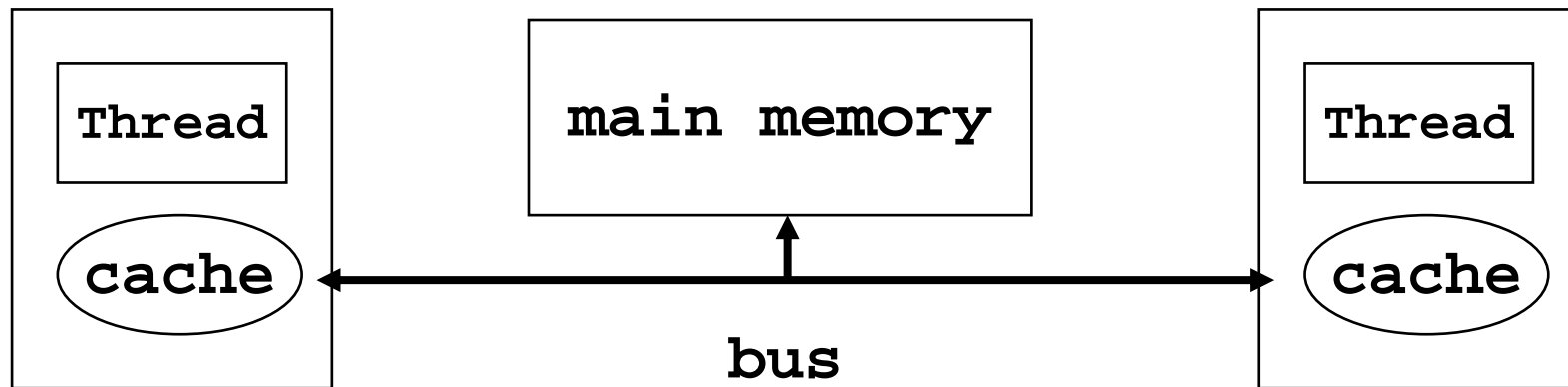
Java Memory Model (JMM)

- specifies minimal guarantees given by the JVM
 - about when writes to variables become visible to other threads
- is an abstraction on top of hardware memory models



Java memory model

- JMM resembles abstract SMP (symmetric multi processing) machine
- key ideas:
 - all threads share the main memory
 - each thread uses a local working memory
 - flushing or refreshing working memory to/from main memory must comply to JMM rules



Java memory model

JMM rules address 3 intertwined issues:

- atomicity
 - which operations must have indivisible effects ?
- visibility
 - under which conditions are the effects of operations taken by one thread visible to other threads ?
- ordering
 - under which conditions can the effects of operations appear out of order to any given thread ?

"operations" means:

- reads and writes to memory cells representing Java variables

hardware memory models

- JVM maps JMM to hardware memory model
 - in shared-memory multiprocessor architectures:
 - each processor (or processor core) has its own cache (or even several layers of caches)
 - cache is periodically reconciled with main memory
 - cache strategies vary among architectures

=> *hardware memory model*

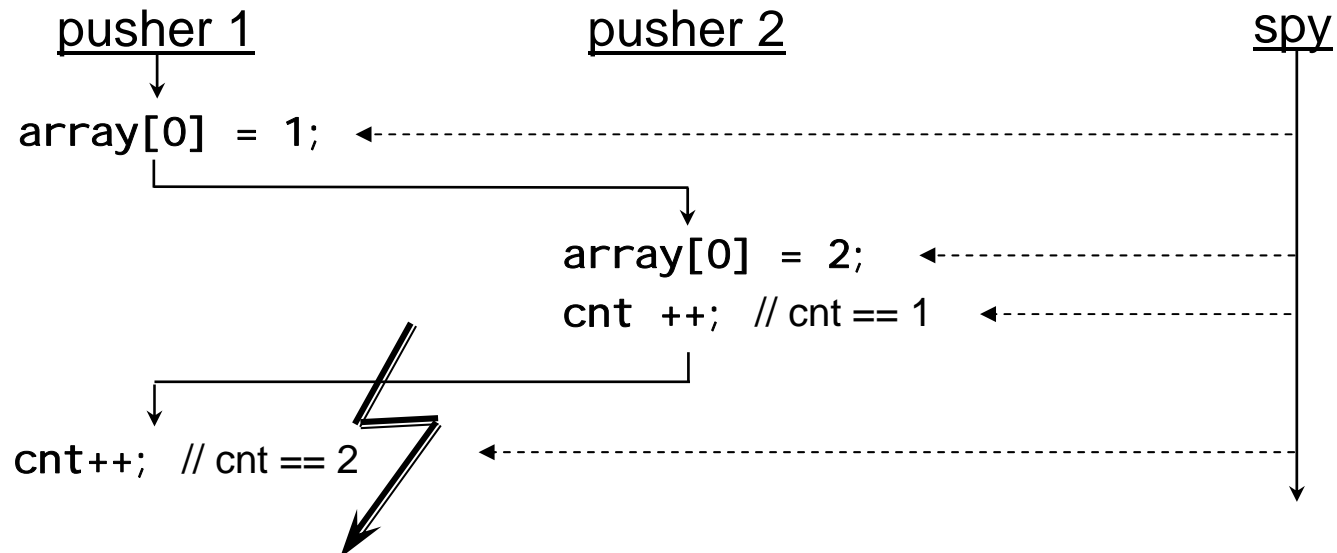
- *memory barriers or fences*
 - JVM uses special instructions for memory coordination
 - to implement the JMM rules
 - to shield developers from hardware differences

sequential consistency

- *sequential consistency* is a convenient (yet unrealistic) mental model:
 - imagine a single order of execution of all programm operations (regardless of the processors used)
 - each read of a variable will see the last write in the execution order

sequential consistency - example

```
private int[] array;  
private int cnt = 0;  
...  
public void push(int elm) { array[cnt++] = elm; }  
public int pop()          { return(array[--cnt]); }  
...
```

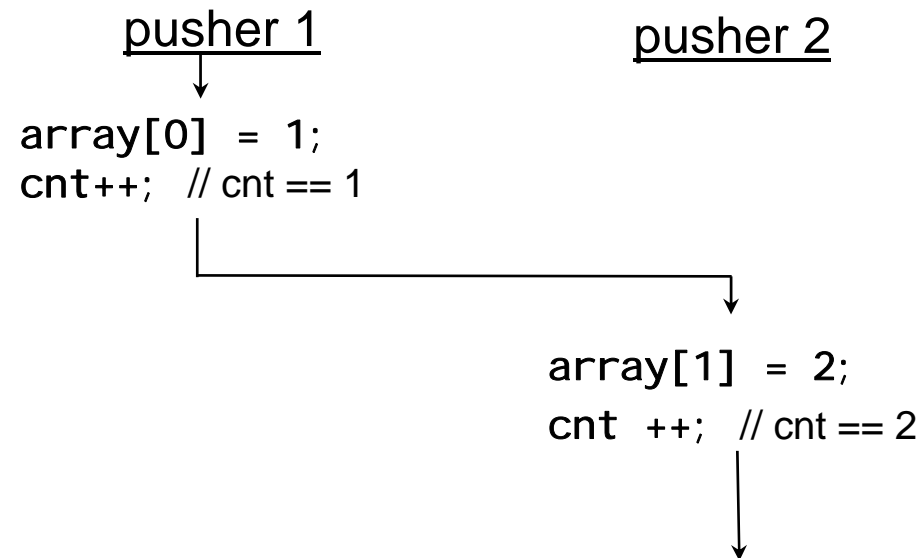


NO sequential consistency

- JMM does NOT guarantee sequential consistency
 - reordering is generally permitted
 - specific rules for
 - synchronization,
 - thread start / join,
 - volatile variables, and
 - final fields

memory effects of synchronisation

```
private int[] array;  
private int cnt = 0;  
...  
public synchronized void push(int elm) { array[cnt++] = elm; }  
public synchronized int pop()          { return(array[--cnt]); }  
...
```

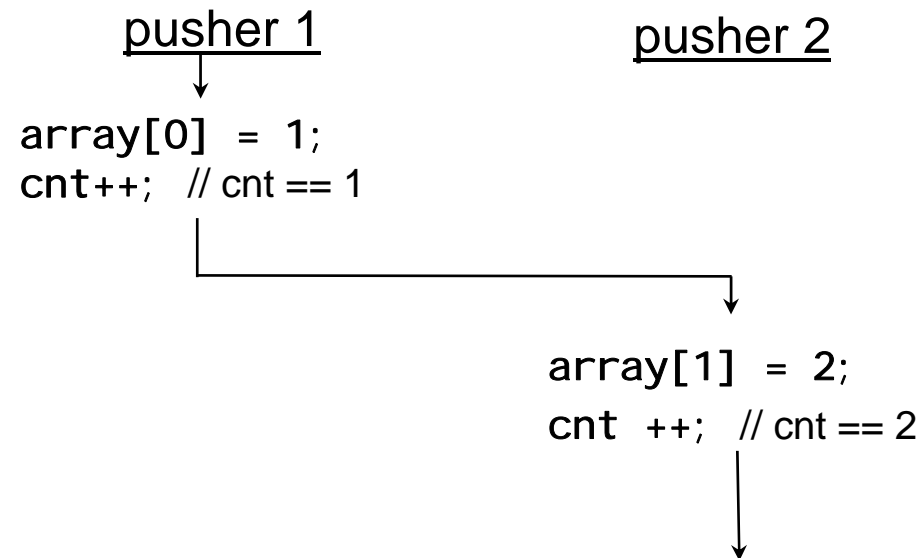


agenda

- history of concurrency & concurrency trends
- **synchronization and memory model**
 - **atomicity**
 - visibility
 - ordering
- fight the serialization – improve scalability
- future trends

atomicity & synchronization

- execution of operations in a synchronized block appears atomic to other synchronized blocks (using the same lock)
- same for operations between acquisition / release of explicit lock



atomicity of unsynchronized access

- access to **primitive** type (except long/double) is atomic
- access to **references** is atomic (does *not* include object access)
- access to **volatile** variables (including long/double) is atomic
- access to **atomic** variables is atomic

- common misconception
 - atomicity means we get the most recent value - wrong!
 - atomic access to a variable just means:
 - we will *not* get some jumble of bits

agenda

- history of concurrency & concurrency trends
- synchronization and memory model
 - atomicity
 - **visibility**
 - ordering
- fight the serialization – improve scalability
- future trends

need for visibility

```
private int[] array;
private int cnt = 0; ← must be volatile
...
public synchronized void push(int elm) { array[cnt++] = elm; }
public synchronized int pop()          { return(array[--cnt]); }
public                  int size()      { return cnt; }
...
```

- access to cnt is atomic
 - no synchronization in size() needed
- visibility problem
 - writes performed in one thread need not be visible to other threads
 - i.e. modification of cnt in push()/pop() need not be visible to size()
- volatile is needed not for atomicity, but for visibility

visibility issues

- visibility not a problem *within* the same thread
 - when values are passed across methods in the same thread
- visibility is a problem *across* threads
 - not even any guarantees for related fields
 - example:
 - object has two fields + both are altered by thread A
 - thread B might see:
 - one field has initial value and
 - other field has updated value

visibility guarantees

- changes made in one thread are guaranteed to be visible to other threads under the following conditions:
 - explicit synchronization
 - thread start and termination
 - read / write of volatiles
 - first read of finals
- note:
 - visibility is only an issue in case of unsynchronized access

explicit synchronization

- releasing a lock forces a flush
 - the releasing thread flushes its working memory to main memory
- acquiring a lock forces a reload from main memory
 - the acquiring thread refreshes its working memory from main memory
- matches our expectation
 - at a synchronization point the effect produced by one thread become visible to the other thread

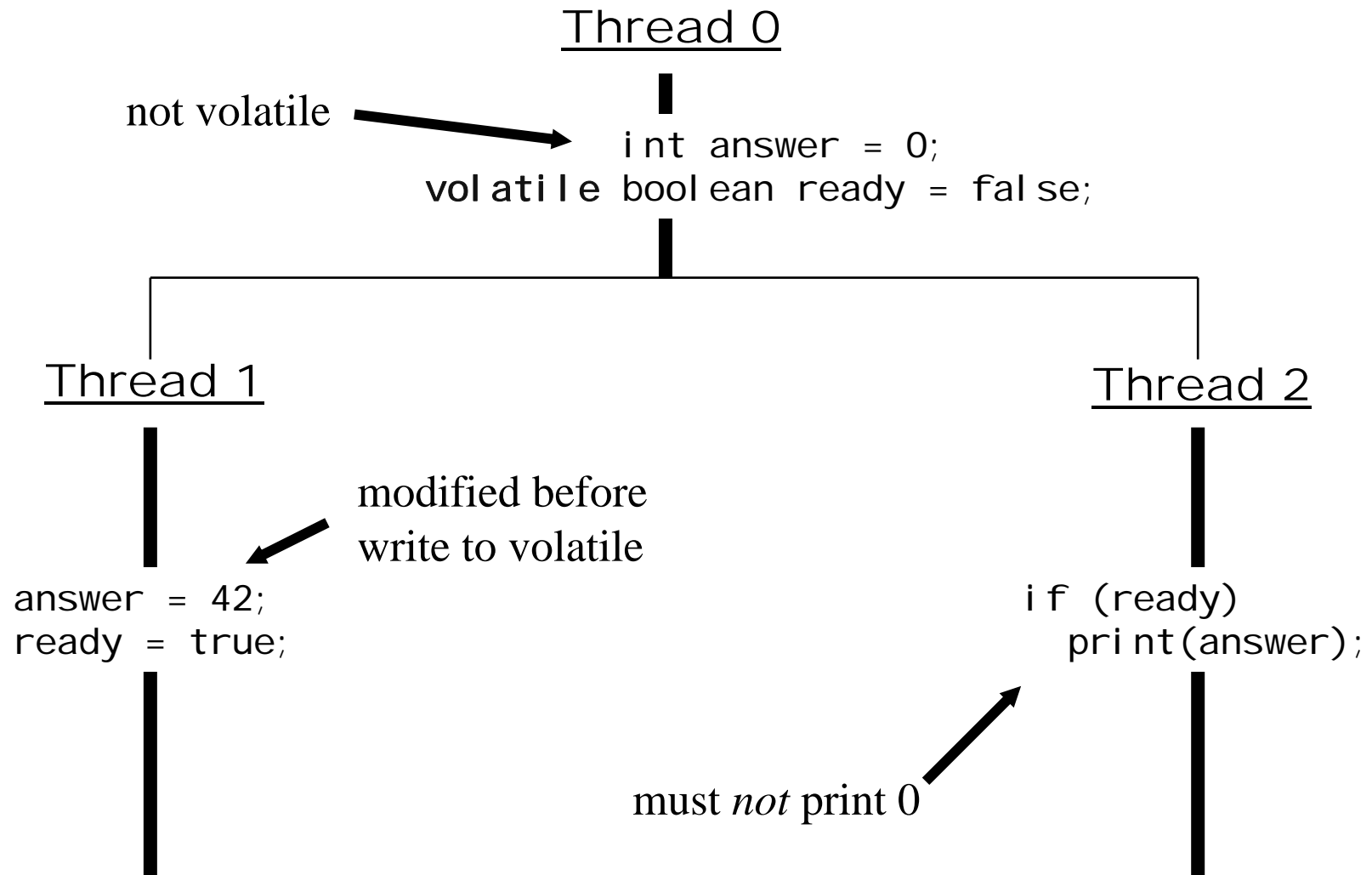
thread start and termination

- thread start forces a reload from main memory
 - starting thread fills working memory from main memory
- thread termination forces a flush to main memory
 - terminating thread flushes working memory to main memory
- matches our expectation
 - `join()` is guaranteed to see the effects produced by the other thread
 - how about `sleep()` and `yield()`, `wait()` and `notify()` ?

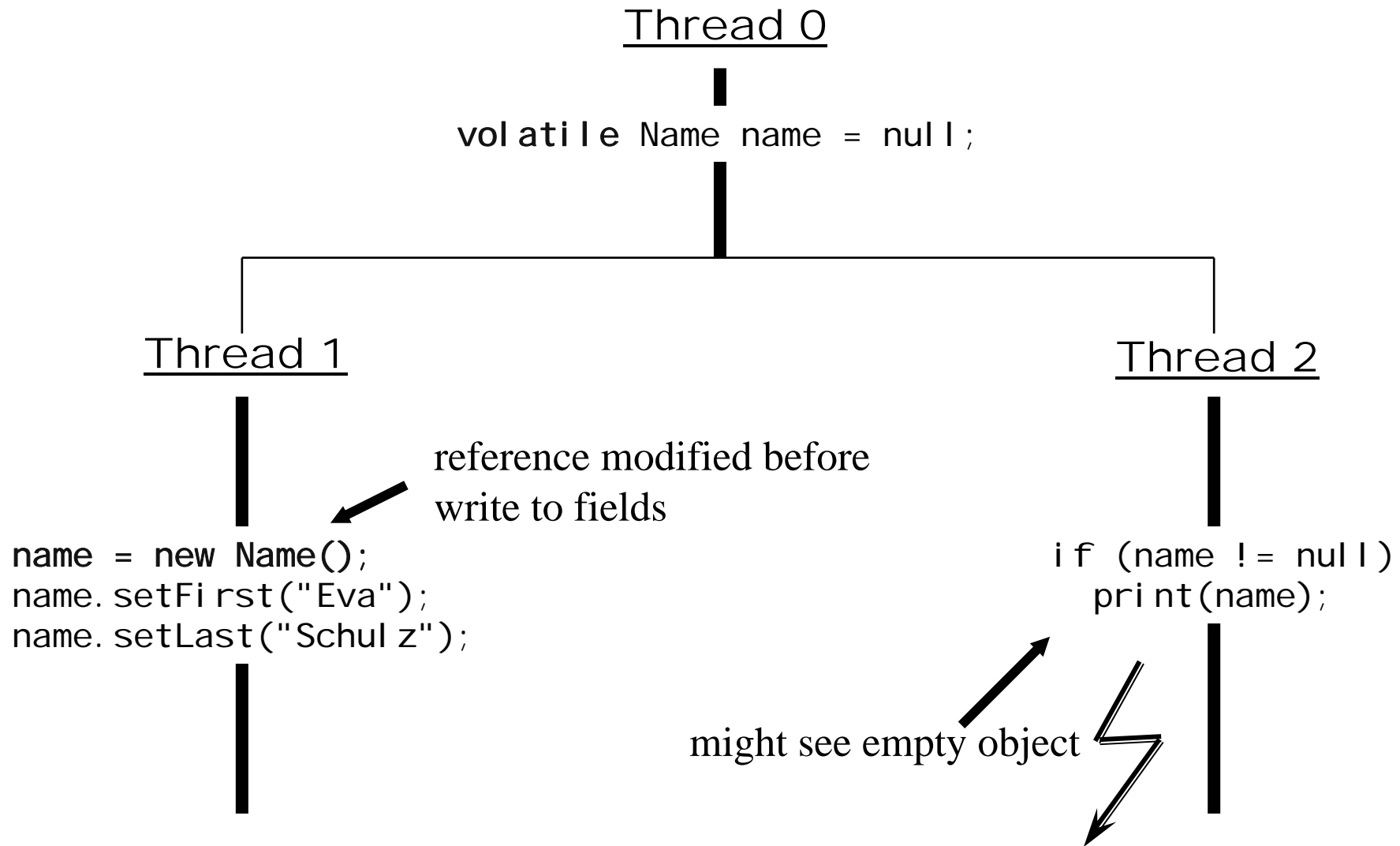
read / write of volatiles

- reading a volatile forces a reload from main memory
 - the reading thread fills its working memory from main memory
- writing to a volatile forces a flush to main memory
 - the writing thread flushes its working memory to main memory
- matches our expectation
 - modifications of a volatile field produced by one thread are visible to other threads
 - ... what does that mean for volatile references ... ???

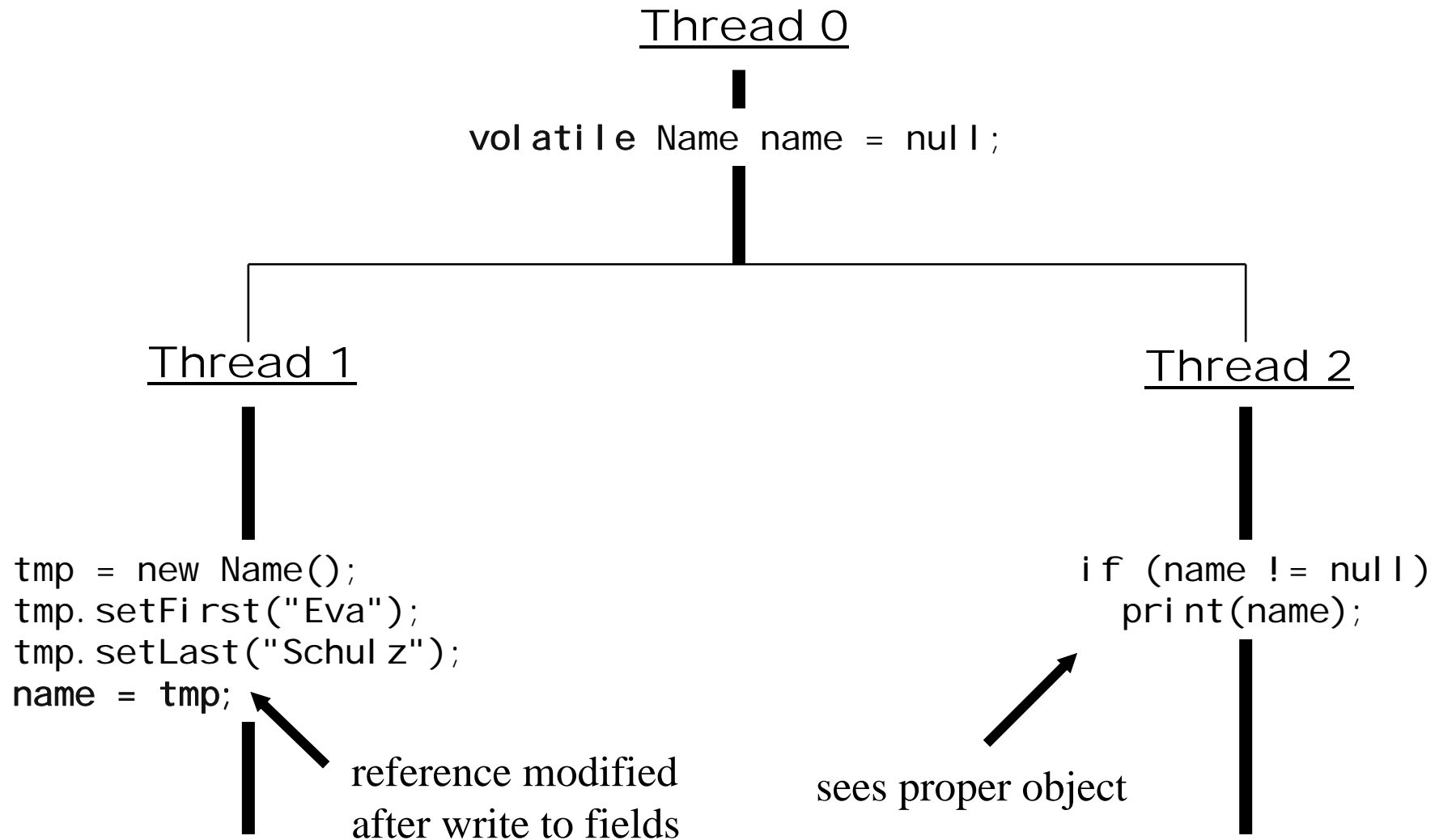
volatile (since Java 5) - example



non-transitive volatile - example



volatile reference - example



volatile references

- what do we do to also make the modified object visible?
 - make all fields of referenced object volatile
 - problem for arrays: array elements cannot be declared volatile
 - modify elements before assignment to volatile reference
 - all changes made prior to writing to the volatile are flushed
 - use explicit synchronization
 - viable fallback, at the expense of synchronization overhead

first read of final

- end of construction forces a partial flush
 - the constructing thread flushes any finals (including any dependent values) from its working memory to main memory
 - final is transitive
- first access of a final field forces a partial reload
 - the reading thread loads the final (including any dependent values) from main memory
 - afterwards the values need never be refreshed again, since they are constant

agenda

- history of concurrency & concurrency trends
- synchronization and memory model
 - atomicity
 - visibility
 - **ordering**
- fight the serialization – improve scalability
- future trends

ordering

- ordering is closely related to visibility
- JMM is specified in terms of *actions*
 - e.g. reads and writes to variables, locks and unlocks of monitors, starting and joining threads
- JMM defines "*happens-before*" rules
 - **partial** ordering on actions
 - if there is **NO** happens-before ordering between two actions, then the JVM is free to reorder them
- common misconception: a "happens-after" rule
 - e.g. an action after a synchronized block can happen before or in the critical section

reordering problem - example

No!

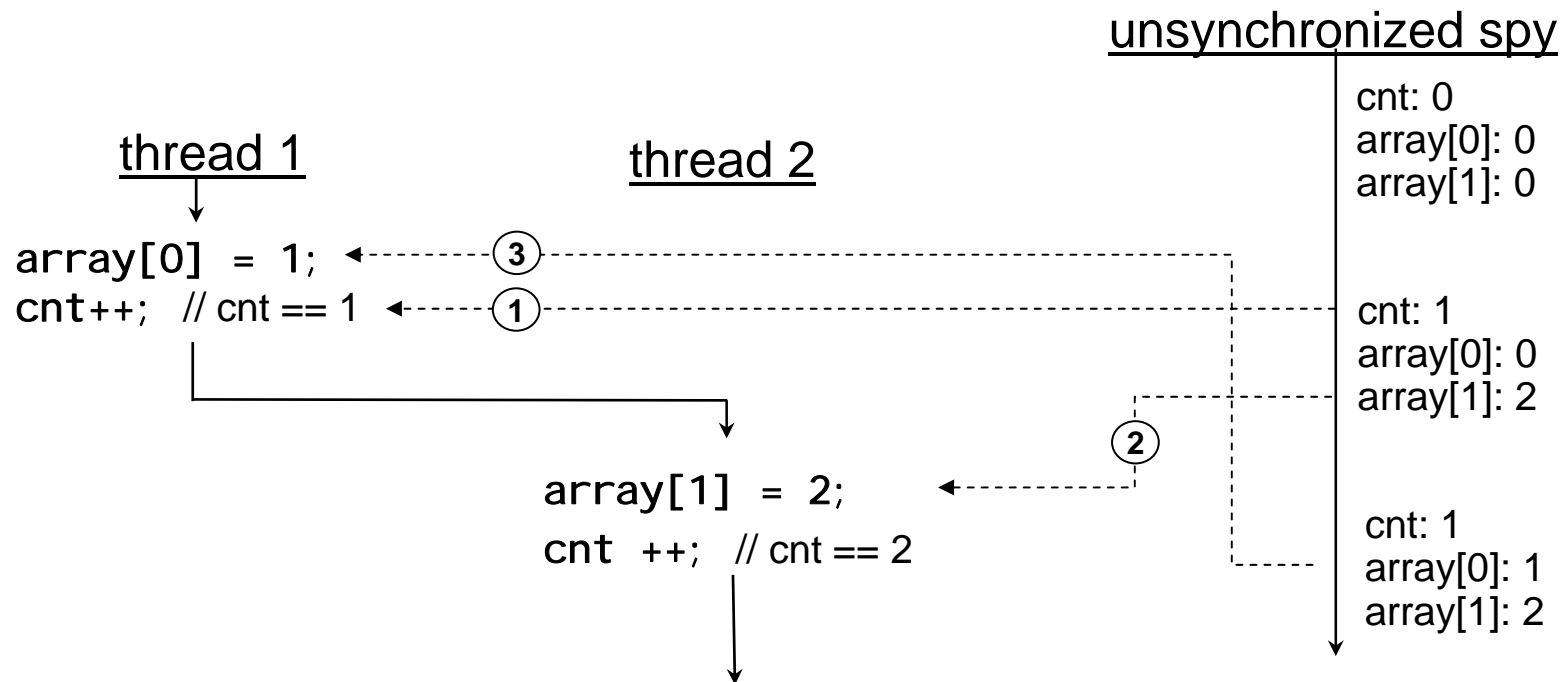
```
public class PossibleReordering {
    private static int x = 0, y = 0;
    private static int a = 0, b = 0;

    public static void main(String[] args) {
        Thread one = new Thread(new Runnable() {
            public void run() { a = 1; x = b; }
        });
        Thread two = new Thread(new Runnable() {
            public void run() { b = 1; y = a; }
        });
        one.start(); two.start(); one.join(); two.join();
        System.out.println("x="+x+", y="+y);
    }
}
```

- possible reordering
 - result is unpredictable; even $x=0$ and $y=0$ can happen

reordering - example

```
private int[] array;  
private int cnt = 0;  
...  
public synchronized void push(int elm) { array[cnt++] = elm; }  
public synchronized int pop()         { return(array[--cnt]); }  
...
```



ordering

- ordering rules have two aspects:
- within thread
 - thread performing actions in a method perceives instructions in normal **as-if-serial** order
- between thread
 - other thread ‘spying’ on this thread (without synchronization) might perceive instructions in **arbitrary** order

ordering guarantees

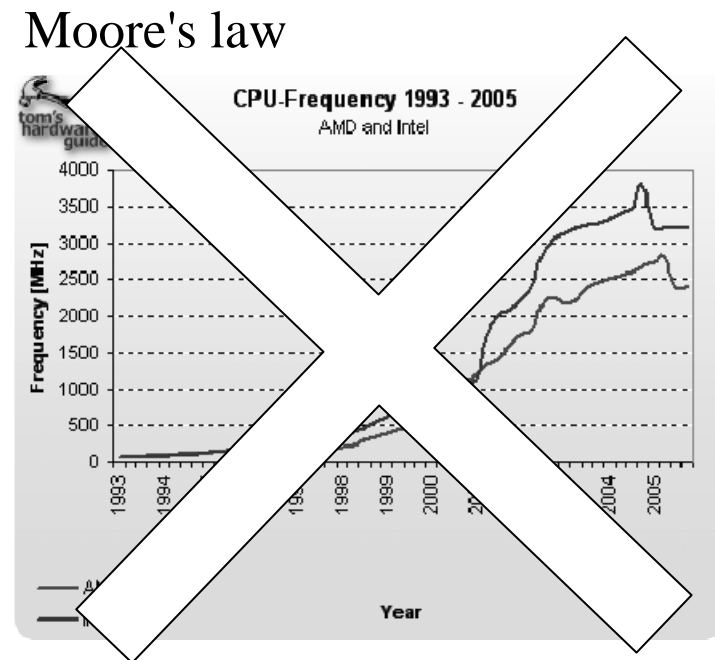
- ordering of synchronized blocks is preserved
 - actions in one synchronized block happen (i.e. effects become visible) *before* another thread acquires the same lock
- ordering of read/write of volatile fields is preserved
 - effect of last write to a volatile is visible to all subsequent reads of the same volatile
- ordering of initialization/access to final fields is preserved
 - all threads will see the correct values of final fields that were set by the constructor

agenda

- history of concurrency & concurrency trends
- synchronization and memory model
- **fight the serialization – improve scalability**
- future trends

performance wise

- cannot buy a faster CPU to speed up the program
 - or hope for a faster CPU six/twelve month from now
 - when you program feels slow during development
- software must be designed so that
 - it can take advantage of the additional cores / CPUs
 - can scale with additional cores / CPUs

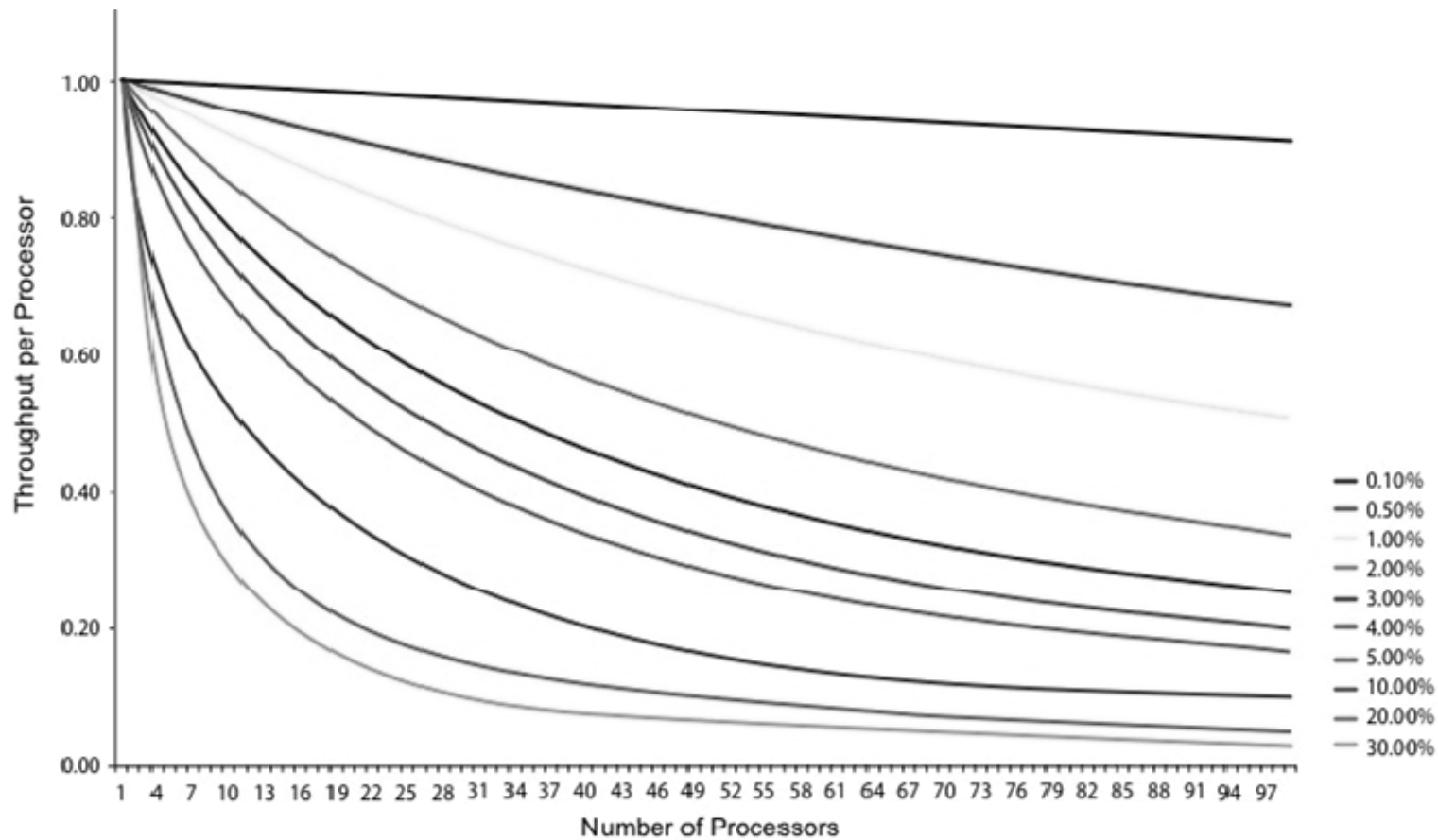


Amdahl's law

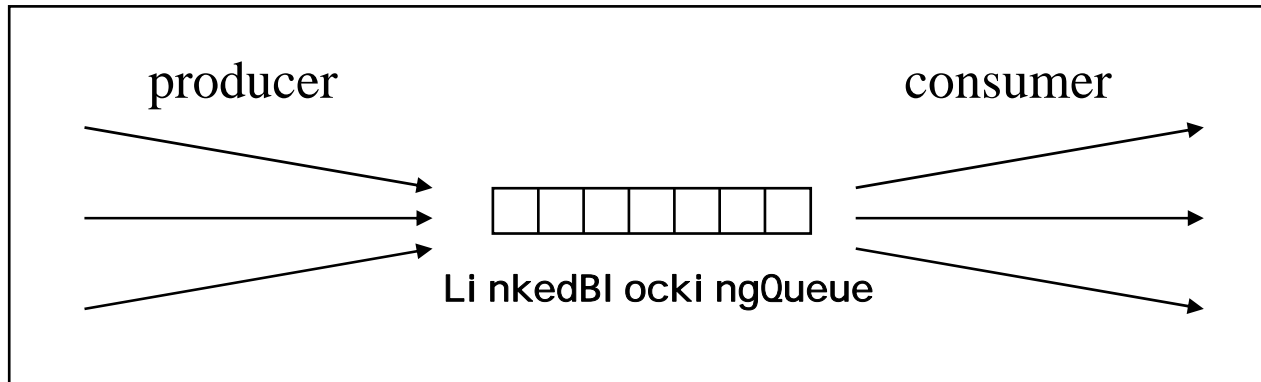
- idea: divide work into **serial** and **parallel** portions
 - serial work cannot be sped up by adding resources
 - parallelizable work can
- Amdahl's Law: $\text{speedup} \leq \left(F + \frac{1 - F}{N} \right)$
 - F is the fraction that must be serialized
 - N is the number of CPUs
- with $N \rightarrow \infty$, $\text{speedup} \rightarrow 1/F$
 - with 50% serialization,
 - your program can only speed up by a factor of 2 (with: ∞ CPUs)
- naïve idea: from 1 to 2 CPUs = factor of 2 ?

(cont.)

- fight serialization to improve performance



example



- looks highly parallelizable
 - (if producers are slow increase their thread pool)
- 0% serialized ?
 - no!
 - need synchronization to maintain the queue's integrity

LinkedListQueue.offer()

```
public boolean offer(E o) {
    if (o == null) throw new NullPointerException();
    final AtomicInteger count = this.count;
    if (count.get() == capacity)
        return false;
    int c = -1;
    final ReentrantLock putLock = this.putLock;
    putLock.lock();
    try {
        if (count.get() < capacity) {
            insert(o);
            c = count.getAndIncrement();
            if (c + 1 < capacity)
                notFull.signal();
        }
    } finally {
        putLock.unlock();
    }
    if (c == 0)
        signalNotEmpty();
    return c >= 0;
}
```

(cont.)

- Doug Lea did an excellent job with the implementation
 - highly optimized
 - split lock: put / take
 - `count` guarded lock-free
 - stack-local variables to speed up the execution inside the critical region
 - ...
- structural problem
 - serialization of offering threads (producers)
 - similar serialization of getting threads (consumers)

serialization

- where/when threads demand concurrent access
- often hidden
 - in frameworks / third party abstractions
- other area: asynchronous service architecture
 - example: `java.nio.channels.Selector`
 - section on concurrency in the respective JavaDoc
 - management to send back the result asynchronously
 - Jetty continuation

fight the serialization ...

... try to reduce lock induced serialization

- smallest critical region possible
 - synchronized block vs. synchronized method
 - or use explicit locks
 - speed up execution inside the critical region
 - replace synchronized counters with `AtomicInteger`
- lock splitting / striping
 - guard different state with different locks
 - reduces likelihood of lock contention

fight the serialization ...

... try to eliminate locking entirely

- replace mutable objects with immutable ones
- replace shared objects with thread-local ones
 - e.g. make a copy before passing it to a concurrent thread
- lock-free programming

agenda

- history of concurrency & concurrency trends
- synchronization and memory model
- fight the serialization – improve scalability
- **future trends**

trends

- lock-free programming
 - supported in Java since JDK 5.0
 - `java.util.concurrent.atomic`, and
 - Concurrent collections in `java.util.concurrent`
- transactional memory
 - neither supported in Java nor in any popular programming language at the moment
- commonality
 - avoid locking to avoid serialization

wrap-up

- a trend towards concurrent, asynchronous computing
 - MT initially for better structure
 - today to overcome synchronicity (messaging, AJAX, ...)
- multicore architecture might reveal yet undetected bugs
 - due to memory model issues (atomicity, visibility, ordering)
- multicore architectures need scalable software to be useful
 - avoid serialization - increase concurrency - Amdahl's law
- a gaze into the crystal ball
 - lock-free programming is already in use (by experts)
 - transactional memory might ease concurrent programming some time in the future

authors

Angelika Langer

Training & Consulting

Klaus Kreft

SEN Group, Munich, Germany

<http://www.AngelikaLanger.com/Forms/Contact.html>

Java Programming in a Multicore World

Q & A