Concurrency & Memory Models

Yu Zhang

Course web site: http://staff.ustc.edu.cn/~yuzhang/pldpa

Parallelism & Concurrency

Parallelism vs. Concurrency

- A parallel program exploits *real* parallel computing resources to *run faster* while computing the *same answer*.
  - Expectation of genuinely simultaneous execution
  - Deterministic
- A concurrent program models independent agents that can communicate and synchronize.
  - Meaningful on a machine with one processor
  - Non-deterministic

The Promise of Concurrency

- Speed
  - If a task takes time *t* on one processor, shouldn't it take time t/n on n processors?
- Availability
  - If one process is busy, another may be ready to help
- Distribution
  - Processors in different locations can collaborate to solve a problem or work together
- Applications
  - Vision, cognition etc. appear to be highly parallel activities

Concurrency on Machines

- Multiprogramming
  - A single computer runs several programs at the same time
  - Each program proceeds sequentially
  - Actions of one program may occur between two steps of another
- Multiprocessors
  - Two or more processors may be connected
  - Programs on one processor communicate with programs on another
  - Actions may happen simultaneously

The Grand Challenge

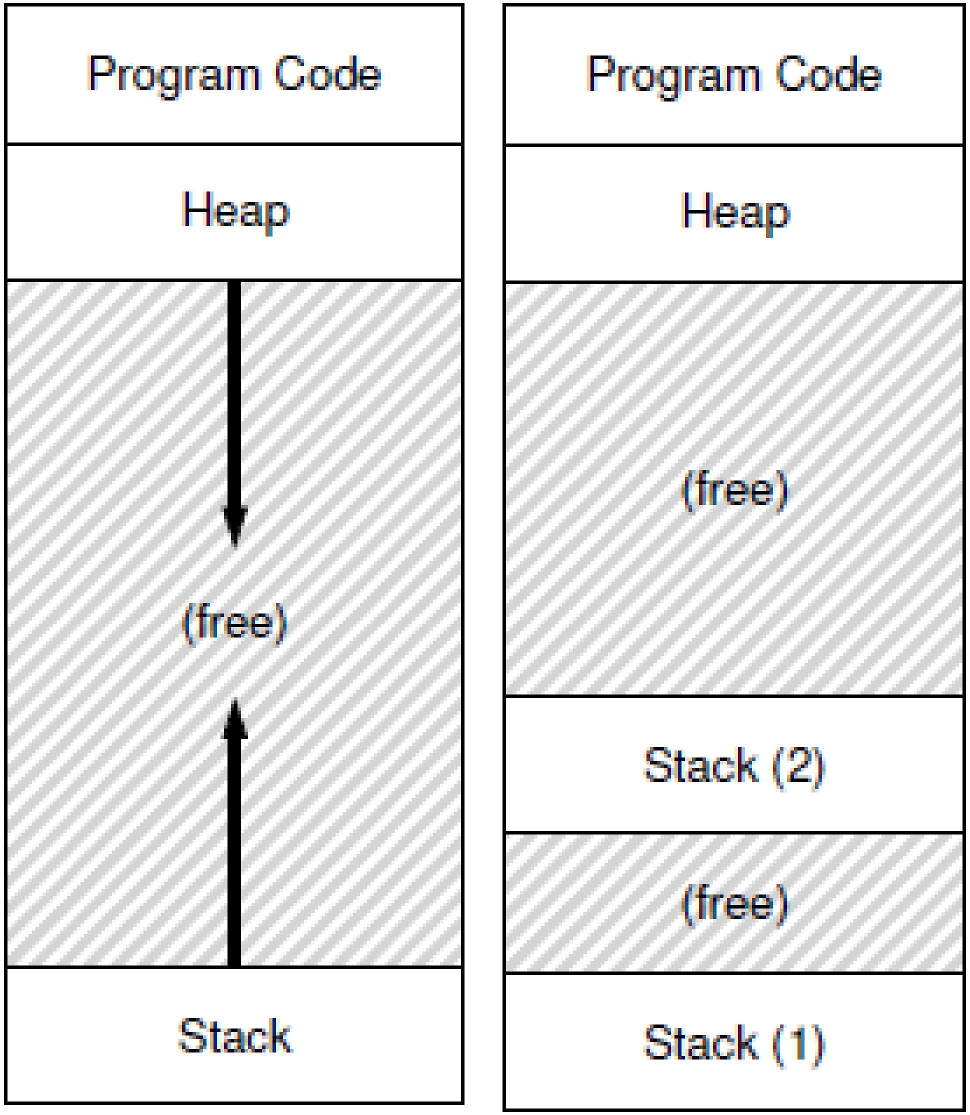
- Making effective use of multi-core hardware is the challenge for programming languages now.
- Hardware is getting increasingly complicated:
  - Nested memory hierarchies
  - Hybrid processors: GPU + CPU, FPGA...
  - Massive compute power sitting mostly idle.
- Need new programming models to program new commodity machines effectively.

Challenges

- Concurrent programs are harder to get right
- Some problems are inherently sequential
- Specific issues
  - Communication: send or receive information
  - Synchronization: wait for another process to act
  - Atomicity: do not stop in the middle and leave a mess

Thread

- A multi-threaded program
  - has multiple PCs (program counter)
  - Threads share the same address space
  - Each thread has its own thread control block (TCB) to store its state (e.g. register state), and its own stack (thread-local storage)



Why Use Threads

- Parallelism
  - Use a thread per CPU to do a work on multiple CPUs
- Avoid blocking program progress due to low I/O
  - Threading enables overlap of I/O with other activities within a single program
    - much like multiprogramming did for processes across programs

Simple Thread Creation C Code

Many possible execution orderings!

```c
#include <stdio.h>
#include <assert.h>
#include <pthread.h>

void *mythread(void *arg) {
    printf("%s\n", (char *) arg);
    return NULL;
}

int
main(int argc, char *argv[]) {
    pthread_t p1, p2;
    int rc;
    printf("main: begin\n");
    rc = pthread_create(&p1, NULL, mythread, "A"); assert(rc == 0);
    rc = pthread_create(&p2, NULL, mythread, "B"); assert(rc == 0);
    // join waits for the threads to finish
    rc = pthread_join(p1, NULL); assert(rc == 0);
    rc = pthread_join(p2, NULL); assert(rc == 0);
    printf("main: end\n");
    return 0;
}
```

Why it Gets Worse: Shared Data

```
static volatile int counter = 0;
void *
mythread(void *arg)
{
    printf("%s: begin\n", (char *) arg);
    int i;
    for (i = 0; i < 1e7; i++) {
        counter = counter + 1;
    }
    printf("%s: done\n", (char *) arg);
    return NULL;
}
```

- Two threads perform mythread()
  - Add a number to the shared counter, and do so 1e7 in a loop

What's the result? counter == 20000000?

Yield different and nondeterministic results for different runs!

Problem: Uncontrolled Scheduling

- Understand the low-level code
  - gcc -g to produce instructions including symbol info.
  - objdump -d main to see the assembly code

counter = counter + 1

Race condition:
results depend on the timing execution of the code

T1:

```
mov 0x8049a1c, %eax

add $0x1, %eax

mov %eax, 0x8049a1c
```

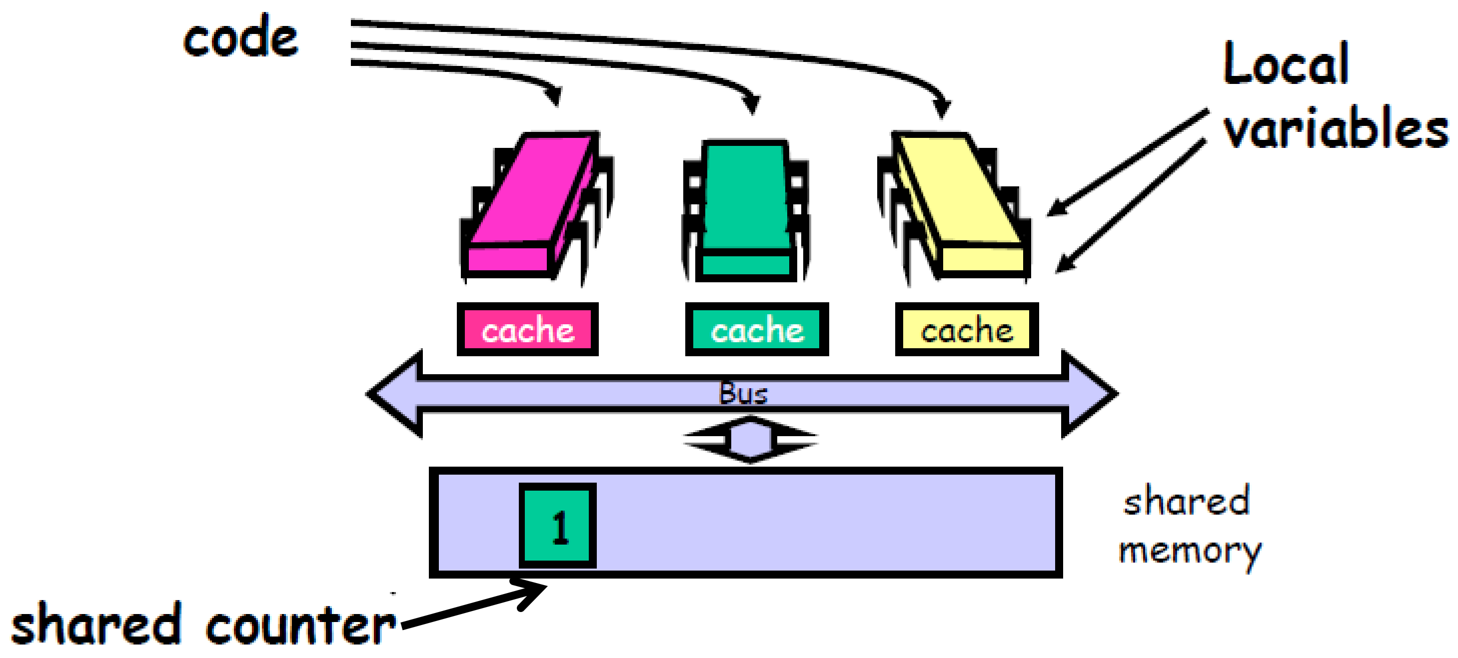
T2:

```
mov 0x8049a1c, %eax
add $0x1, %eax

mov %eax, 0x8049a1c
```

Problem: Uncontrolled Scheduling

- Sudden unpredictable delays
  - Cache misses (short)
  - Page faults (long)
  - Scheduling quantum used up (really long)



Wish for Atomicity

- Atomicity
  - Execute an instruction sequence as a unit, “all or none”
  - Method 1: use atomic instruction
  - Method 2: use atomic block supported by transaction memory (TM) system
- Synchronization
  - Critical section
    - Access shared resource, only one process/thread in the section
  - lock … unlock
    - Problem: deadlock

```
pthread_mutex_t mutex;
 ...
pthread_mutex_lock(&mutex);
counter = counter + 1;
pthread_mutex_unlock(&mutex);
```

Locks

- Mutual exclusion
  - Deadlock-free, Fairness (lock starve?), performance
- Locking strategies:
  - Coarse-grained
  - Fine-grained: protect different data with different locks
- How to build a lock?
  - Hardware primitives
  - OS support

Peterson’s Algorithm [1981]

```
int flag[2];
int turn;
void init() {
        flag[0] = flag[1] = 0;     // 1->thread wants to grab lock
        turn = 0;                  // whose turn? (thread 0 or 1?)
}
void lock() {
        flag[self] = 1;            // self: thread ID of caller
        turn = 1 - self;           // make it other thread’s turn
        while ((flag[1-self] == 1) && (turn == 1 - self))
                ;                  // spin-wait
}
void unlock() {
        flag[self] = 0;            // simply undo your intent
}
```

Mutual Exclusive Primitives

- Atomic test-and-set
  - Instruction atomically reads and writes some location
  - Common hardware instruction
  - Used to implement a busy-waiting loop to get mutual exclusion
- Semaphore
  - Avoid busy-waiting loop
  - Keep queue of waiting processes
  - Scheduler has access to semaphore, process sleeps
  - Disable interrupts during semaphore operations
    - OK since operations are short

State of the Art

- Concurrent programming is difficult
  - Race conditions, deadlock are pervasive
- Languages should be able to help
  - Capture useful paradigms, patterns, abstractions
    - Concurrent data structures
    - Parallel pattern: fork-join, pipeline, data parallelism, MapReduce, ...
- Other tools are needed
  - Testing is difficult for multi-threaded programs
    - Record-replay
    - Deterministic multi-threading execution

State of the Art

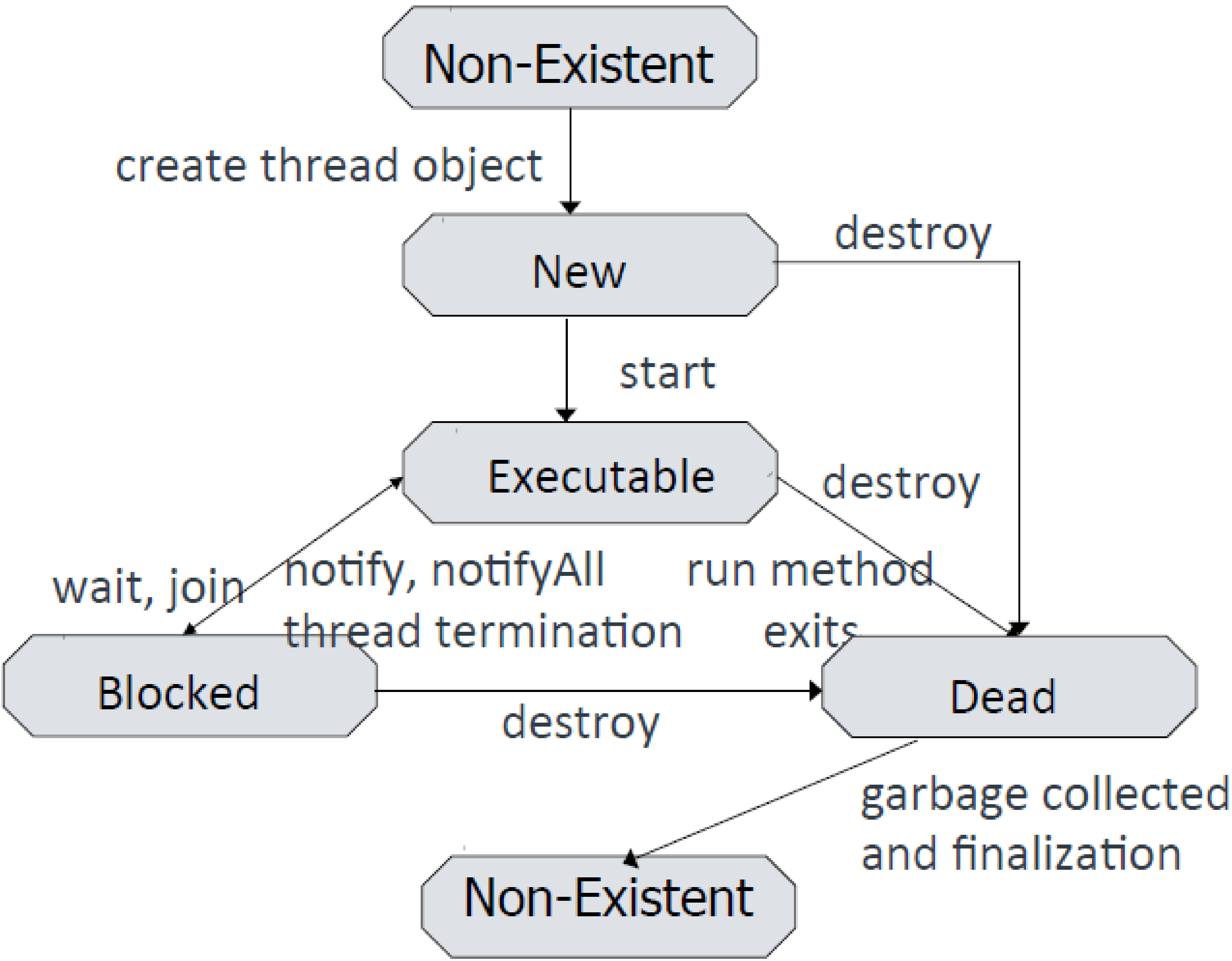
- Other tools are needed
  - Testing is difficult for multi-threaded programs
  - Many race-condition detectors being built today
    - Static detection: conservative, may be too restrictive
      - LockSmith [TOPLAS, 33(1), 2011], Jeffrey S. Foster
    - Run-time detection: may be more practical for now
      - FastTrack [PLDI 2009], Cormac Flanagan and Stephen N. Freund
    - Kernel
      - DataCollider [OSDI 2010] , Microsoft

Java Concurrency

- Threads
- Communication
  - Shared variables
  - Method calls
- Mutual exclusion and synchronization
  - Every object has a lock (inherited from class Object)
    - Synchronized methods and blocks
  - Synchronization operations(inherited from class Object)
    - wait
    - notify

```
public class Counter {
    private long value;
    public long getAndIncrement() {
        synchronized {
            temp = value;
            value = temp + 1;
        }
        return temp;
    }
}
```

Java Thread States



Interaction Between Threads

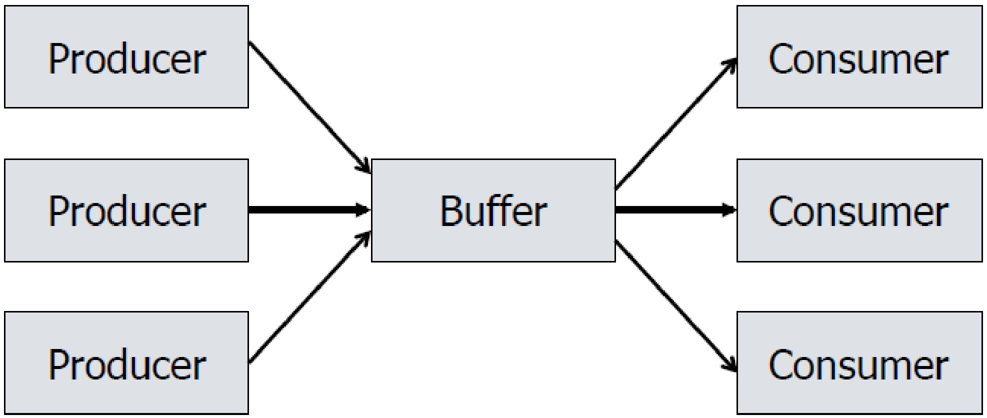
- Shared variables
  - Two threads may assign/read the same variable
  - Programmer responsibility
    - Avoid race conditions by explicit synchronization
- Method calls
  - Two threads may call methods on the same object
- Synchronization primitives
  - Each object has internal lock, inherited from Object
  - Synchronization primitives based on object locking

Synchronized Methods

```
class LinkedCell {         // Lisp-style cons cell containing
   protected double value;  // value and link to next cell
   protected final LinkedCell next;
   public LinkedCell (double v, LinkedCell t) {
            value = v; next = t;
   }
   public synchronized double getValue() {
            return value;
   }
   public synchronized void setValue(double v) {
            value = v;   // assignment not atomic
   }
   public LinkedCell next() {   // no synch needed
            return next;
   }
```

Stack<T>: produce,consume Methods

```
public synchronized void produce (T object) {
    stack.add(object);
    notify();
}
public synchronized T consume () {
    while (stack.isEmpty()) {
        try {
            wait();
        } catch (InterruptedExcepZon e) {  }
     }
     Int lastElement = stack.size() - 1;
    T object = stack.get(lastElement);
    stack.remove(lastElement);
    return object;
}
```



Wait-notify

http://www1.coe.neu.edu/~kokar/java/tut.html

Rust

- [16. Fearless Concurrency]()

```
use std::sync::Mutex;
use std::thread;
fn main() {
    let counter = Arc::new(Mutex::new(0));
    let mut handles = vec![];
    for _ in 0..10 {
        let counter = Arc::clone(&counter);
        let handle = thread::spawn( move || {
            let mut num = counter.lock().unwrap();
            *num += 1;
        });
        handles.push(handle);
    }
   for handle in handles {
        handle.join().unwrap();
    }
    println!("Result: {}", *counter.lock().unwrap());
}
```

Arc<T>

atomic reference counting

Rust: produce - consume

```
use std::thread;
use std::sync::mpsc;
fn main() {
    let (tx, rx) = mpsc::channel();
    thread::spawn(move || {
        let val = String::from("hi");
        tx.send(val).unwrap();
      });
    let received = rx.recv().unwrap();
    println!("Got: {}", received);
}
```

Thread Safety

- Concept
  - The fields of an object or class always maintain a valid state, as observed by other objects and classes, even when used concurrently by multiple threads
- Why is this important?
  - Each method preserves state invariants
  - Invariants hold on method entry and exit
  - What’s “valid state”? Serializability …

Example

```
public class RGBColor {
   private int r;    private int g;     private int b;
   public RGBColor(int r, int g, int b) {
      checkRGBVals(r, g, b);
      this.r = r;     this.g = g;     this.b = b;
   }

   ...

 private static void checkRGBVals(int r, int g, int b) {
      if (r < 0 || r > 255 || g < 0 || g > 255 ||
         b < 0 || b > 255) {
         throw new IllegalArgumentException();
      }
   }
}
```

Example

```
public class RGBColor {
  private int r;   private int g;    private int b;
  public RGBColor(int r, int g, int b) {
    checkRGBVals(r, g, b);
    this.r = r;    this.g = g;    this.b = b;
  }

  ...

 private static void checkRGBVals(int r, int g, int b) {
    if (r < 0 || r > 255 || g < 0 || g > 255 ||
      b < 0 || b > 255) {
      throw new IllegalArgumentException();
    }
  }
}
```

```
public void setColor(int r, int g, int b) {
    checkRGBVals(r, g, b);
    this.r = r;    this.g = g;     this.b = b;
  }

public int[] getColor() {   //  returns array of three ints: R, G, and B
    int[] retVal = new int[3];
    retVal[0] = r;    retVal[1] = g;   retVal[2] = b;
    return retVal;
  }

  public void invert() {
    r = 255 - r;     g = 255 - g;     b = 255 - b;
  }
```

Question: what goes wrong with multi-threaded use of this class?

Some Issues with RGB Class

- Read/Write conflicts
  - If one thread reads while another writes, the color that is read may not match the color before or after
- Write/write conflicts
  - It two threads try to write different colors, result may be a “mix” of R,G,B from two different colors.

How to Make Classes Thread-safe

- Synchronize critical sections
  - Make fields private
  - Synchronize sections that should not run concurrently
- Make objects immutable
  - State cannot be changed after object is created
  - Use pure functional programming for concurrency
- Use a thread-safe wrapper
  - New thread-safe class has objects of original class as fields
  - Wrapper class provides methods to access original class object

Thread-safe Wrapper

```
public synchronized void setColor(int r, int g, int b) {
    color.setColor(r, g, b);
}
public synchronized int[] getColor() {
    return color.getColor();
}
public synchronized void invert() {
    color.invert();
}
```

Comparison

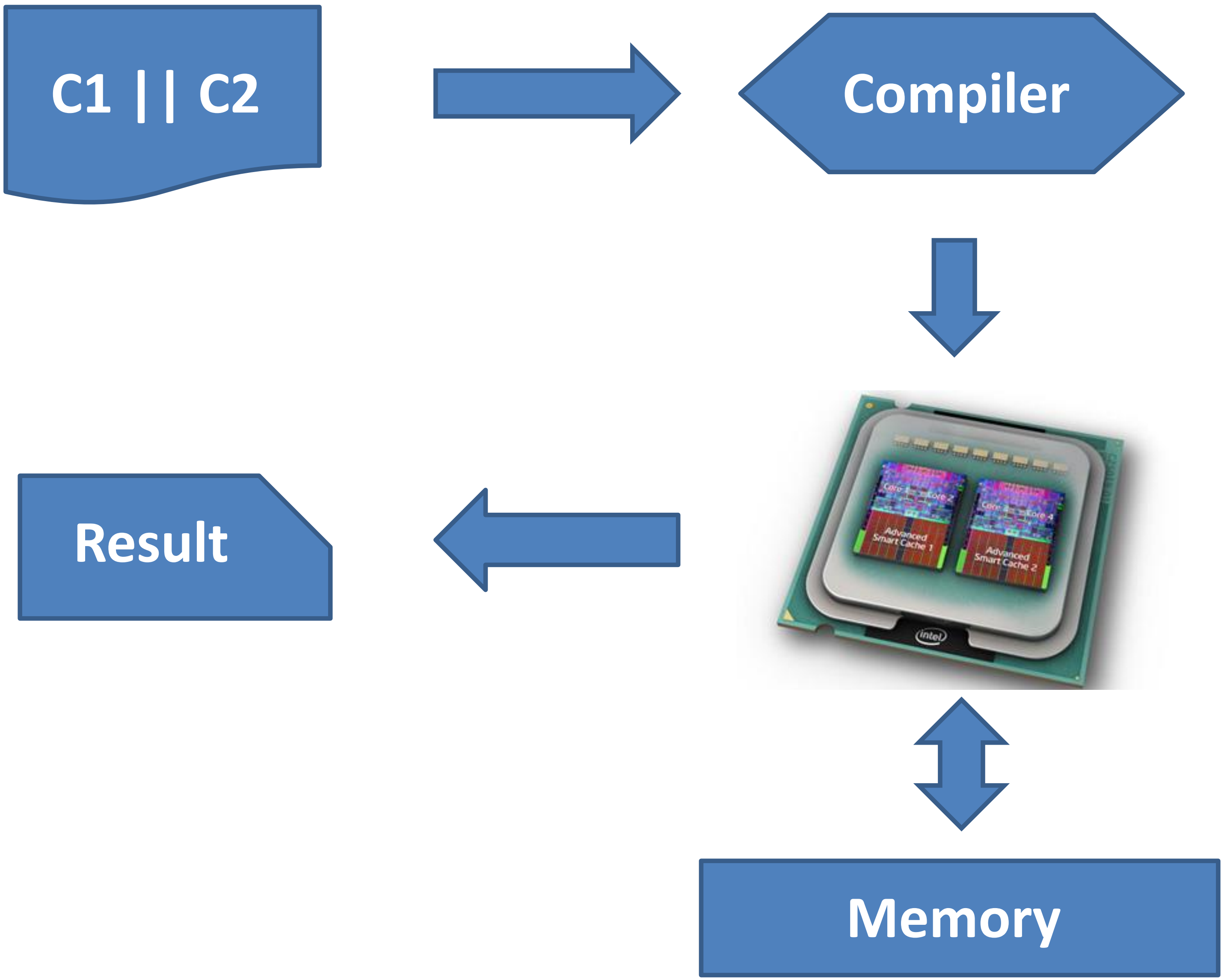
- Synchronize critical sections
  - Good default approach for building thread-safe classes
  - Only way to allow wait() and notify()
- Make objects immutable
  - Good if objects are small, simple abstract data type
  - Pass to methods without alias issues
- Use a thread-safe wrapper
  - Can give clients choice between thread-safe and non-safe
  - Works with existing class that is not thread-safe

Performance Issues

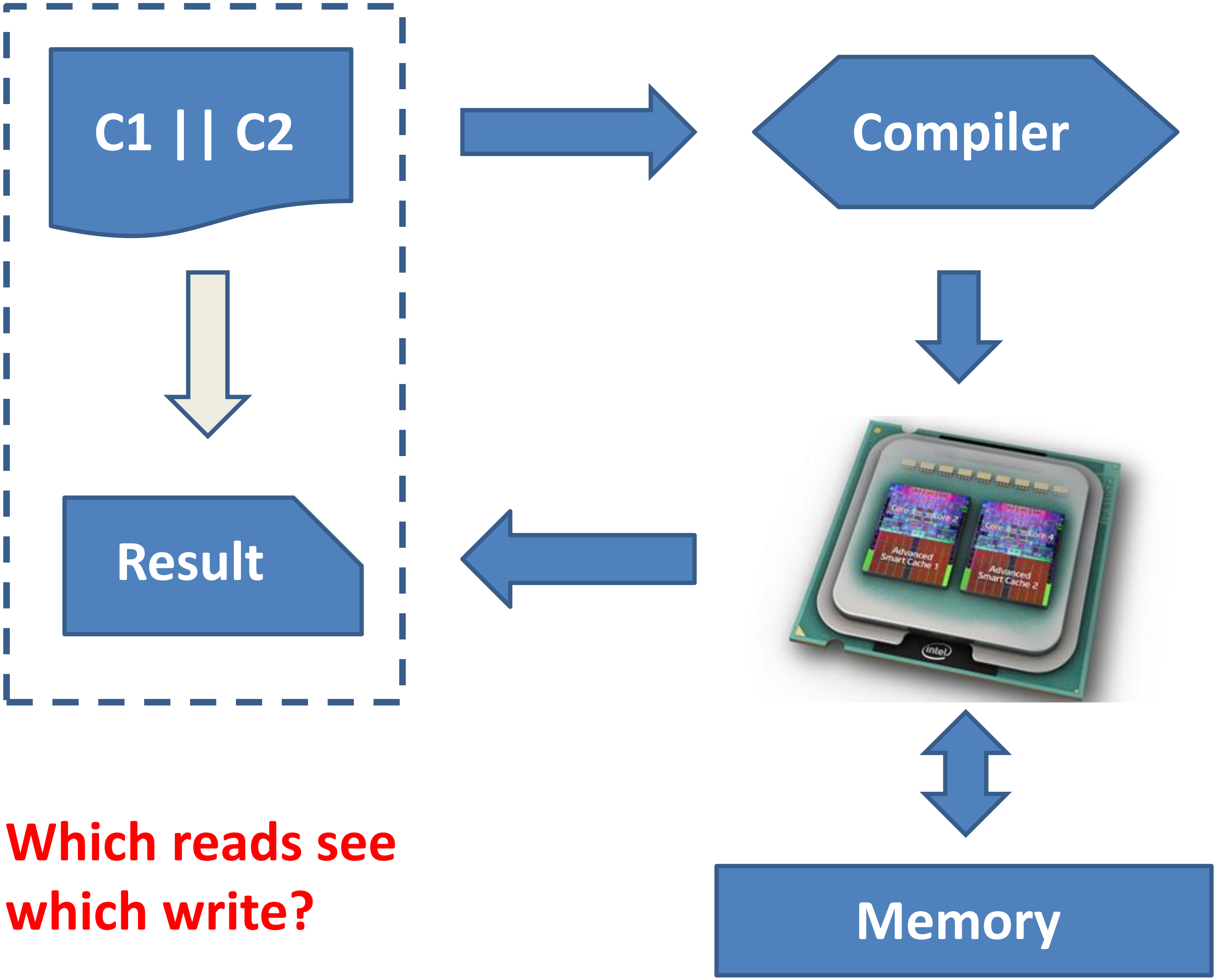
- Why not just synchronize everything?
  - Performance costs
    - Synchronized methods are 4 to 6 times slower than non-synchronized
  - Risks of deadlock from too much locking
- Performance in general
  - Unnecessary blocking and unblocking of threads can reduce concurrency
  - Immutable objects can be short-lived, increase garbage collector

Memory Models

Why Memory Models



Why Memory Models

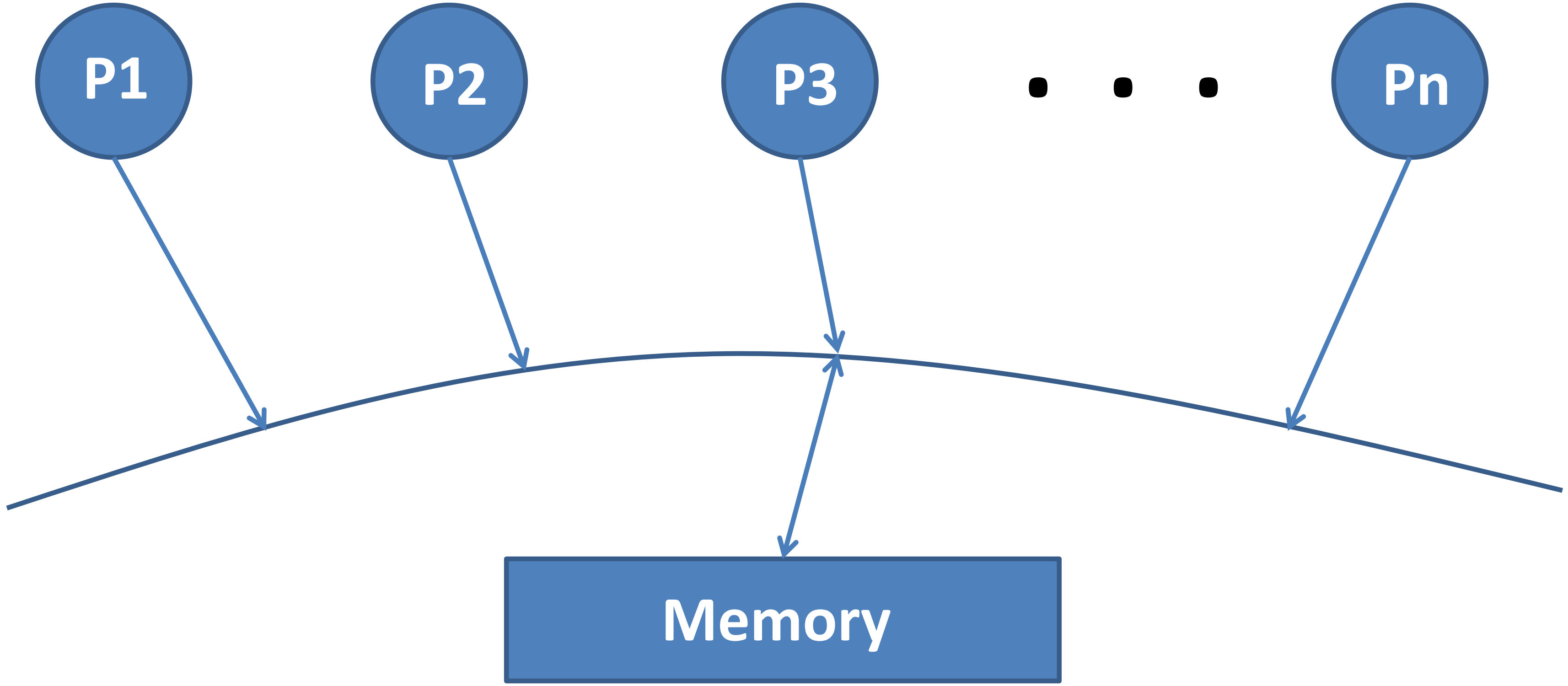


Which reads see which write?

Sequential Consistent (SC) Model

[Lamport 1979]

Interleaving semantics:



The need of weak memory models

SC model prohibits many optimization:

Initially: x = y = 0

x = 1;	‖	**y = 1;**
r1 = y;	‖	**r2 = x;**

⟹

r1 = y;	‖	**y = 1;**
x = 1	‖	**r2 = x;**

r1 = r2 = 0?

Impossible in SC model, but allowed in x86 or Java.

Weak memory model allow more behaviors.

Design Criteria

- Usability: DRF(data-race free) guarantee
  - DRF programs have the same behaviors as in SC model
- Not too strong
  - Allow common optimization techniques
  - In some sense hijacked by the mainstream compiler
- Preserve type-safety and security guarantee
  - Cannot be too weak

Very challenging to satisfy all the requirements!

Compiler Optimization Can Be Smart

Initially: x = 0, y = 1

```
r1 = x;          ||   r3 = y;
r2 = x;          ||   x = r3;
if (r1 == r2)
    y = 2;
```

```
y = 2;
r1 = x;
r2 = r1;
if (true)
```

r1 = r2 = r3 = 2?

Redundant read elim.

Must be allowed!

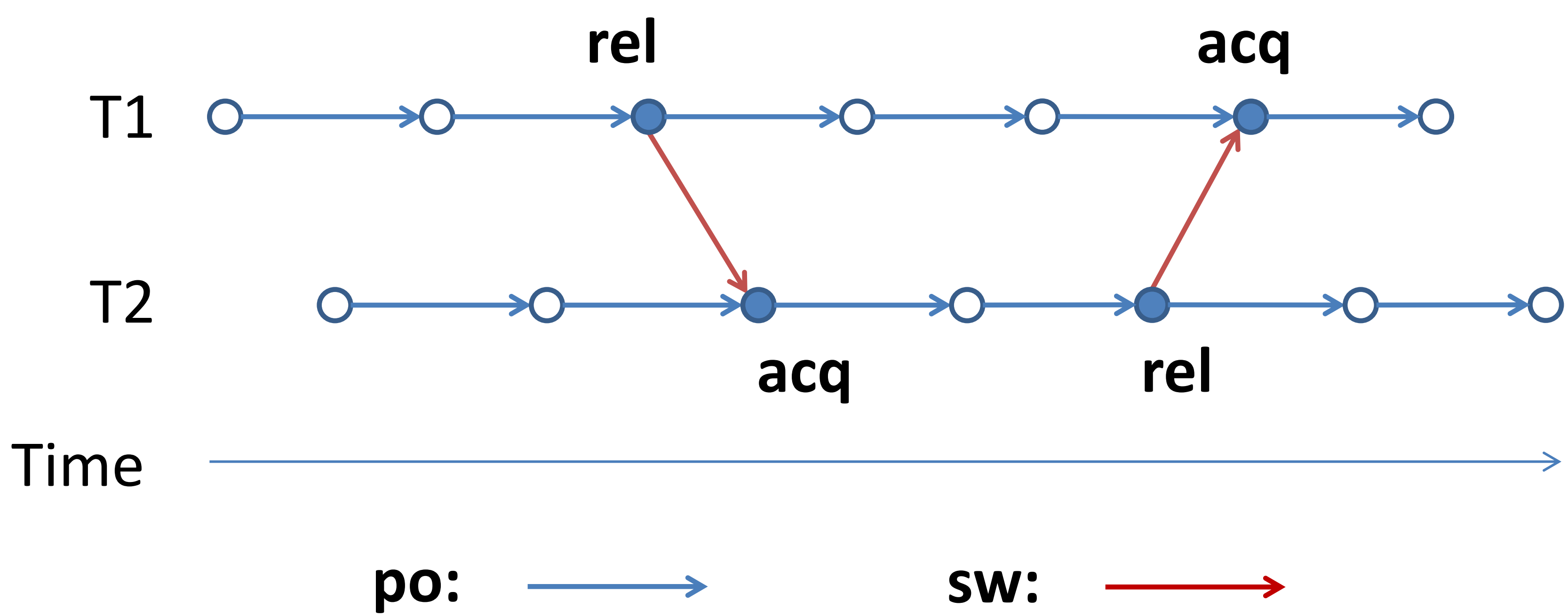
Efforts for Java Memory Model (JMM)

- First edition in Java Language Spec
  - Fatally flawed, not support key optimizations **[Pough 2000]**
- Current JMM [Manson et al. POPL 2005]
  - Based on 5-year discussion and many failed proposals
  - "very complex" **[Adve & Boehm CACM 2010]**
  - Surprising behaviors and bugs **[Aspinall & Sevcik TPHOLs 2007]**
- Next generation: JEP 188, Doug Lea, Dec. 2013, updated Jun. 2016

Happens-Before Order

[Lamport 1978]

Program execution: a set of events, and some orders between them.



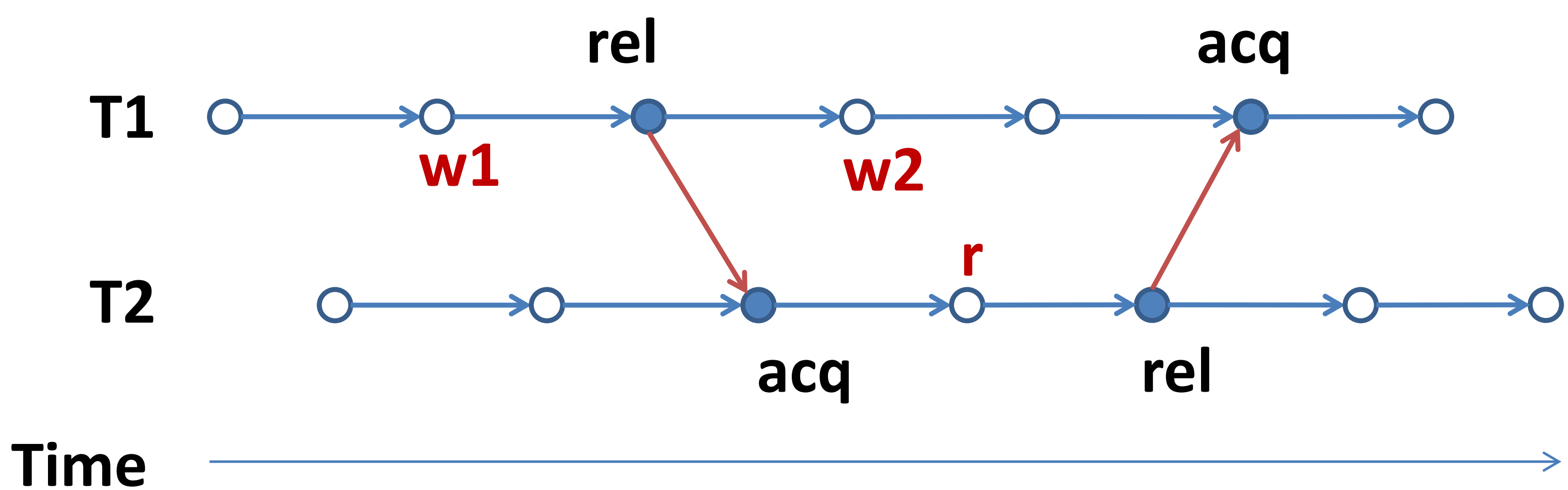
Happens-before order (hb): transitive closure of po∪sw

po: program order **sw**: synchronize-with

Happens-Before Order

$w1 \xrightarrow{hb} w2$ $\quad$ $w1 \xrightarrow{hb} r$

Not: $w2 \xrightarrow{hb} r$ $\quad$ $r \xrightarrow{hb} w2$

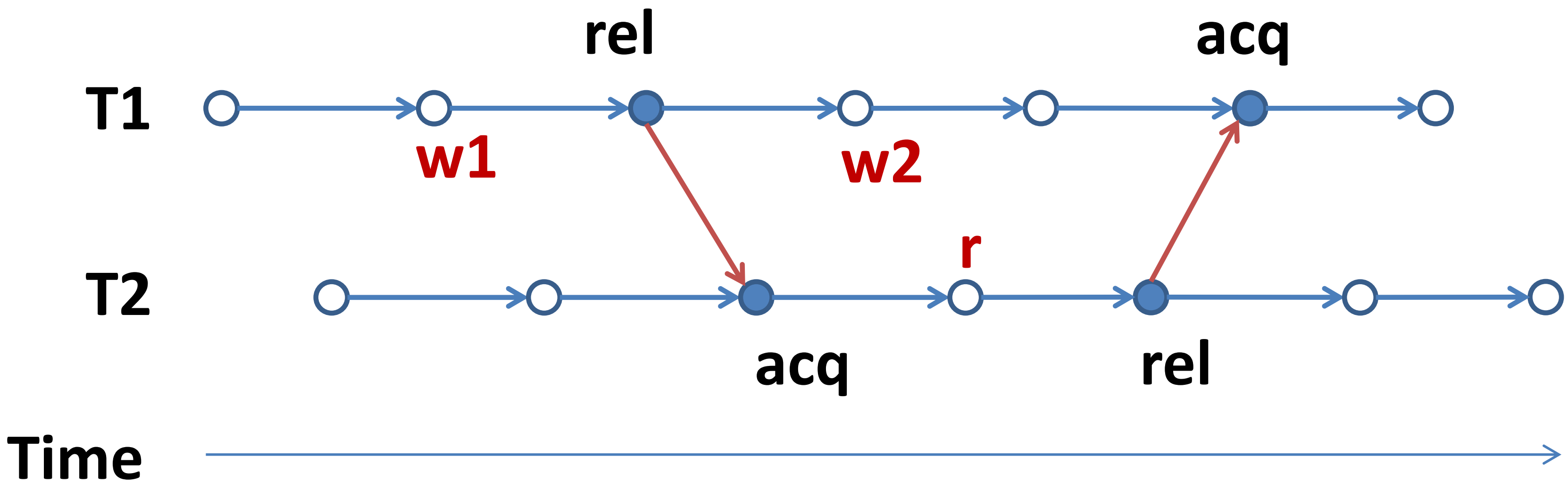


po: → **sw:** →

Happens-before order (hb): transitive closure of po∪sw

Happens-Before Memory Model (HMM)

Read can see
(1) the most recent write that happens-before it, or
(2) a write that has no happens-before relation.



r could see both **w1** (which happens-before it)
and **w2** (with which there is no happens-before relation)

HMM – Relaxed Ordering

Initially: x = y = 0

x = 1;	**y = 1;**
r1 = y;	**r2 = x;**

r1 = r2 = 0?

Allowed in HMM

HMM – Examples with Global Analysis

Initially: x = 0, y = 1

Speculation: r3 reads 2 ⇒ **Justified!**

Thread 1		Thread 2
r1 = x;		
r2 = x;		**r3 = y;**
if (r1 == r2)		**x = r3;**
y = 2;		

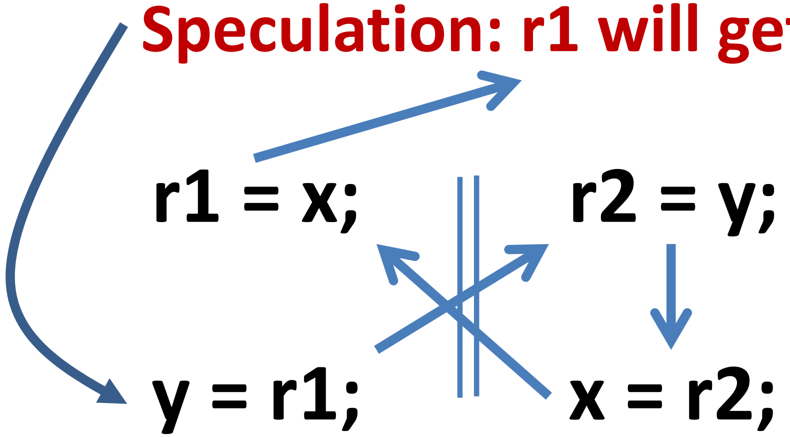
r1 = r2 = r3 = 2?

Allowed in HMM!

HMM – Out-of-Thin-Air Read

Initially: x = y = 0

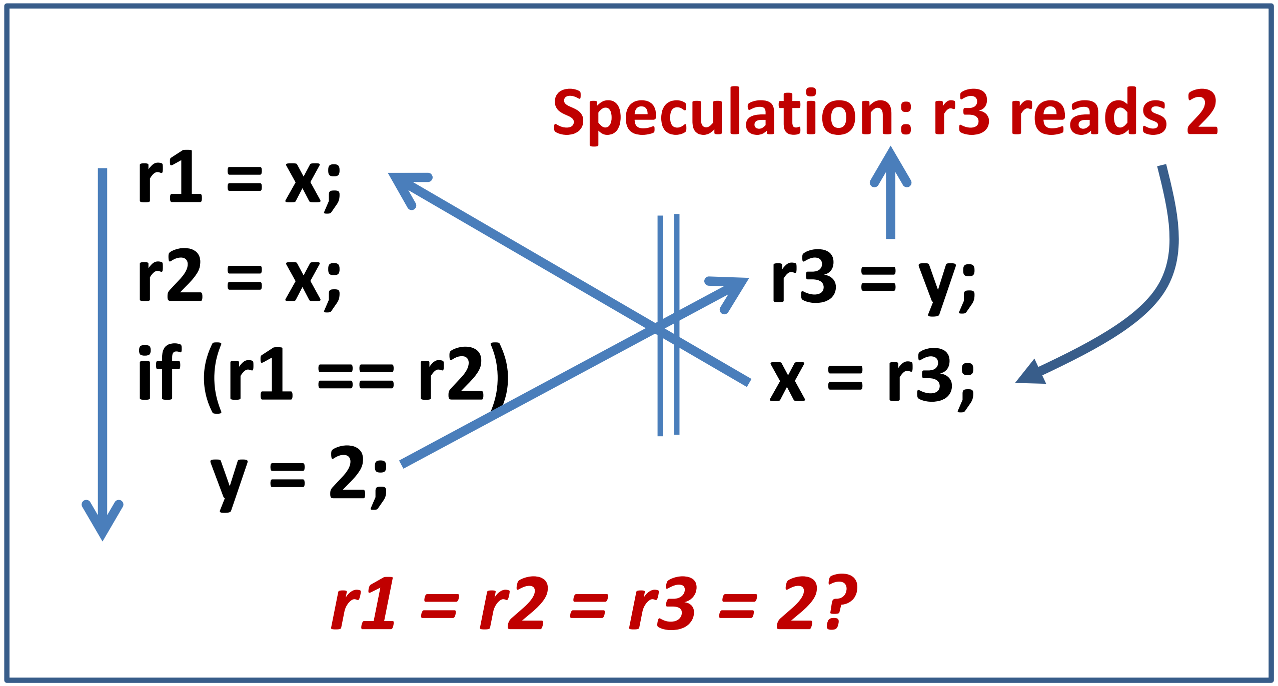
Speculation: r1 will get 42 ⇨ **Justified!**



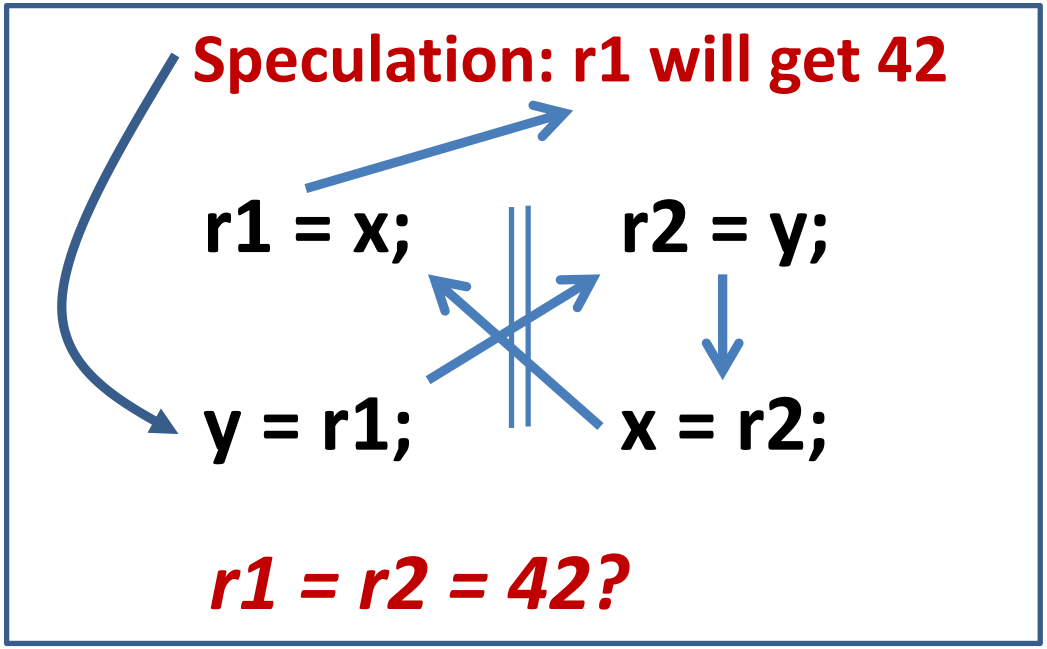
r1 = r2 = 42?

May break the security and type-safety of Java!

Allowed in HMM!



Good speculation.
Should allow!



Bad speculation.
Disallow!

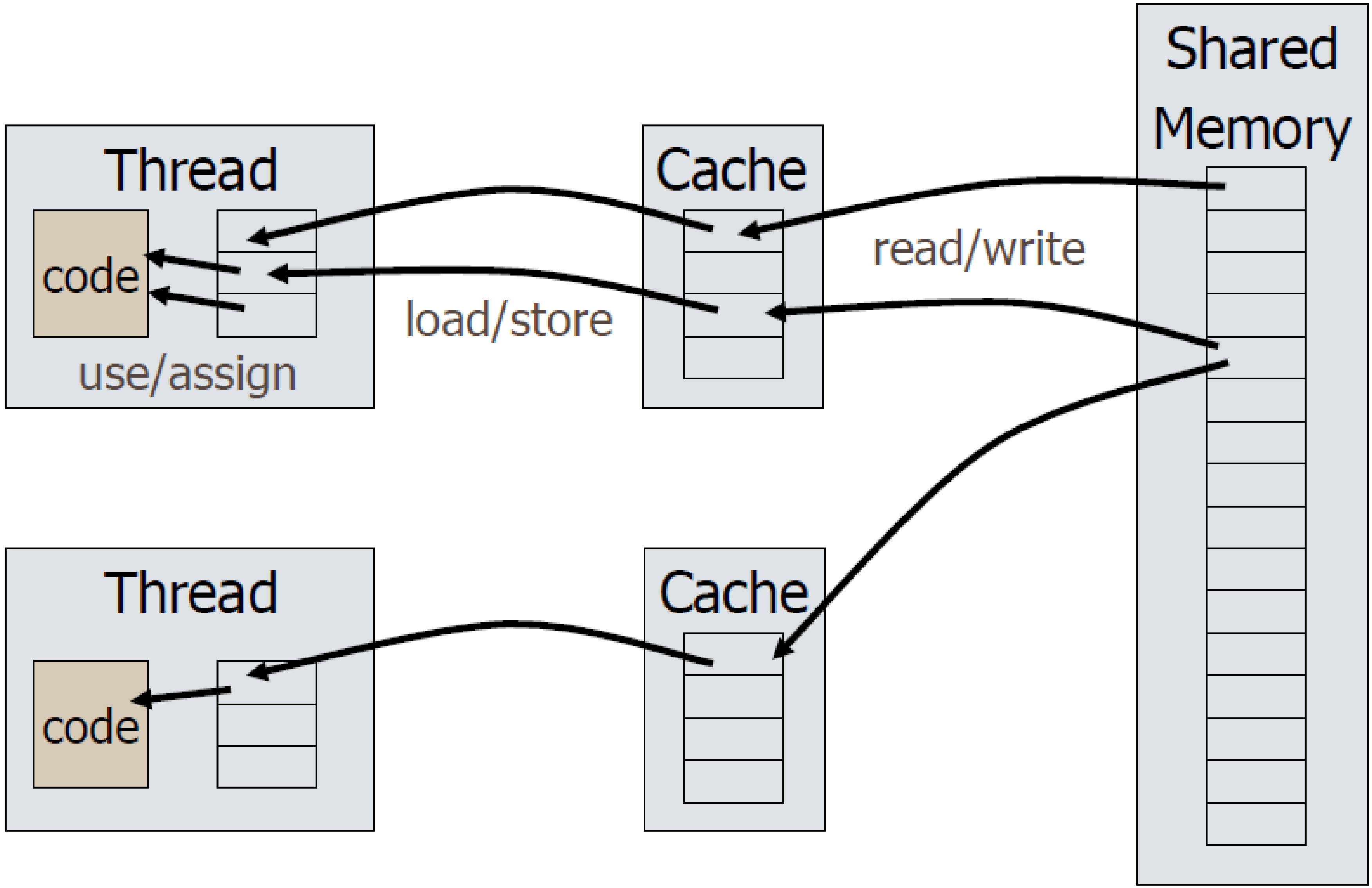
Java Memory Model

- Semantics of multithreaded access to shared memory
  - Competitive threads access shared data
  - Can lead to data corruption
  - Need semantics for incorrectly synchronized programs
- Determines
  - Which program transformations are allowed
    - Should not be too restrictive
  - Which program outputs may occur on correct implementation
    - Should not be too generous

http://www.cs.umd.edu/users/pugh/java/memoryModel/jsr-133-faq.html

Memory Hierarchy

Old memory model placed complex constraints on read, load, store, etc.



Race Conditions

- “Happens-before” order
  - Transitive closure of program order and synchronizes-with order
- Conflict
  - An access is a read or a write
  - Two accesses conflict if at least one is a write
- Race condition
  - Two accesses form a data race if they are from different threads, they conflict, and they are not ordered by happens-before

Race Conditions

Subtle issue: program order as written, or as compiled and optimized?

- "Happens-before" order
  - Transitive closure of program order and synchronizes-with order
- Conflict
  - An access is a read or a write
  - Two accesses conflict if at least one is a write
- Race condition
  - Two accesses form a data race if they are from different threads, they conflict, and they are not ordered by happens-before

Memory Model Question

- How should the compiler and run-time system be allowed to schedule instructions?
- Possible partial answer
  - If instruction A occurs in Thread 1 before release of lock, and B occurs in Thread 2 after acquire of same lock, then A must be scheduled before B
- Does this solve the problem?
  - Too restrictive: if we prevent reordering in Thread 1,2
  - Too permissive: if arbitrary reordering in threads
  - Compromise: allow local thread reordering that would be OK for sequential programs

JMM – Surprising Results

```
C1;                 ||
lock l;             ||    C3
   C2;              ||
unlock l;           ||
```

→

```
lock l;             ||
   C1;              ||    C3
   C2;              ||
unlock l;           ||
```

→

```
lock l;             ||
   C1;              ||
unlock l;           ||    C3
lock l;             ||
   C2;              ||
unlock l;           ||
```

Adding more synchronization may increase behaviors!

JMM – Surprising Results (2)

C1; || C2; || C3 ⟶ C1; C2; || C3;

Inlining threads may increase behaviors!

More:

Re-ordering independent operations may change behaviors.

Adding/removing redundant reads may change behaviors.

Instruction Order and Serializability

- Compilers can reorder instructions
  - If two instructions are independent, do in any order
  - Take advantage of registers, etc.
- Correctness for sequential programs
  - Observable behavior should be same as if program instructions were executed in the order written
- Sequential consistency for concurrent programs
  - If program P has no data races, then memory model should guarantee sequential consistency
  - Question: what about programs with races?
    - Much of complexity of memory model is for reasonable behavior for programs with races (need to test, debug, …)

Happens-Before Orderings

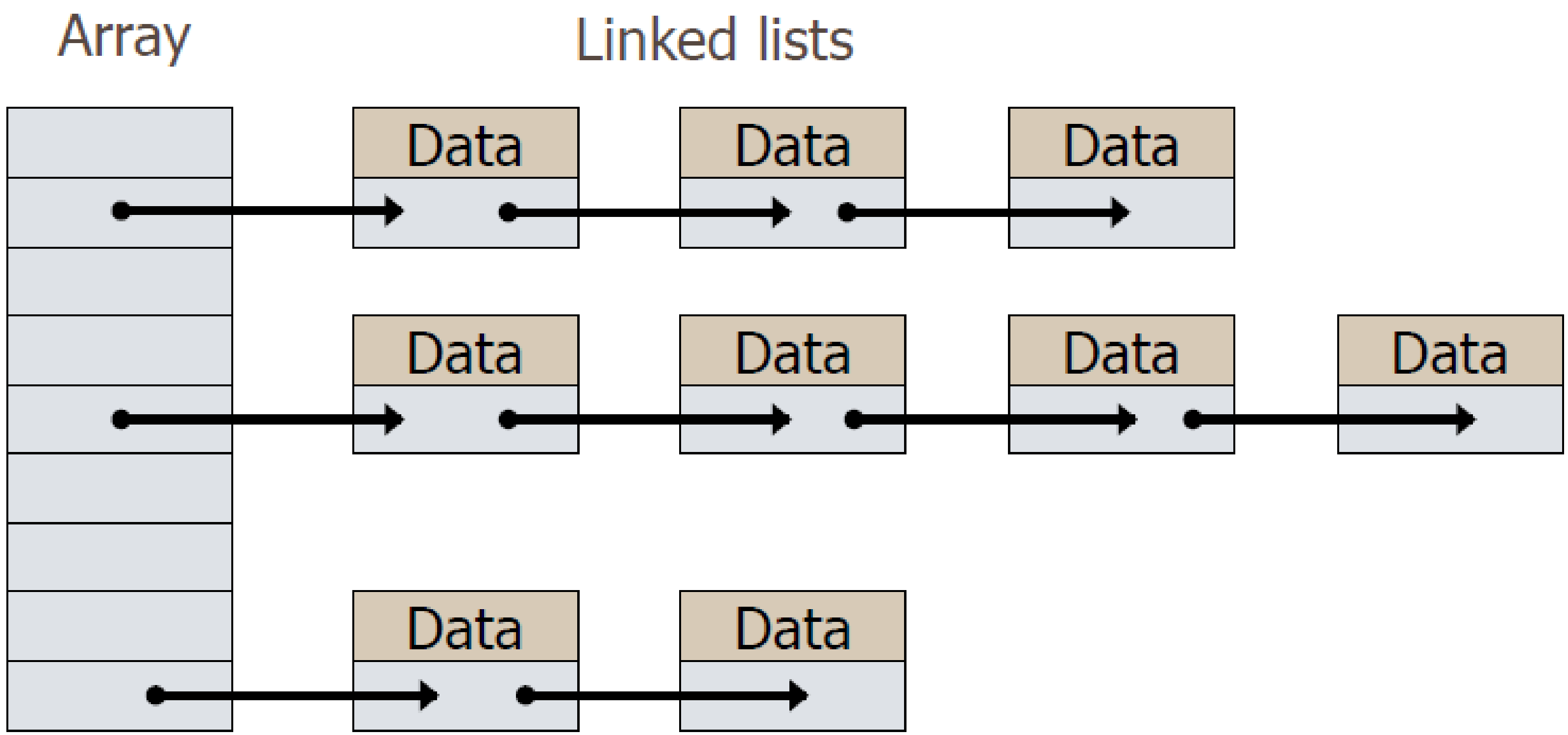
- Starting a thread happens-before the run method of the thread
- The termination of a thread happens-before a join with the terminated thread
- Volatile fields
- Many util.concurrent methods set up happens-before orderings
  - Placing an object into any concurrent collection happen-before the access or removal of that element from the collection

Example: Concurrent Hash Map

- Implements a hash table
  - Insert and retrieve data elements by key
  - Two items in same bucket placed in linked list
  - Allow read/write with minimal locking
- Tricky: https://www.ibm.com/developerworks/java/library/j-jtp08223/

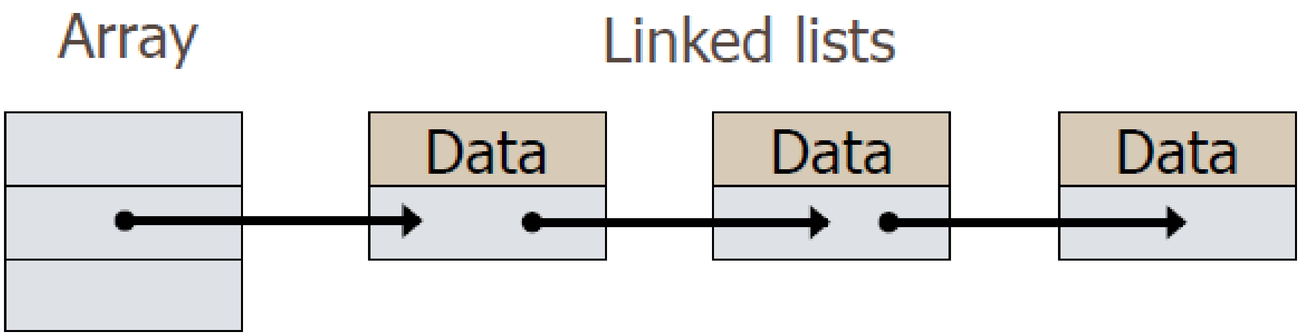
"ConcurrentHashMap is both a very useful class for many concurrent applications and a fine example of a class that understands and exploits the subtle details of the JMM to achieve higher performance. ConcurrentHashMap is an impressive feat of coding, one that requires a deep understanding of concurrency and the JMM. Use it, learn from it, enjoy it -- but unless you're an expert on Java concurrency, you probably shouldn't try this on your own. "

ConcurrentHashMap



- Concurrent operations
  - Read: no problem
  - Read/write: OK if different lists
  - Read/write to same list: clever tricks sometimes avoid locking

ConcurrentHashMap Tricks



- Immutability
  - List cells are immutable, except for data field

    =>read thread sees linked list, even if write in progress
- Add to list
  - Can cons to head of list, like Lisp lists
- Remove from list
  - Set data field to null, rebuild list to skip this cell
  - Unreachable cells eventually garbage collected

Problem with Language Specification

- Java Lang. Spec. allows access to partial objects

```
class Broken {
  private long x;
  Broken() {
    new Thread() {
        public void run() { x = -1; }
    }.start();
    x = 0;
} }
```

Thread created within constructor can access the object not fully constructed

Nested Monitor Lockout Problem

- Background: wait and locking
  - wait and notify used within synchronized code
    - Purpose: make sure that no other thread has called method of same object
  - wait within synchronized code causes the thread to give up its lock and sleep until notified
    - Allow another thread to obtain lock and continue processing
- Problem
  - Calling a blocking method within a synchronized method can lead to deadlock

Nested Monitor Lockout Example

```
class Stack {
    LinkedList list = new LinkedList();
    public synchronized void push(Object x) {
            synchronized(list) {
                    list.addLast( x ); notify();
    } }
    public synchronized Object pop() {
            synchronized(list) {
                    if ( list.size() <= 0 ) wait();
                    return list.removeLast();
    } }
}
```

Releases lock on Stack object but not lock on list; a push from another thread will deadlock

Preventing Nested Monitor Deadlock

- Two programming suggestions
  - No blocking calls in synchronized methods, or
  - Provide some non-synchronized method of the blocking object
- No simple solution that works for all programming situations

Reading

http://www.cs.umd.edu/~pugh/java/memoryModel/

http://openjdk.java.net/jeps/188

Foundations of the C++ concurrency memory model, Boehm & Adve, PLDI 2008