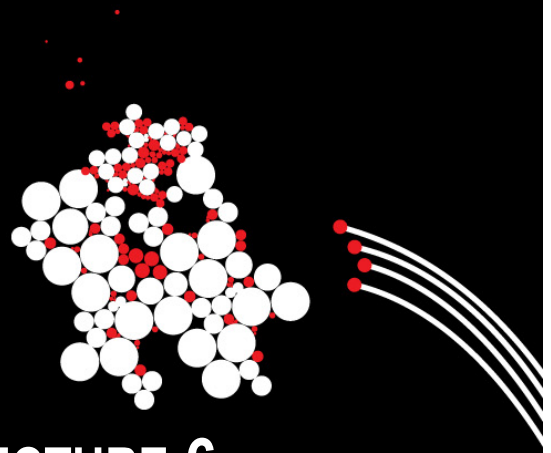
UNIVERSITY OF TWENTE.

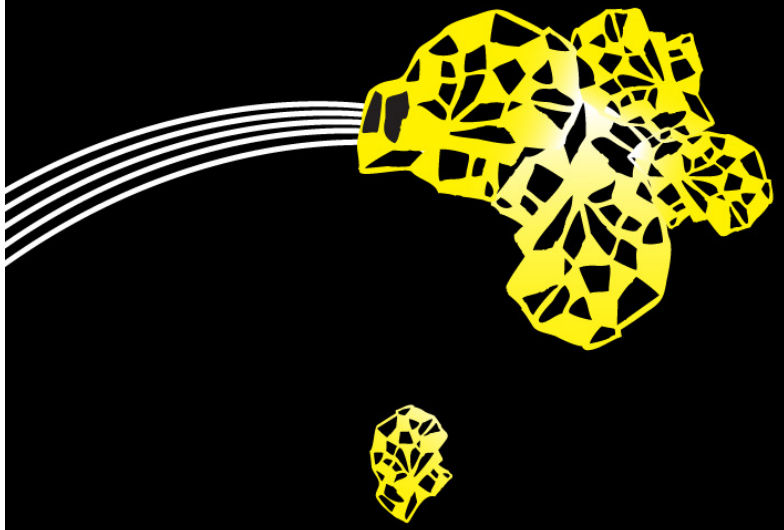


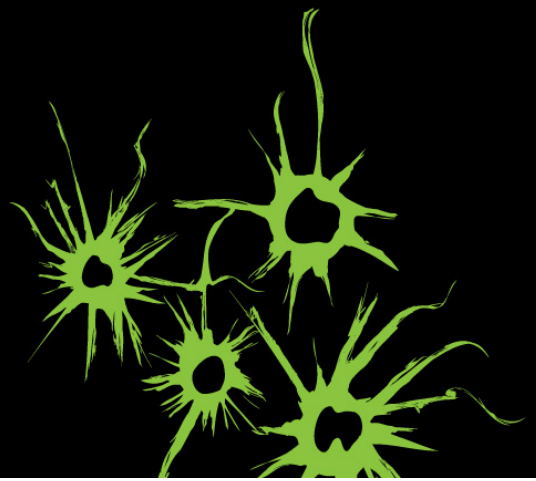
CONCURRENT PROGRAMMING – LECTURE 6

FINE-GRAINED CONCURRENCY, MEMORY MODELS

MODULE 8: PROGRAMMING PARADIGMS

7 JUNE 2019

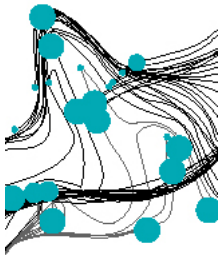




CONCURRENT PROGRAMMING

CONTENTS

2	Tue 30/4	Basics of concurrency, thread safety, testing concurrent systems	Ch. 1-3, 12 (+ SS material)
3	Tue 7/5	Synchronisation	Ch. 4, 5, 13, 14.1-14.4
4	Wed 15/5	Liveness, performance, and fairness	Ch. 10,11
5	Thu 23/5	Homogeneous threading (OpenMP, OpenCL)	Papers
6	Mon 3/6	Safe concurrency (Software Transactional Memory, Rust)	Papers
7	Fri 7/6	Fine-grained concurrency, memory models	Ch. 14.5-6, 15, 16

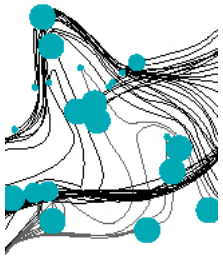


FINE-GRAINED CONCURRENCY

- Progress properties
- Optimistic programming
- Lock-free programming
- Wait-free programming
- Memory models

Literature:
JCIP: Ch. 14.5-14.6, Ch. 15, and Ch. 16

Recommended:
The Art of Multiprocessor Programming
M. Herlihy & N. Shavit
Revised first edition
In particular sections 9.8, 10.5, 11.2



PROGRESS PROPERTIES

Picking a progress property for a given application depends on its needs

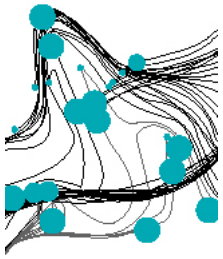
- Blocking
  - Deadlock-free: *some* thread trying to get the lock eventually succeeds
  - Starvation-free: *every* thread trying to get the lock eventually succeeds
- Non-blocking
  - Lock-free: infinitely often *some* method call finishes in a finite number of steps
  - Wait-free: *every* method call finishes in a finite number of steps

Lock- and wait-freeness disallow blocking methods like locks.

Lock-free: guaranteed system-wide progress
Wait-free: guaranteed per-thread progress

BLOCKING VS NON-BLOCKING

Locking is a defensive technique: lock everybody out!	Disadvantage of locking: ▪ suspending/resuming has large overhead when lock contended ▪ when a thread holds lock, all others blocked
Lock-free programming improves: ▪ scalability ▪ liveness	Disadvantage of lock-free: ▪ error-prone



OPTIMISTIC PROGRAMMING

A NONBLOCKING COUNTER

Note that `AtomicInteger` itself already provides an atomic `increment` method.

```
@ThreadSafe
public class CASCounter {
    private AtomicInteger value;

    public int getValue() {
        return value.get();
    }

    public int increment() {
        int v;
        do {
            v = value.get();
        } while (! value.compareAndSet(v, v+1));
        return v+1;
    }
}
```

Optimistic programming: just perform the update and check that nobody interfered

OPTIMISTIC PROGRAMMING

TYPICAL PATTERN

Note resemblance with software transactional memory

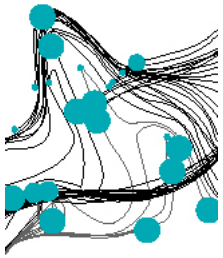
```
lock.lock()
var = operation_on(var.get());
lock.unlock();
```

protect update by lock

```
AtomicX var = ...   // shared by threads

do {
   oldv = var.get();
   newv = operation_on(old_value);
} while (! var.compareAndSet(oldv, newv))
```

optimistic: we hope that no thread interferes

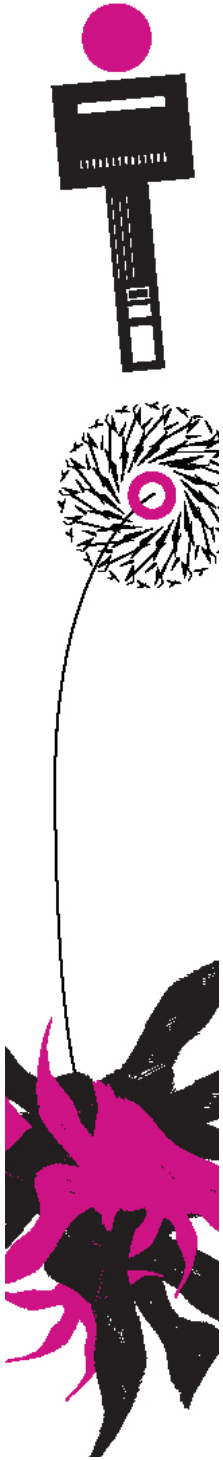
If some other thread interfered (i.e., `var.get` has changed), try again



LOCK-FREE PROGRAMMING

APPROACH

- Uses optimistic programming pattern
- Reading and updating atomics
- Uses `compareAndSet` = atomic commit
- Examples:
  - Lock-free Stack
  - Unbounded Queue (from locks to lock-free)



LOCK-FREE STACK

TREIBER'S STACK (1986)

```
public class LockFreeStack<T> {
  AtomicReference<Node<T>> head =
      new AtomicReference<Node<T>>();

  public void push(T item) { ... }
  public T    pop() { ... }

  static class Node<T> {
    final T item;
    Node<T> next;
    public Node(T item) {
      this.item = item;
    }
}}
```

points to top of stack

LOCK-FREE STACK

POP

```
public T pop() {
  Node<T> oldHead;
  Node<T> newHead;

  do {
    oldHead = head.get();
    if (oldHead == null)
      return null;
    newHead = oldHead.next;
  } while (!head.compareAndSet(oldHead, newHead));

  return oldHead.item;
}
```

LOCK-FREE STACK

PUSH

```
public void push(T item) {
  Node<T> newHead = new Node<T>(item);
  Node<T> oldHead;

  do {
    oldHead = head.get();
    newHead.next = oldHead;
  } while (!head.compareAndSet(oldHead, newHead));
}
```

CAS approach: best-known low-load method, but scales poorly due to contention and inherent sequential bottleneck.

better:
- elimination
- backoff
- stack

LINEARISABILITY

CORRECTNESS PRINCIPLE

Linearisation point: the moment the new state becomes visible to other threads

Linearisability:

Each method call should appear to take effect instantaneously at some moment between its invocation and return

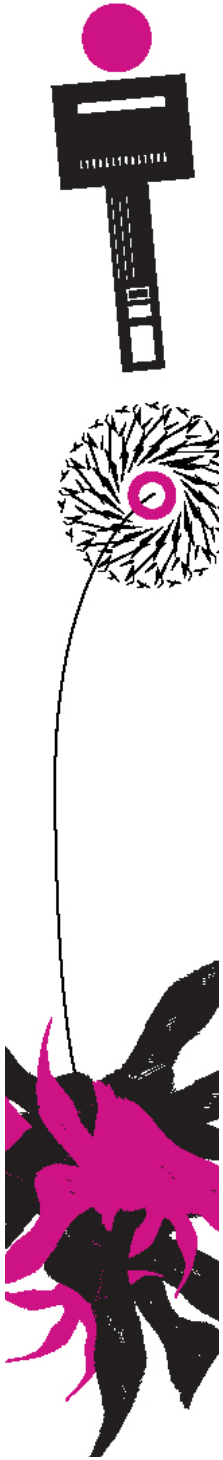
- real-time behavior of method calls must be preserved
- concurrent object is linearisable, if all its possible executions are linearisable

Linearisable: behaviour can be described as sequences of linearisation points

```
public T pop() {
   ...
   do {
      ...
   } while (!head.compareAndSet(.
   ...
}
```

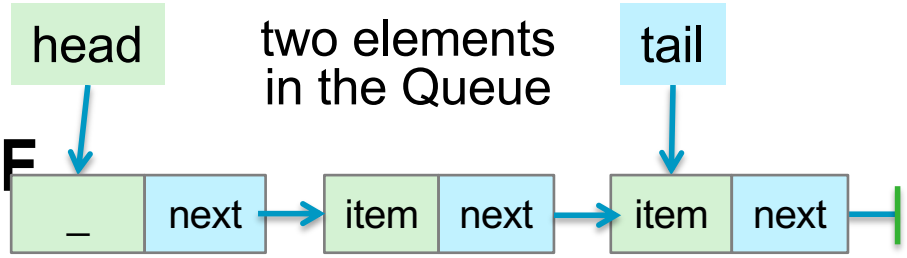
linearisation points

```
id push(T item) {
   ...
   do {
      ...
   } while (!head.compareAndSet(...));
}
```



UNBOUNDED LOCKED QUEUE

LOCKED




```
class UnboundedQueue<T> {
  final Lock enqLock = new ReentrantLock();
  final Lock deqLock = new ReentrantLock();
  final Condition notEmpty
    = deqLock.newCondition();
  Node<T> head, tail;

  public UnboundedQueue() {
    head = tail = new Node<T>(null);
  }

  public void enq(T x) {...}
  public T deq() {...}
}
```

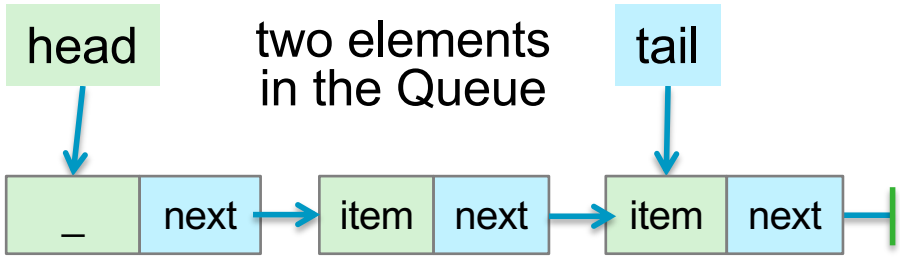
independent locks:

- enqueuer works on tail
- dequeuer works on head

sentinel (dummy) node

UNBOUNDED LOCKED QUEUE

ENQUEUE AND DEQUEUE




```
public void enq(T x) {
  enqLock.lock();
  try {
    Node<T> e
      = new Node<T>(x);
    tail.next = e;
    tail = e;
    notEmpty.signal();
  } finally {
    enqLock.unlock();
  }
}
```

```
public T deq() {
  deqLock.lock();
  try {
    while (head.next == null)
      notEmpty.await();
    T result = head.next.item;
    head = head.next;
    return result;
  } finally {
    deqLock.unlock();
  }
}
```

linearisation points

UNBOUNDED LOCK-FREE QUEUE

LOCK-FREE

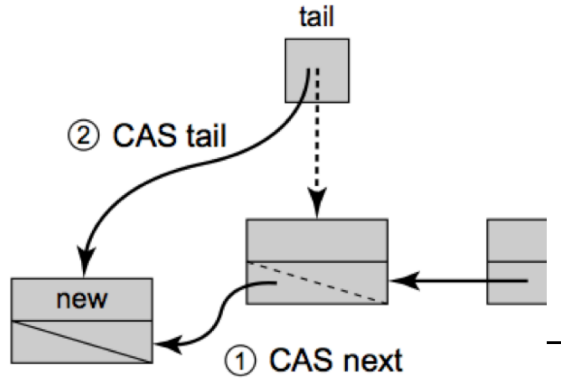
```
class LockFreeQueue<T> {
  AtomicReference<Node<T>> head, tail;

  public LockFreeQueue() {
    Node<T> n = new Node<T>(null);
    head = tail =
      new AtomicReference<Node<T>>(n);
  }

  public void enq(T x)
  public T deq()  { ...
}
```

sentinel node

```
class Node<T> {
  public final T value;
  public AtomicReference<Node<T>> next;
  public Node(T value) {
    this.value = value;
    next = new AtomicReference<Node<T>>();
  }
}
```

LOCK-FREE QUEUE

ENQUEUE




```
public void enq(T x) {
  Node<T> node = new Node<T>(x);
  while (true) {
    Node<T> last = tail.get();
    Node<T> next = last.next.get();
    if (last == tail.get()) {
      if (next == null) {
        if (last.next.compareAndSet(next, node)) {
          tail.compareAndSet(last, node);
          return;
        }
      }
      else
        tail.compareAndSet(last, next);
} } }
```

locate last node

consistency check

check if still last

append new node

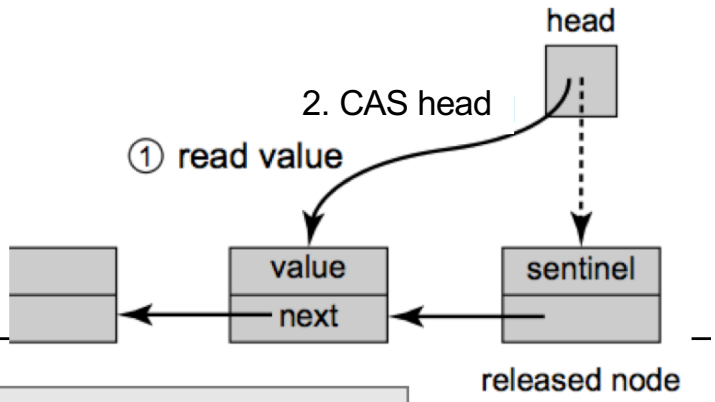
update tail

help advance lagging tail of other thread

linearization point

locate last/append/tail update is not atomic: other threads must 'help' completing half-finished `enq` calls

UNBOUNDED DEQUEUE




```
public T deq() {
  while (true) {
    Node<T> first = head.get();
    Node<T> last  = tail.get();
    Node<T> next  = first.next.get();
    if (first == head.get()) {
      if (first == last) {
        if (next == null) return null;
        tail.compareAndSet(last, next);
      } else {
          T value = next.value;
          if (head.compareAndSet(first, next))
            return value;
} } } }
```

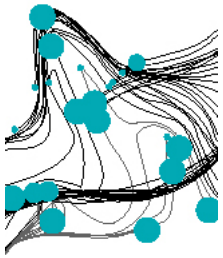
consistency check

check if queue not empty

advance lagging tail

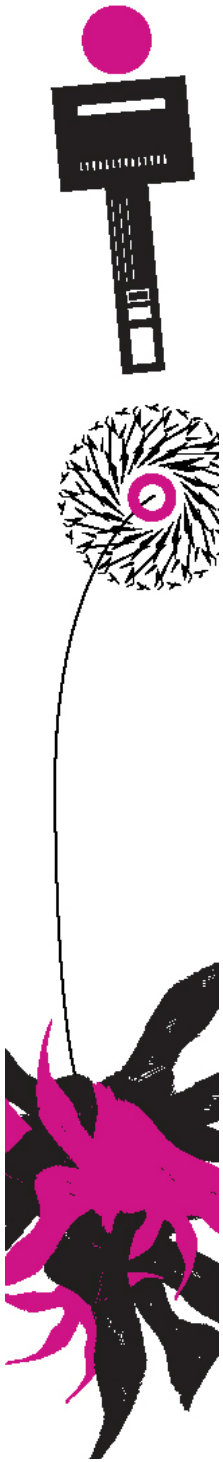
try again if other thread got value

As with `enq`: `deq` must help with half-finished `enq` call



WAIT-FREE ALGORITHMS

- All threads make progress
- Atomic updates always succeed
- Examples:
  - Single enqueuer/single dequeuer queue
  - Linked list with wait-free contains



WAIT-FREE FIFO QUEUE

SINGLE ENQUEUER & SINGLE DEQUEUER

Single Enqueuer and
Single Dequeuer!

```
class WaitFreeQueue<T> {
  volatile int head = 0, tail = 0;
  T[] items;

  public WaitFreeQueue(int capacity) {
    items = (T[]) new Object[capacity];
  }

  public void enq(T x) { ... }
  public T deq()       { ... }
}
```

WAIT-FREE FIFO QUEUE

ENQUEUE & DEQUEUE

Single Enqueuer and Single Dequeuer!

```
public void enq(T x) {
  while (tail - head ==
         items.length) {
    // spin
  }

  items[tail % items.length]
    = x;
  tail++;
}
```

```
public T deq() {
  while (tail == head) {
    // spin
  }

  T x =
    items[head % items.length];
  head++;
  return x;
}
```

linearization points

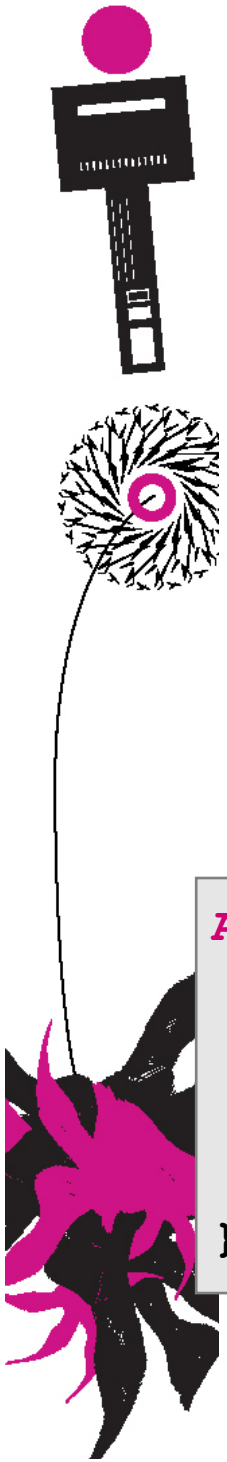
Why is spinning not fatal here?

Why does it work without locks?

- Addition and removal are lock-free
- Membership test is wait-free

HIGHLY OPTIMISED LINKED LIST

USING ATOMICMARKABLEREFERENCE



- LinkedList implementation without any locks
- Nodes are marked as removed before actually being removed
- Markable reference: cannot be updated when marked
- Logical removal: marking the next pointer
- Physical removal: during list traversal when encountering marked pointers

```
AtomicMarkableReference<T> interface {
  public boolean compareAndSet(
                    T expectedReference, T newReference,
                    boolean expectedMark, boolean newMark);
  public T get(boolean[] marked);
}
```

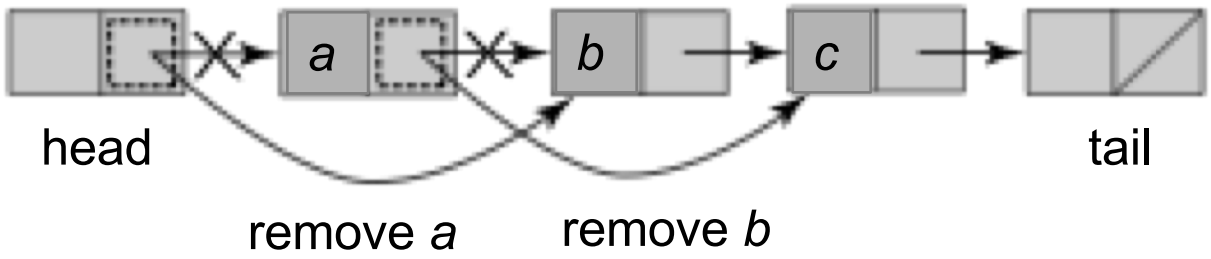
returns reference and stores the mark at position 0 of array

```
boolean[1] holder;
ref = v.get(holder);
```

HIGHLY OPTIMISED LINKED LIST

ADVANTAGE OF MARKING

- Avoids having to re-traverse the list



- Suppose removal of b in parallel with removal of a
- Net effect might be only removal of a
- Instead: mark the pointer as removed, and physical removal during next traversal

wait-free

HIGHLY OPTIMISED LINKED LIST

CONTAINS

```
public boolean contains(T item) {
  boolean[] marked = {false};
  int key = item.hashCode();
  Node curr = head;

  while (curr.key < key) {
    curr = curr.next;
    Node succ = curr.next.get(marked);
  }
  return (curr.key == key && !marked[0]);
}
```

list is sorted on hash code, no duplicates

marked nodes are not removed

auxiliary, lock-free method

HIGHLY OPTIMISED LINKED LIST

FIND

```
public class Window {
  public Node pred, curr;
  Window(Node p, Nody c) {
    pred = p; curr = c;
  }
```

pred.key < key <= curr.key

```
public Window find(Node head, int key) {
  Node pred = null; curr = null; succ = null;
  boolean[] marked = {false};
  retry: while (true) {
    pred = head; curr = pred.next.getReference();
    while (true) {
      succ = curr.next.get(marked);
      while(marked[0]) {
        if (!pred.next.compareAndSet(curr, succ, false, false)
          continue retry;
        curr = succ; succ = curr.next.get(marked);
      }
      if (curr.key >= key)
        return new Window(pred, curr);
      pred = curr;
      curr = succ;
} } }
```

Marked references are being removed

lock-free

HIGHLY OPTIMISED LINKED LIST

ADD

```
public boolean add(T item) {
  int key = item.hashCode();
  while (true) {
    Window window = find(head, key);
    Node pred = window.pred;
    Node curr = window.curr;
    if (curr.key == key) {
      return false;
    }
    else {
      Node node = new Node(item);
      node.next = new AtomicMarkableReference(curr, false);
      if (pred.next.compareAndSet(curr, node, false, false)) {
        return true;
} } } }
```

Find insertion point for `item`

lock-free

HIGHLY OPTIMISED LINKED LIST

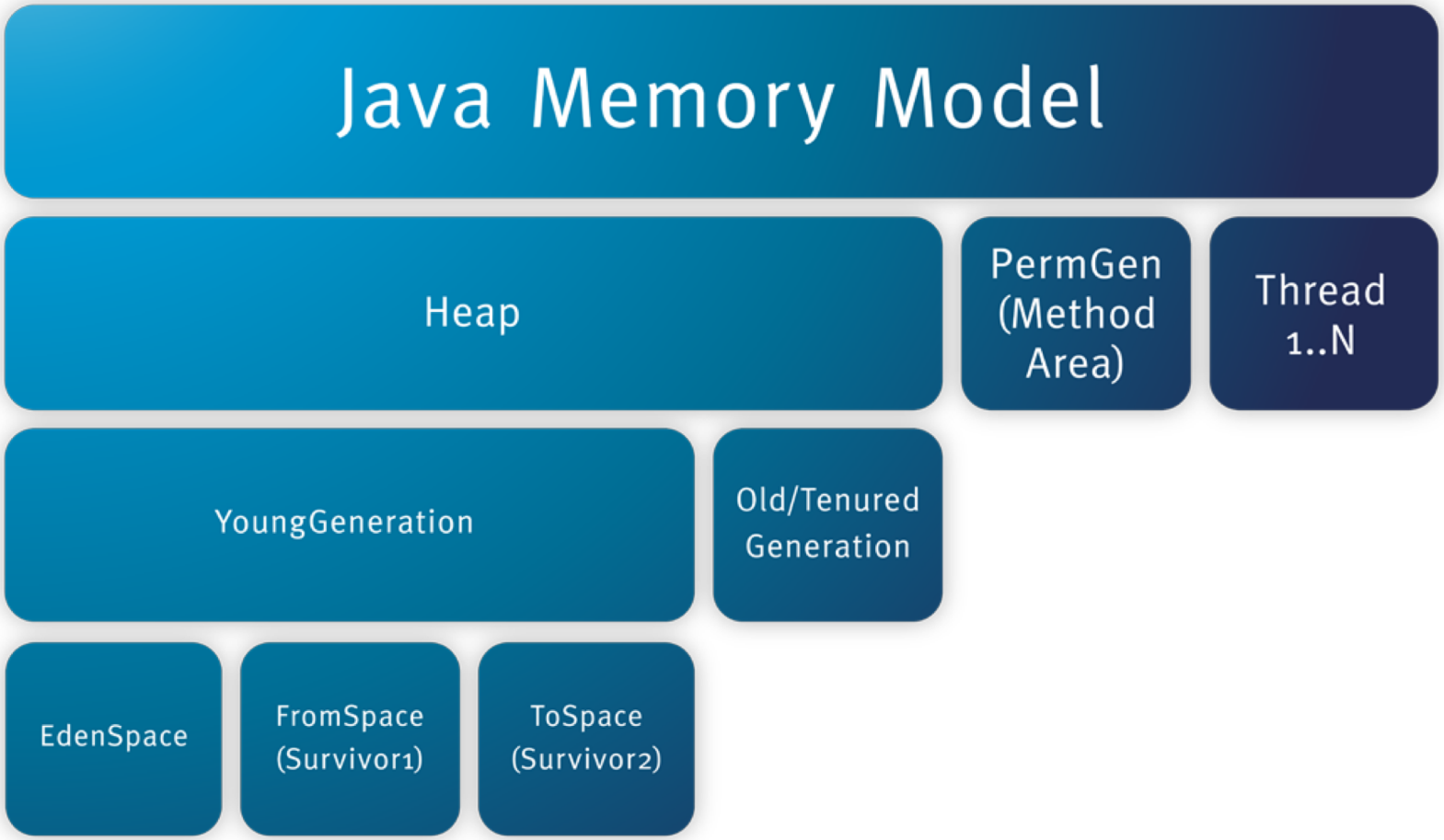
REMOVE

```
public boolean remove(T item) {
  int key = item.hashCode();
  while (true) {
    Window window = find(head, key);
    Node pred = window.pred;
    curr = window.curr;
    if (curr.key != key) {
      return false;
    } else {
      Node succ = curr.next.getReference();
      if (! curr.next.attemptMark(succ, true)) { continue; }
      pred.next.compareAndSet(curr, succ, false, false);
      return true;
} } }
```

Mark `curr.next` as removed

Single attempt to also physically remove the node.

MEMORY MODELS



WHAT IS A MEMORY MODEL?

A memory model describes the interactions of threads through memory and their shared use of the data

- Essentially: when (under what conditions) does a write become visible to other threads

Compilers do many optimisations:

- invisible under sequential execution
- but important for concurrent execution

BEHAVIOUR OF CONCURRENT PROGRAMS

convention:
- `x`, `y`, `z`: global variables
- `r1`, `r2`: local variables (registers)

Thread 1

```
x = 42;
if (x == 41) {
    x = 3;
}
r1 = y;
```

Thread 2

```
y = 64;
if (z == 0) {
    z = y + 5;
}
r2 = x;
```

If initially `x`, `y` and `z` are 0, what are the possible final values of `r1` and `r2`?

```
(0 ,42)
(64,0)
(64,42)
(0 ,0)
```

… `r1` and `r2` can also both be 0 due to compiler reorderings!

POSSIBLE COMPILER REORDERINGS

- register allocation

```
update x (repeatedly)
```

```
r0 = x
update r0 (repeatedly)
x = r0
```

compiler 'knows' that `x` resides in `r0`

- speculative execution
  (e.g., branch prediction)

```
if (condition)
  update x
```

```
r0 = x
update x
if (not condition)
  x = r0 // undo
```

- optimization of loop condition

```
x = 0;
while (x == 255);
```

```
x = 0;
while (false);
```

complete `while`-loop will be removed

… it seems that the compiler / run-time system 'invents' updates

EFFECTS OF COMPILER OPTIMISATIONS

Initially `x == y == 0`

Thread 1

```
r1 = x;
if (r1 >= 0) {
    y = 1;
}
```

Thread 2

```
r2 = y;
if (r2 >= 0) {
    x = 1;
}
```

… can this result in `r1 == r2 == 1`?

Yes!

- All stores to `x` and `y` are of constants `0` or `1`
- Therefore `r1` and `r2` are non-negative
- Therefore `if`-guards are true
- Therefore writes can be moved early

SEQUENTIAL CONSISTENCY

Sequential consistency is a property that requires that the result of any execution is the same as

- if the operations of all the processors were executed in some sequential order
- the operations of each individual processor appear in this sequence in the order specified by its program

Naïve translation:

- interleaving of

  atomic instructions

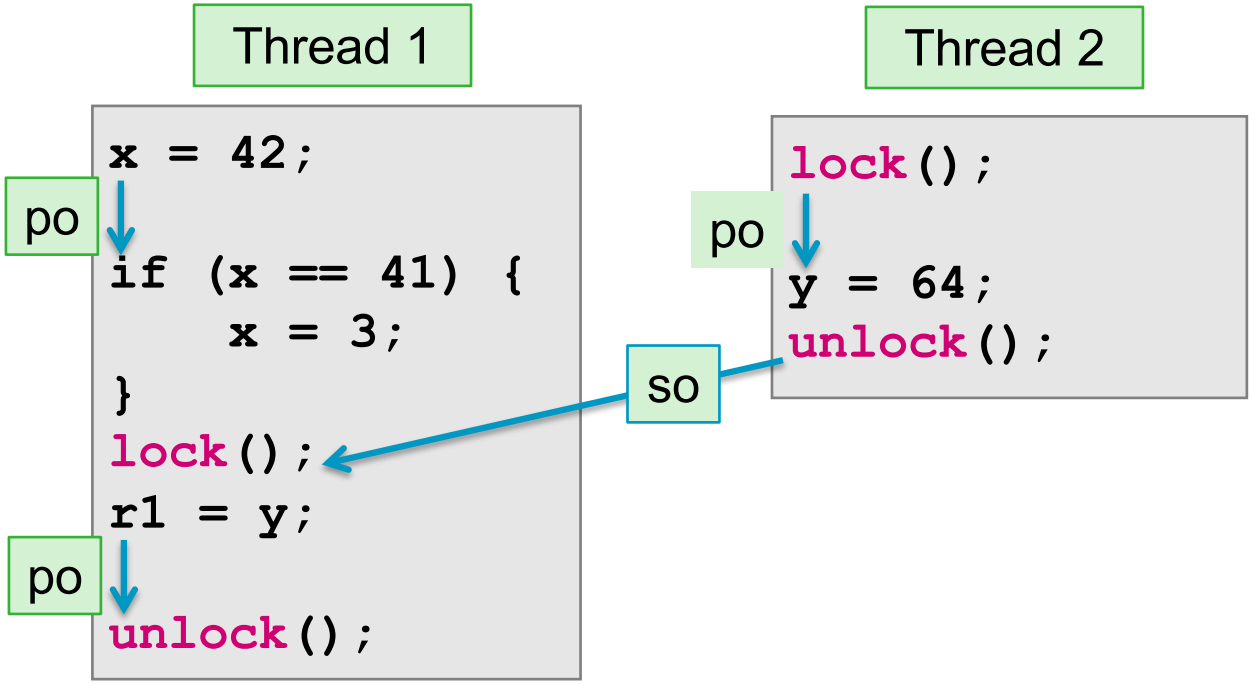
… sequential consistency is too expensive to implement!

MEMORY MODEL

INGREDIENTS

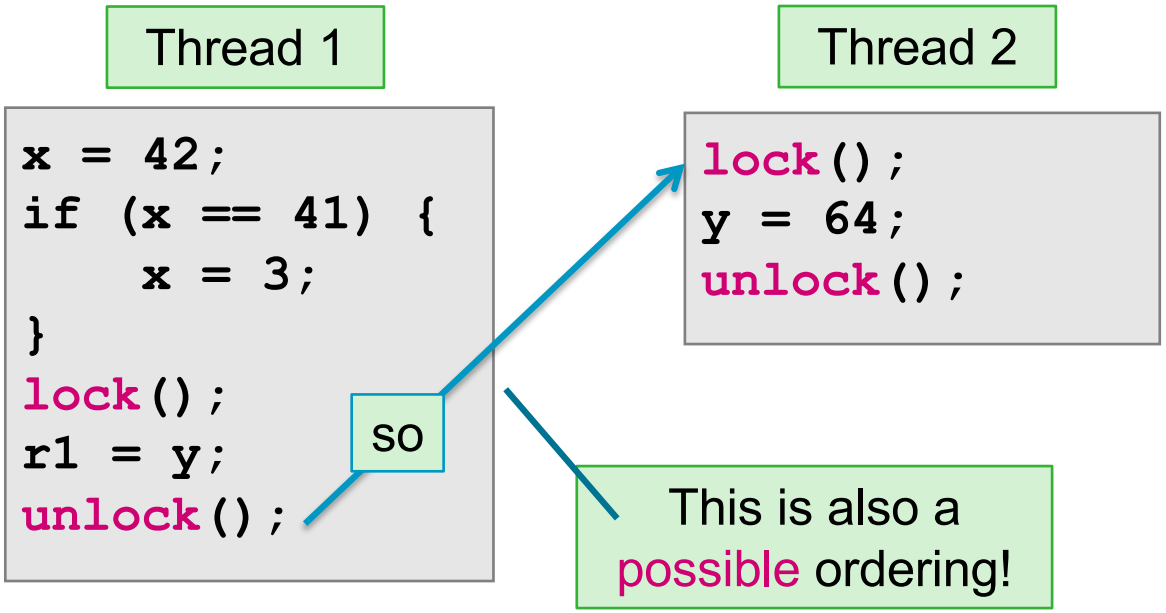
- program order (po)
- synchronisation order (so)

} together this induces a happens before relation — partial order



MEMORY MODELS

… ARE ABOUT EXECUTIONS!



DATA-RACE-FREE GUARANTEE

Any legal execution E of a well-formed data race free program is sequentially consistent

Translation: any data race free program is 'understandable'

Data race freedom:

- If in each sequentially consistent execution of the program, each conflicting pair of actions a and b is ordered by happens-before
  - conflicting pair of actions: access same memory location, and at least one of the accesses is a write

DRF guarantee

If all sequentially consistent executions have no data races then all executions are sequentially consistent

DATA RACE?

EXAMPLE

Thread 1

```
x = 42;
if (x == 41) {
    x = 3;
}
r1 = y;
```

Thread 2

```
y = 64;
if (z == 0) {
    z = y + 5;
}
r2 = x;
```

Data race!

- write and read to `y` are not ordered by happens-before
- write and read to `x` are not ordered by happens-before

DATA RACE?

EXAMPLE

Thread 1

```
x = 42;
if (x == 41) {
    x = 3;
}
lock();
r1 = y;
unlock();
```

Thread 2

```
lock();
y = 64;
unlock();
```

This program has no data races!
All writes and reads to `y` are ordered by happens-before

VOLATILES

WITHIN MEMORY MODELS

Thread 1

```
volatile int x;
x = 42;
if (x == 41) {
    x = 3;
}
r1 = y;
```

Thread 2

```
volatile int y;
y = 64;
if (z == 0) {
    z = y + 5;
}
r2 = x;
```

This program has no data races!

- Every volatile write is (synchronisation) ordered before volatile read
- Now all accesses are ordered by happens-before

DATA RACE?

EXAMPLE

Thread 1

```
Lock l1 =
  new ReentrantLock();
l1.lock();
x++;
l1.unlock();
```

Thread 2

```
Lock l1 =
  new ReentrantLock();
l1.lock();
x++;
l1.unlock();
```

Data race! Both threads use different locks!

DATA RACE?

EXAMPLE

Thread 1

```
int r1;
r1 = x;
if (r1 != 0) {
    y = 42;
}
```

Thread 2

```
int r2;
r2 = y;
if (r2 != 0) {
    x = 42;
}
```

This program has no data races! The conditions will never hold in any sequentially consistent execution

HARDWARE RELAXED MEMORY MODELS

write: store, FLUSH, and atomic load-store instructions

- TSO: total store order
  - a read can complete before an earlier write to a different address
  - a read cannot return the value of a write by another processor unless all processors have seen the write
  - all writes are in total order

- PSO: partial store order
  - reorders write instructions: not necessarily the same as how they are issued by the CPU

When building (the back-end of) a compiler, the hardware memory model might be very relevant

QUESTION 1

```
fn main () {
    let x = vec![5, 8, 10];
    let t = thread::spawn (move || {
        let y = x;
        return y;
    });
    let y = t.join().unwrap();
    println!("{}", x[0] + y[0]);
}
```

What will happen with this program?

a. It will print 10
b. It will print 5
c. It crashes during execution
d. It does not compile

QUESTION 2

```
fn main () {
    let x = 5;
    let t = thread::spawn (move || {
        let y = x;
        return y;
    });
    let y = t.join().unwrap();
    println!("{}", x + y);
}
```

What will happen with this program?

a. It will print 10
b. It will print 5
c. It crashes during execution
d. It does not compile

QUESTION 3

```
let (tx, rx) = mpsc::channel();
thread::spawn(move ||{
    let mut c = 0;
    loop {
        rx.recv().unwrap();
        c = c + 1;
        }
    });
}
```

```
loop {
    let txc = tx.clone();
    thread::spawn(move || {
        txc.send(1).unwrap();
    });
}
```

What is implemented here?

a. Shared counter
b. Broadcast operation
c. This does not compile
d. Semaphore

QUESTION 4

```
public LockFreeStack(long m) {
    this.maxLength     = m;
    this.currentLength = new AtomicInteger(0);
    this.top           = new AtomicReference<Node>();
}

@Override
public void push(T x) {
    if (this.currentLength.get() >= this.maxLength) {
        System.err.println("Stack overflow!");
        return;
    }

    Node newNode;
    Node currentTop;

    do {
        currentTop = this.top.get();
        newNode    = new Node(x, currentTop);
    } while (! this.top.compareAndSet(currentTop, newNode));

    this.currentLength.incrementAndGet();
}
```

What is the problem with this bounded stack implementation?

a. It is not synchronized
b. Length will not always correspond to number of elements
c. Updating the length is not atomic
d. There is no problem

QUESTION 5

```
lock();
x = 3;
unlock();
Print(x);
```

```
lock();
unlock();
x = 3;
Print(x);
```

Would this be an acceptable program transformation if x is a shared variable?

a. Yes
b. Only if we remove the lock()/unlock() pair completely
c. No, this changes when the update on x becomes visible
d. No, this will change the value that is printed

QUESTION 6

```
class Hotel {
  volatile String[] rooms = new String[34];

  //@ requires 0 <= number && number < 34;
  void checkIn(Guest name, int number) {
    rooms[number] = name;
  }

  //@ requires 0 <= number && number < 34;
  void checkOut(int number) {
    rooms[number] = null;
  }
}
```

Which problem do we have here?

a. Data race
b. Deadlock
c. No problem
d. Unnecessary volatile keyword

QUESTION 7

```
public class NonReentrantSpinLock {
        private final int U = 0; L = 1;
        private AtomicInteger sync;
        SpinLock() {
                sync = new AtomicInteger(U);
        }

        void lock() {
                while(!sync.compareAndSet(U, L));
        }

        void unlock() {
                sync.set(U);
}
```

Suppose all threads correctly follow the lock-unlock protocol. Why is it okay to do a set on line 11, instead of a compareAndSet?

a. That is wrong, it should be changed to compareAndSet
b. To make sure the update is visible
c. To avoid data races
d. Because no other thread can change sync when it has value L

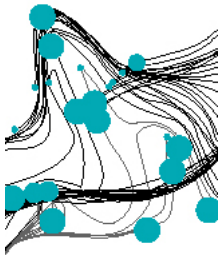
QUESTION 8

Initially: volatile answer = 21, ready = false, x = 0

answer = 42;	||	r1 = x;
ready = true;		if (ready) then
		r1 = answer

What are the possible values of r1 at the end of this program?

a. 0, 21 or 42
b. 0 or 42
c. 42
d. 0



IMPORTANT POINTS

- Atomics
  - wrapper for volatiles
  - makes array elements atomic
  - CAS: compare-and-set/swap
- Lock-free algorithms
  - based on optimistic update pattern
- Wait-free algorithms
  - separate logical and physical update
- Synchronization using atomics
- Memory models: allowed legal behaviors of a concurrent program