NOVA SCHOOL OF
SCIENCE & TECHNOLOGY
COMPUTER SCIENCE DEPARTMENT

Alternative Synchronization Strategies — Transactional Memory —

lecture 25 (2021-06-14)

Master in Computer Science and Engineering

— Concurrency and Parallelism / 2020-21 —

João Lourenço <joao.lourenco@fct.unl.pt>

Alternative Synchronization Strategies

- Contents:
  - – Coarse-Grained Synchronization
  - – Fine-Grained Synchronization
  - – Optimistic Synchronization
  - – Lazy Synchronization
  - – Lock-Free Synchronization
  - – Transactional Memory
- Reading list:
  - – Chapter 10 of the Textbook
  - – Chapter 18 of "The Art of Multiprocessor Programming" by Maurice Herlihy & Nir Shavit *(available at clip)*

Parallel Computing

is here to stay!

And locks are just **not** good enough!

Why locking doesn’t scale?

- **Not Robust**
  - What happens if the thread holding a lock dies?
- Relies on conventions
- Hard to Use
  - Conservative
  - Deadlocks
  - Lost wake-ups
- Not Composable

Why locking doesn't scale?

- Not Robust
- **Relies on conventions**
  - Lock bit and object bits
  - Exists only in programmer's mind
- Hard t
  - Cons
  - Dea
  - Lost v
- Not Composable

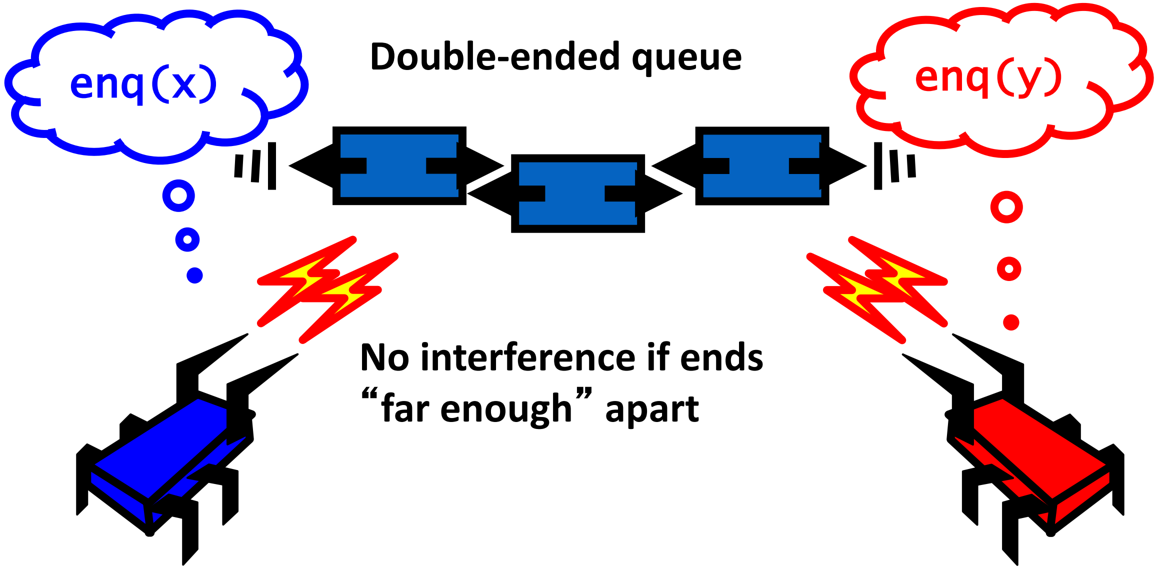
Actual comment from Linux Kernel

(hat tip: Bradley Kuszmaul)

```
/*
 * When a locked buffer is visible to the I/O layer
 * BH_Launder is set. This means before unlocking
 * we must clear BH_Launder,mb() on alpha and then
 * clear BH_Lock, so no reader can see BH_Launder set
 * on an unlocked buffer and then risk to deadlock.
 */
```

Why locking doesn't scale?

- Not Robust
- Relies on conventions
- **Hard to use**
  - Conservative
  - Deadlocks
  - Lost wake-ups
- Not composable



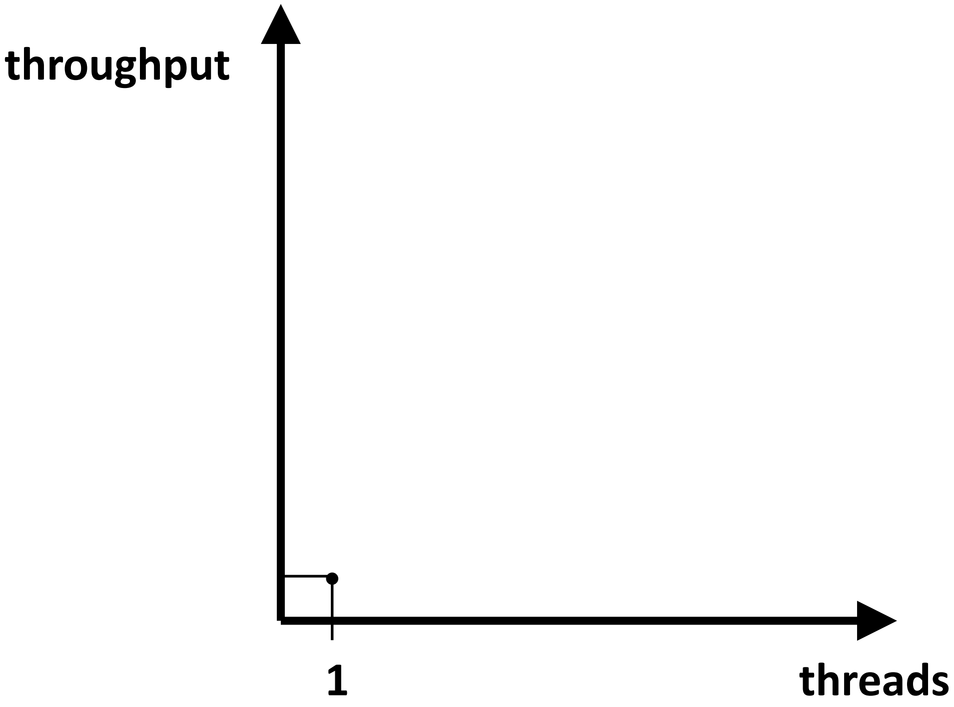
Why locking doesn't scale?

- Not Robust
- Relies on conventions
- Hard to use
  - Conservative
  - Deadlocks
  - Lost wake-ups
- **LOCKS ARE NOT COMPOSABLE!**

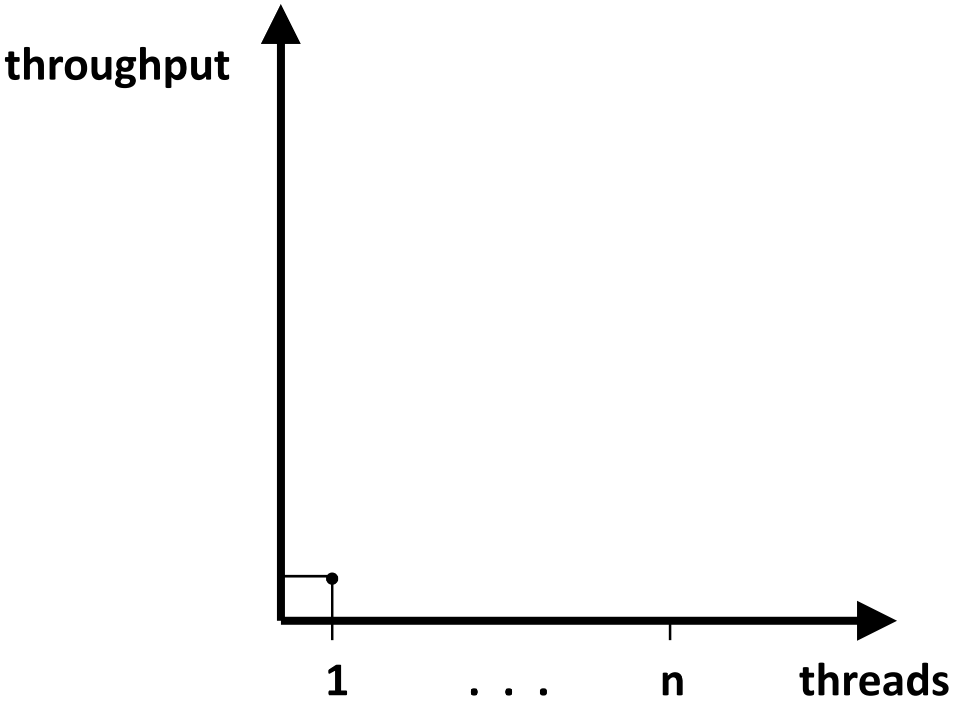
```
class Queue {

    /* private fields... */
    synchronized bool is_empty();
    synchronized bool is_full();
    synchronized bool is_enqueue();
    synchronized bool is_dequeue();
}
```

```
class QueueOperations {
    synchronized void q_transfer(...);
}
```

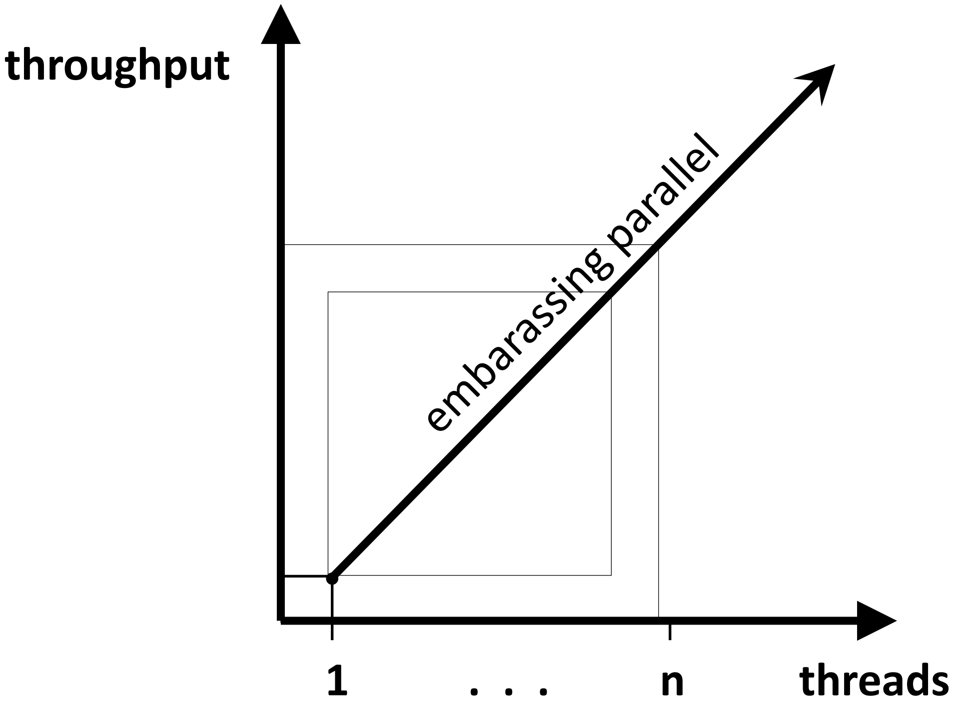
Parallel Throughput



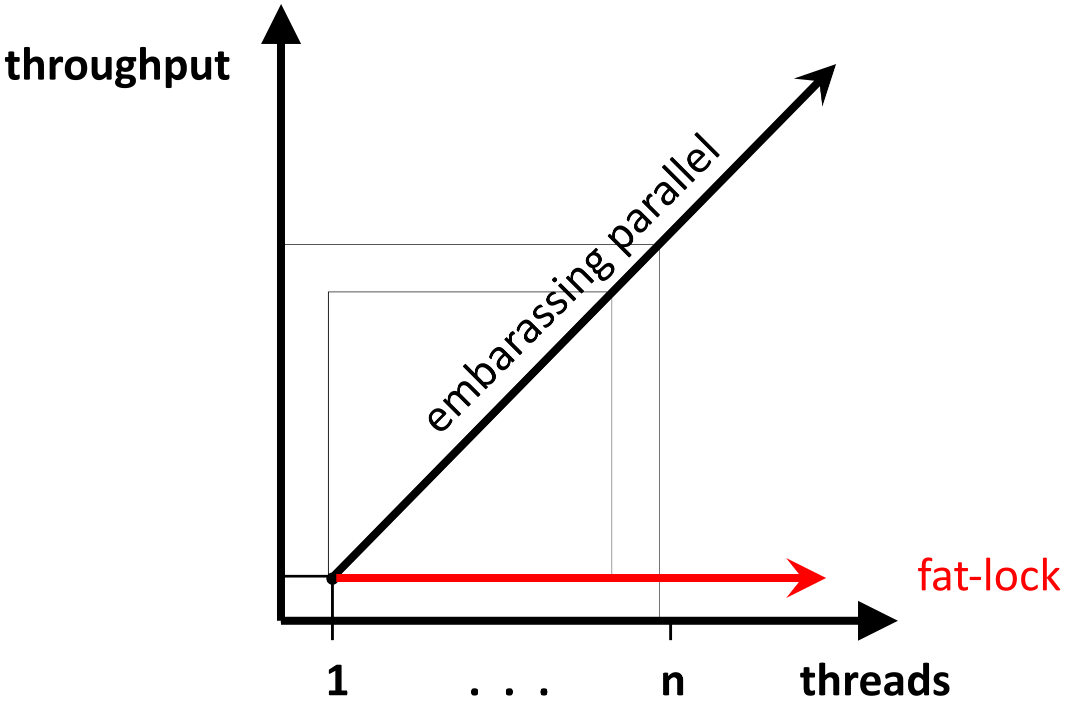
Parallel Throughput



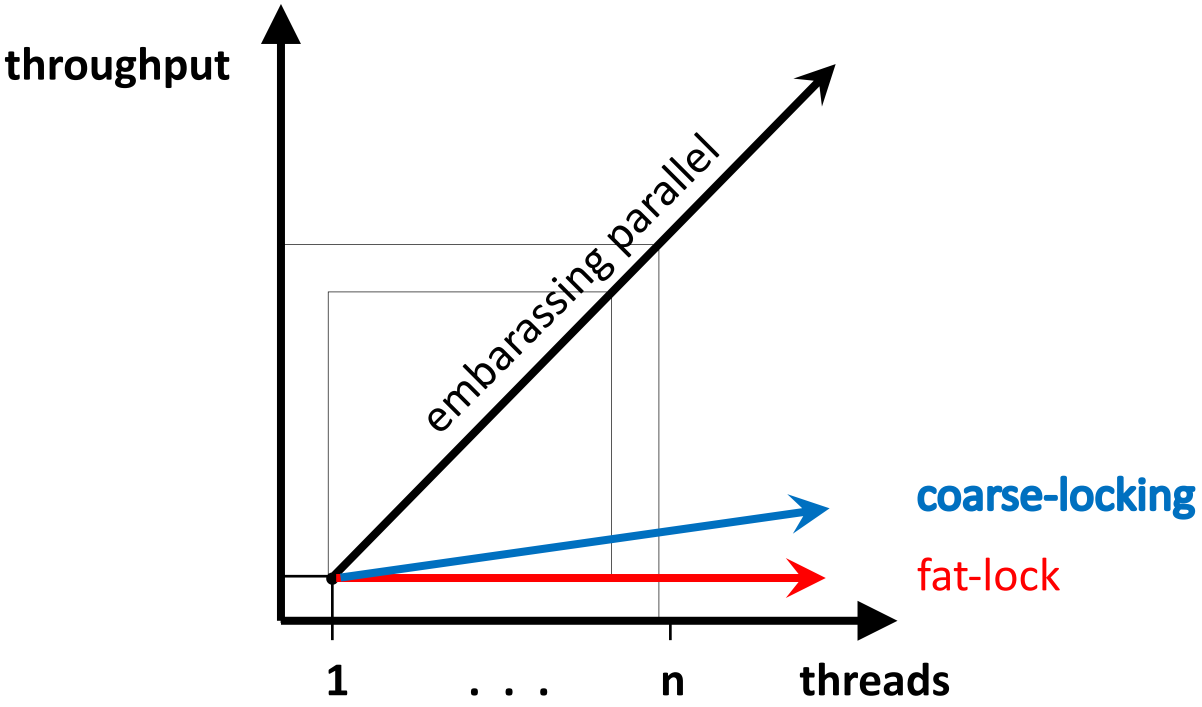
Parallel Throughput



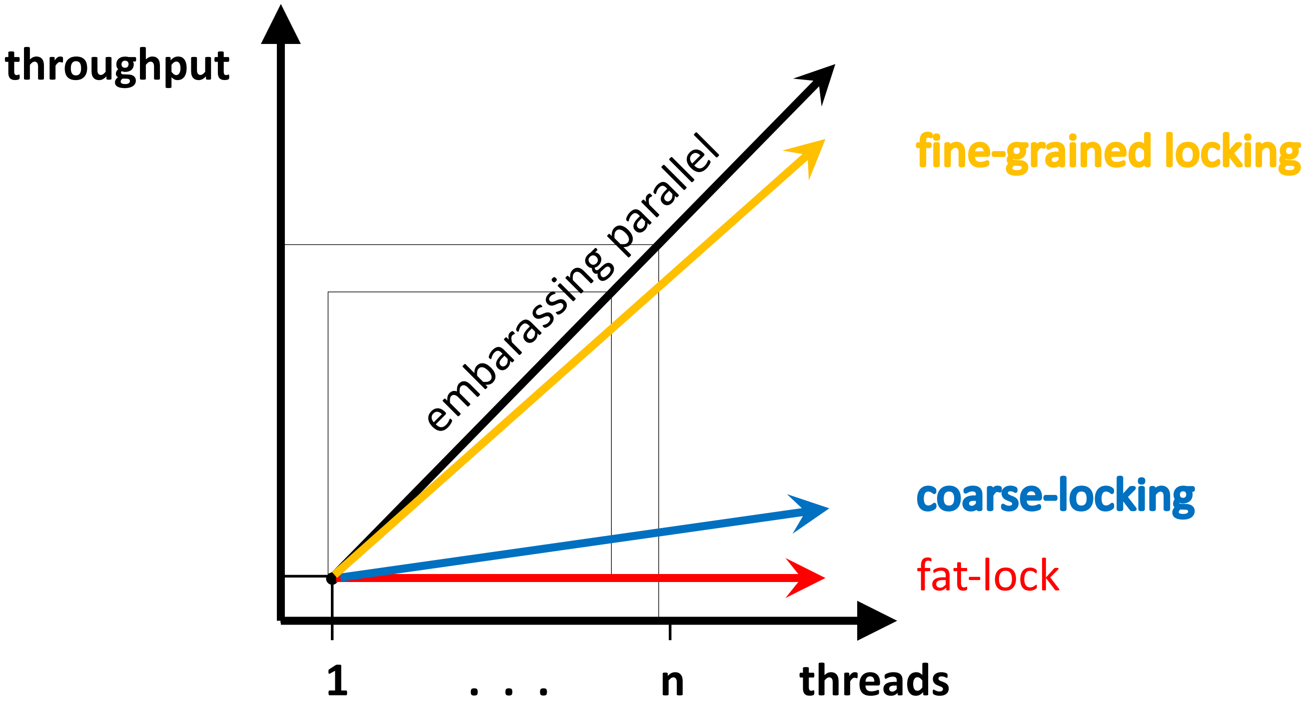
Parallel Throughput



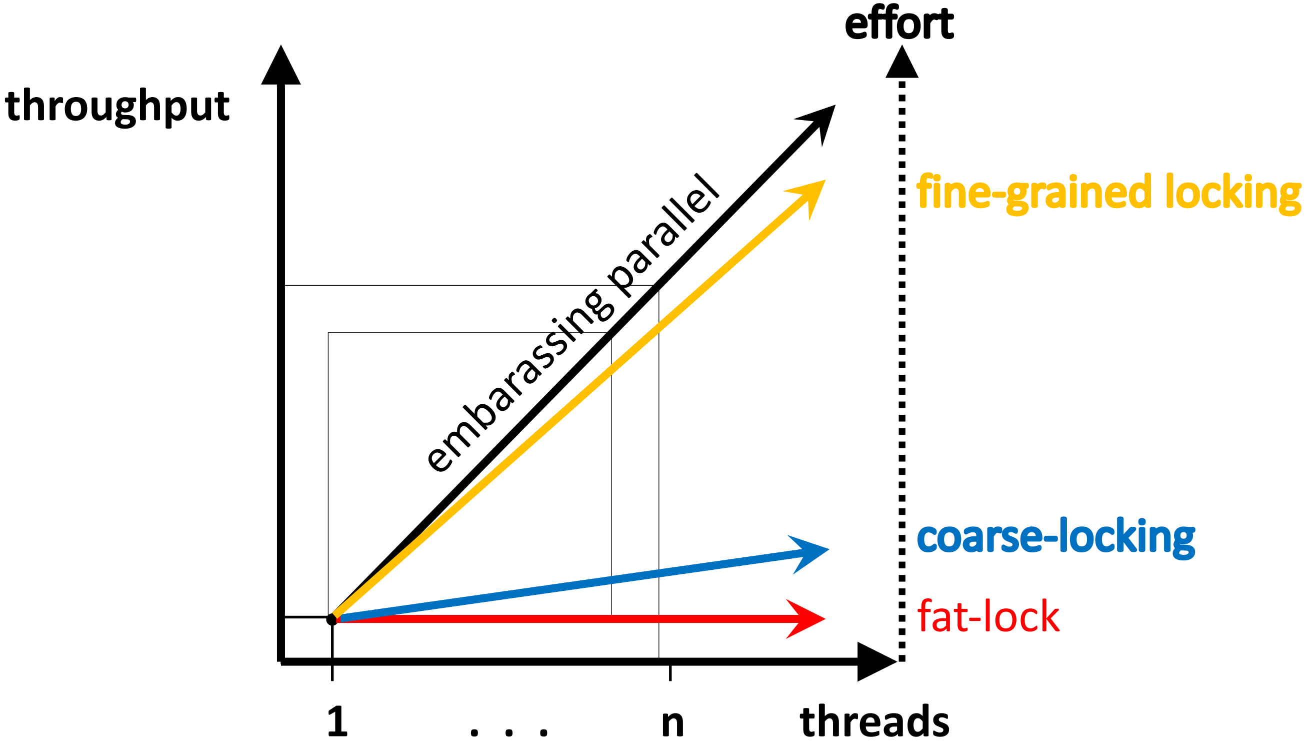
Parallel Throughput



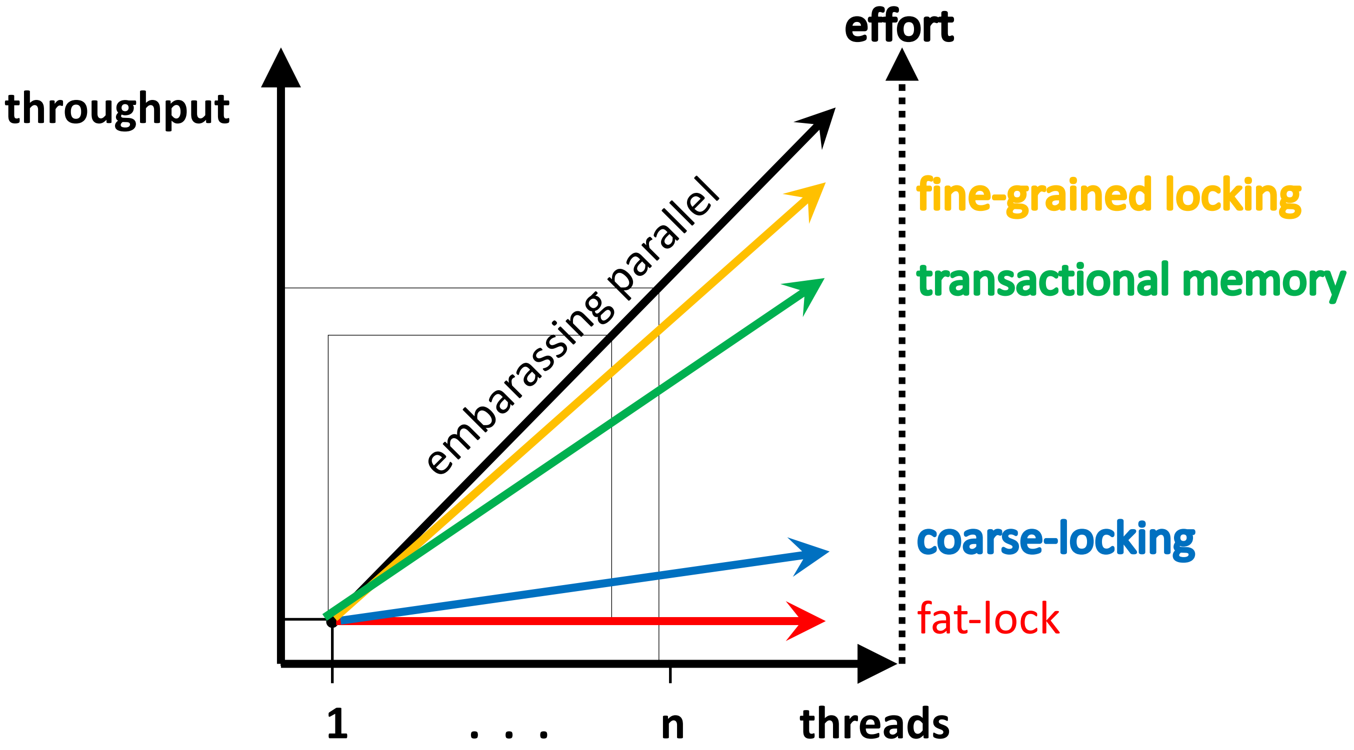
Parallel Throughput



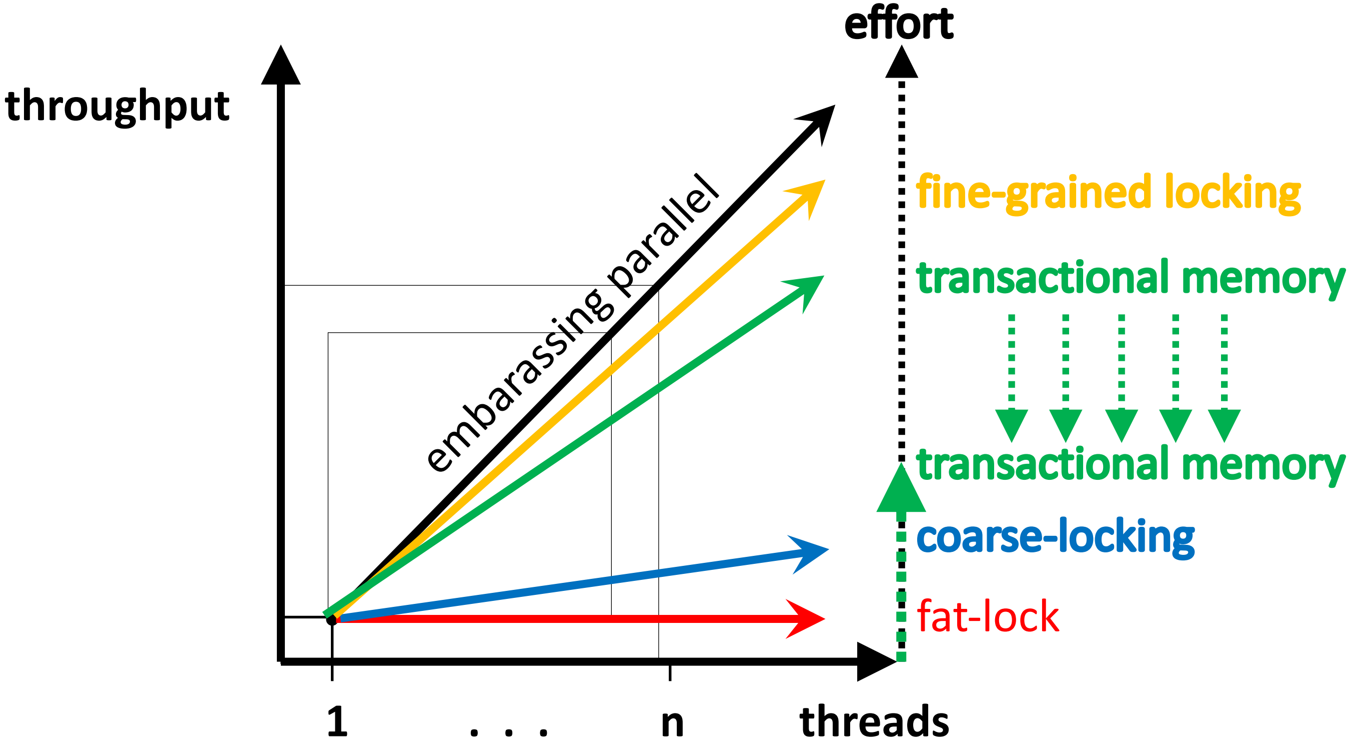
Parallel Throughput



Parallel Throughput



Parallel Throughput



What is

Transactional Memory?

What is transactional memory?

- Is an abstraction for simplifying concurrent programming that uses the concept of “transactions” to synchronize the accesses to shared data in main memory.

Transactional computing

- If critical section locking is superfluous most of the time, aborts are rare
  - Typically threads manipulate different parts of the shared memory
  - Consider, e.g., web server serving pages for different users
- Optimistic approach
  - Instead of assuming that conflicts will happen in critical sections, assume they don't
  - Rely on conflict detection: abort and retry if necessary

- References
  - Lomet 1977, Herlihy & Moss 1993, Shavit & Touitou 1995, Herlihy et al. 2003

Transactional memory

- “**atomic**” ≈ “**transaction**” =>
  => *atomicity, consistency, isolation*

Transactional memory

- “**atomic**” ≈ “**transaction**” =>
  => *atomicity, consistency, isolation*
- Atomicity
  - One-or-nothing
- Consistency
  - Transactions take the program from one correct state into another correct state (assuming a bug-free program)
- Isolation
  - Transactions appear to execute alone in the system
    - i.e., with no interference from other concurrent transactions

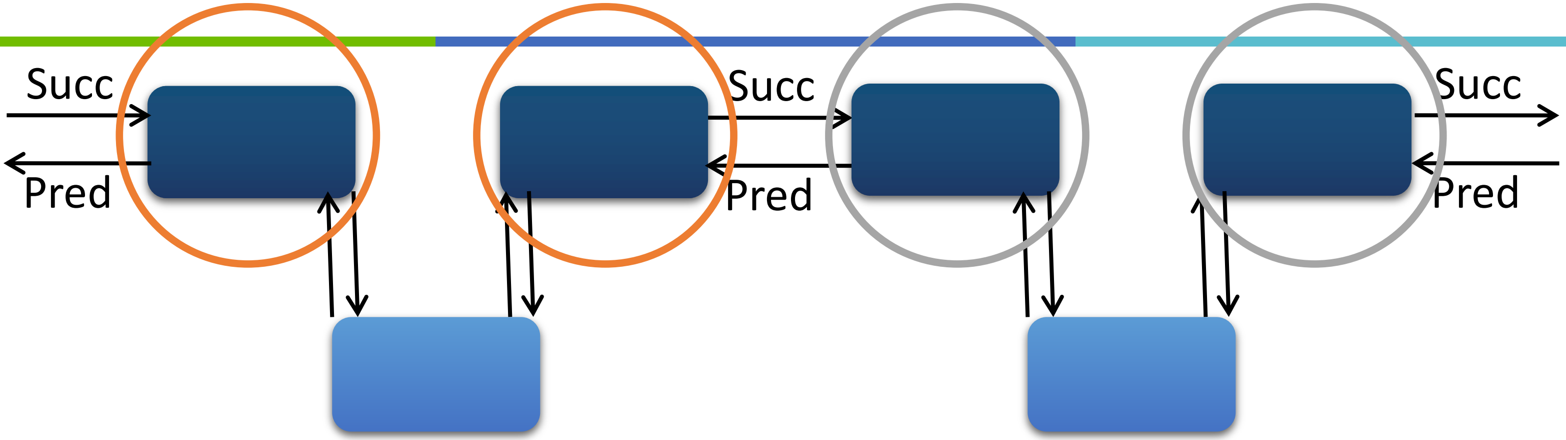
Transactional memory

- “**atomic**” ≈ “**transaction**” =>
  => *atomicity, consistency, isolation*
- Atomicity
  - Commit: takes effect
  - Abort: effects rolled back
    - Usually retried
- Isolation
  - Serializable: any parallel execution of the transactions is equivalent one sequential execution of those same transactions — they always appear to happen in a one-at-a-time order

The Brief History of (S)TM

1995	STM (Shavit,Touitou)
1997	Trans Support TM (Moir)
2003	WSTM (Fraser, Harris)
2003	OSTM (Fraser, Harris)
2003	DSTM (Herlihy et al)
2004	ASTM (Marathe et al)
2004	T-Monitor (Jagannathan…)
2004	Soft Trans (Ananian, Rinard)
2004	Meta Trans (Herlihy, Shavit)
2004	HybridTM (Moir)
2005	Lock-OSTM (Ennals)
2005	McTM (Saha et al)
2005	TL (Dice, Shavit))
2006	AtomJava (Hindman…)
2006	LSA (Riegel et al
2006	DSTM2 (Herlihy, Luchangco)
2006	TL2 (Dice, Shavit, Shalev)
2007	Tanger
2008	*Rock* (Sun)
2009	Deuce (Korland et al)
2013	*Haswell* (Intel)
2013	*PowerPC* (IBM)

Locks:
Insert in a Double Linked List



Transactional memory

```
public void insertNode (node,  precedingNode) {
   atomic {   //means "transaction"
      node.prec = precedingNode;
      node.succ = precedingNode.succ;
      precedingNode.succ.prec = node;
      precedingNode.succ = node;
   }
}
```

Software or Hardware run-time manages the conflicting accesses!

Support for TM run-time(s)

- Hardware

– Sun Rock processor	(abandoned)
– AMD HTM specification	(in simulator)
– IBM BlueGene/Q	(available)
– IBM Power8	(available)
– IBM zEC12	(available)
– Intel Haswell processor	(available)
– Arm Processor V9	(available)

- Software
  - Intel C++ compiler / GNU GCC 4.7+
  - Haskell, Scala and Closure programming languages
  - Java (annotations)

TM Pros & Cons

- Advantages
  - Code blocks are simply marked as atomic/isolated
    - Less concerns about parallelism
  - Run-time ensure the ACI properties
    - Hide away synchronization issues from the programmer
- Disadvantages
  - Oveerhead
  - Successive collisions ➔ repetitive restart ➔ livelocks
  - Memory transactions can not contain certain operations (e.g., I/O operations)
  - Incompatibility with legacy code
  - …

Memory transaction life-cycle

1. Start
2. Access shared data (read / write)
3. (Attempt to) Commit
4. If commit fails, go to 1

Locks vs. Transasctions

```
bool lk-contains(val) {
    int results;
    node_lk *prev, *next;

    curr = set → head;

    next = curr → next;
    while (next → val < val) do {

        curr = next;

        next = curr → next;
    }

    result = (next → val == val);

    return result;
}
```

```
bool lk-contains(val) {
    int results;
    node_lk *prev, *next;

    curr = set → head;

    next = curr → next;
    while (next → val < val) do {

        curr = next;

        next = curr → next;
    }

    result = (next → val == val);

    return result;
}
```

Locks vs. Transasctions

```
bool lk-contains(val) {
    int results;
    node_lk *prev, *next;
    lock(&set → head → lock);
    curr = set → head;
    lock(&curr → next → lock);
    next = curr → next;
    while (next → val < val) do {
        unlock(&curr → lock);
        curr = next;
        lock(&next → next → lock);
        next = curr → next;
    }
    unlock(&curr → lock);
    result = (next → val == val);
    unlock(&next → lock);
    return result;
}
```

```
bool lk-contains(val) {
    int results;
    node_lk *prev, *next;
    lock(&set → head → lock);
    curr = set → head;
    lock(&curr → next → lock);
    next = curr → next;
    while (next → val < val) do {
        unlock(&curr → lock);
        curr = next;
        lock(&next → next → lock);
        next = curr → next;
    }
    unlock(&curr → lock);
    result = (next → val == val);
    unlock(&next → lock);
    return result;
}
```

Locks vs. Transasctions

```
bool lk-contains(val) {
    int results;
    node_lk *prev, *next;
    lock(&set → head → lock);
    curr = set → head;
    lock(&curr → next → lock);
    next = curr → next;
    while (next → val < val) do {
        unlock(&curr → lock);
        curr = next;
        lock(&next → next → lock);
        next = curr → next;
    }
    unlock(&curr → lock);
    result = (next → val == val);
    unlock(&next → lock);
    return result;
}
```

```
bool lk-contains(val) {
    int results;
    node_lk *prev, *next;
    ~~lock(&set → head → lock);~~
    curr = set → head;
    ~~lock(&curr → next → lock);~~
    next = curr → next;
    while (next → val < val) do {
        ~~unlock(&curr → lock);~~
        curr = next;
        ~~lock(&next → next → lock);~~
        next = curr → next;
    }
    ~~unlock(&curr → lock);~~
    result = (next → val == val);
    ~~unlock(&next → lock);~~
    return result;
}
```

Locks vs. Transasctions

```
bool lk-contains(val) {
    int results;
    node_lk *prev, *next;
    lock(&set → head → lock);
    curr = set → head;
    lock(&curr → next → lock);
    next = curr → next;
    while (next → val < val) do {
        unlock(&curr → lock);
        curr = next;
        lock(&next → next → lock);
        next = curr → next;
    }
    unlock(&curr → lock);
    result = (next → val == val);
    unlock(&next → lock);
    return result;
}
```

```
bool lk-contains(val) {
    int results;
    node_lk *prev, *next;
  transaction {
    curr = set → head;
    ~~lock(&curr → next → lock);~~
    next = curr → next;
    while (next → val < val) do {
        ~~unlock(&curr → lock);~~
        curr = next;
        ~~lock(&next → next → lock);~~
        next = curr → next;
    }
    ~~unlock(&curr → lock);~~
    result = (next → val == val);
  }
    return result;
}
```

Locks vs. Transasctions

```
bool lk-contains(val) {
    int results;
    node_lk *prev, *next;
    lock(&set → head → lock);
    curr = set → head;
    lock(&curr → next → lock);
    next = curr → next;
    while (next → val < val) do {
        unlock(&curr → lock);
        curr = next;
        lock(&next → next → lock);
        next = curr → next;
    }
    unlock(&curr → lock);
    result = (next → val == val);
    unlock(&next → lock);
    return result;
}
```

```
bool lk-contains(val) {
    int results;
    node_lk *prev, *next;
  transaction {
    curr = set → head;

    next = curr → next;
    while (next → val < val) do {

        curr = next;

        next = curr → next;
    }

    result = (next → val == val);
  }
    return result;
}
```

Locks vs. Transasctions

```
bool lk-contains(val) {
   int results;
   node_lk *prev, *next;
   lock(&set → head → lock);
   curr = set → head;
   lock(&curr → next → lock);
   next = curr → next;
   while (next → val < val) do {
      unlock(&curr → lock);
      curr = next;
      lock(&next → next → lock);
      next = curr → next;
   }
   unlock(&curr → lock);
   result = (next → val == val);
   unlock(&next → lock);
   return result;
}
```

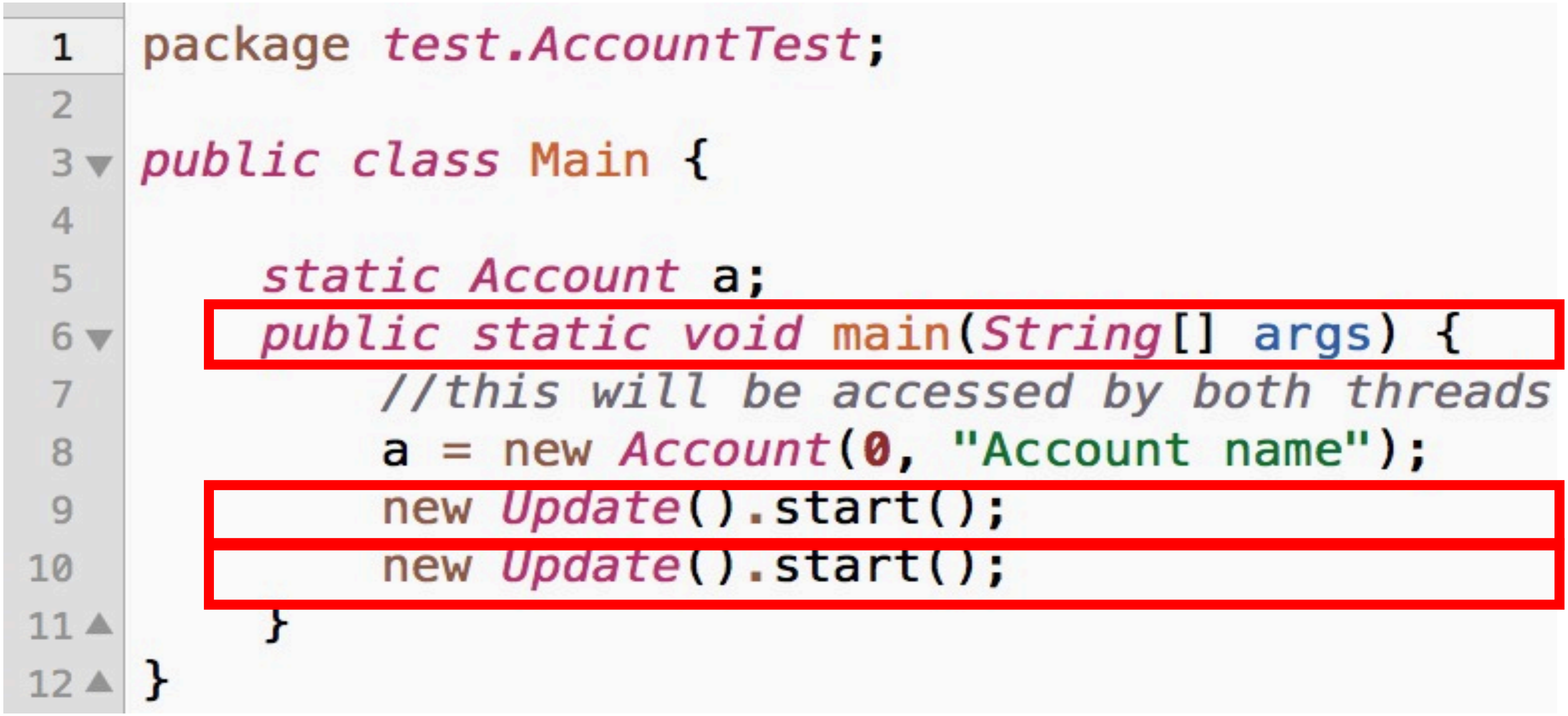
```
bool lk-contains(val) {
   int results;
   node_lk *prev, *next;
 atomic {
   curr = set → head;

   next = curr → next;
   while (next → val < val) do {

      curr = next;

      next = curr → next;
   }

   result = (next → val == val);
 }
   return result;
}
```

Locks vs. Transasctions

```
bool lk-contains(val) {
   int results;
   node_lk *prev, *next;

   curr = set → head;

   next = curr → next;
   while (next → val < val) do {

      curr = next;

      next = curr → next;
   }

   result = (next → val == val);

   return result;
}
```

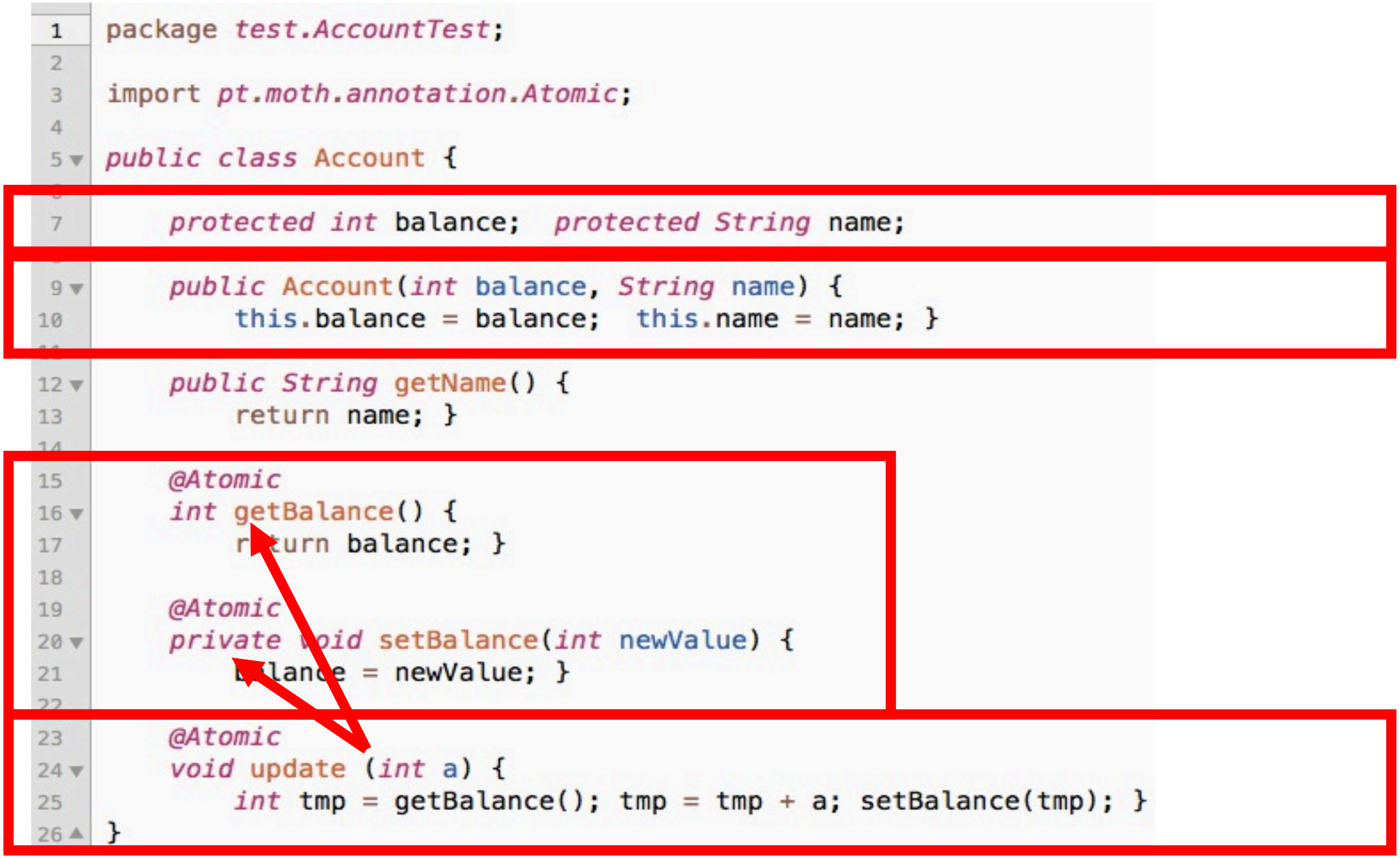
```
bool lk-contains(val) {
   int results;
   node_lk *prev, *next;
 atomic {
   curr = set → head;

   next = curr → next;
   while (next → val < val) do {

      curr = next;

      next = curr → next;
   }

   result = (next → val == val);
 }
   return result;
}
```

An example



```java
package test.AccountTest;

public class Main {

    static Account a;
    public static void main(String[] args) {
        //this will be accessed by both threads
        a = new Account(0, "Account name");
        new Update().start();
        new Update().start();
    }
}
```

An example



```java
package test.AccountTest;

import pt.moth.annotation.Atomic;

public class Account {

    protected int balance;  protected String name;

    public Account(int balance, String name) {
        this.balance = balance;  this.name = name; }

    public String getName() {
        return name; }

    @Atomic
    int getBalance() {
        return balance; }

    @Atomic
    private void setBalance(int newValue) {
        balance = newValue; }

    @Atomic
    void update (int a) {
        int tmp = getBalance(); tmp = tmp + a; setBalance(tmp); }
}
```

An example

```java
package test.AccountTest;

import pt.moth.annotation.Atomic;

public class Account {

    protected int balance;  protected String name;

    public Account(int balance, String name) {
        this.balance = balance;  this.name = name; }

    public String getName() {
        return name; }

    @Atomic
    int getBalance() {
        return balance; }

    @Atomic
    private void setBalance(int newValue) {
        balance = newValue; }

    @Atomic
    void update (int a) {
        int tmp = getBalance(); tmp = tmp + a; setBalance(tmp); }
}
```

An example

```
package test.AccountTest;

import pt.moth.annotation.Atomic;

public class Account {

    protected int balance;  protected String name;

    public Account(int balance, String name) {
        this.balance = balance;  this.name = name; }

    public String getName() {
        return name; }

    @Atomic
    int getBalance() {
        return balance; }

    @Atomic
    private void setBalance(int newValue) {
        balance = newValue; }

    @Atomic
    void update (int a) {
        int tmp = getBalance(); tmp = tmp + a; setBalance(tmp); }
}
```

```
package test.AccountTest;

public class Main {

    static Account a;
    public static void main(String[] args) {
        //this will be accessed by both threads
        a = new Account(0, "Account name");
        new Update().start();
        new Update().start();
    }
}
```

```
package test.AccountTest;

import java.util.Random;

public class Update extends Thread {
    public void run() {
        while(true){
            Random r = new Random();
            int n = r.nextInt();
            //Example.a.update(123);
            Main.a.update(n);
        }
    }
}
```

The END