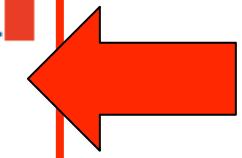
SPP (Synchro et Prog Parallèle)

Unit 2: Locks

François Taïani

Revisiting Unit 1's problem

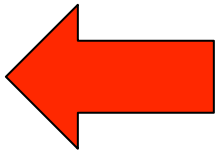
```
public class MultiThreadedCounter extends Thread {

  static long count = 0;

  public void run() {
    for(int i=1; i<=10000000; i++) {
      MultiThreadedCounter.count++ ;
    } // EndFor
  } // EndMethod run

  public static void main(String[] args) {
    Thread t1 = new MultiThreadedCounter();
    Thread t2 = new MultiThreadedCounter();
    t1.start();
    t2.start();
  }

} // EndClass MultiThreadedCounter
```



Revisiting Unit 1's problem

- Key problem: ++ is not atomic
  - ➔ implemented as several byte code instructions
  - ➔ the 2 threads overwrite each other's results

```
public class MultiThreadedCounter extends Thread {

  static long count = 0;

  public void run() {
    for(int i=1; i<=10000000; i++) {
      MultiThreadedCounter.count++ ;
    } // EndFor
  } // EndMethod run

  public static void main(String[] args) {
    Thread t1 = new MultiThreadedCounter();
    Thread t2 = new MultiThreadedCounter();
    t1.start();
    t2.start();
  }

} // EndClass MultiThreadedCounter
```



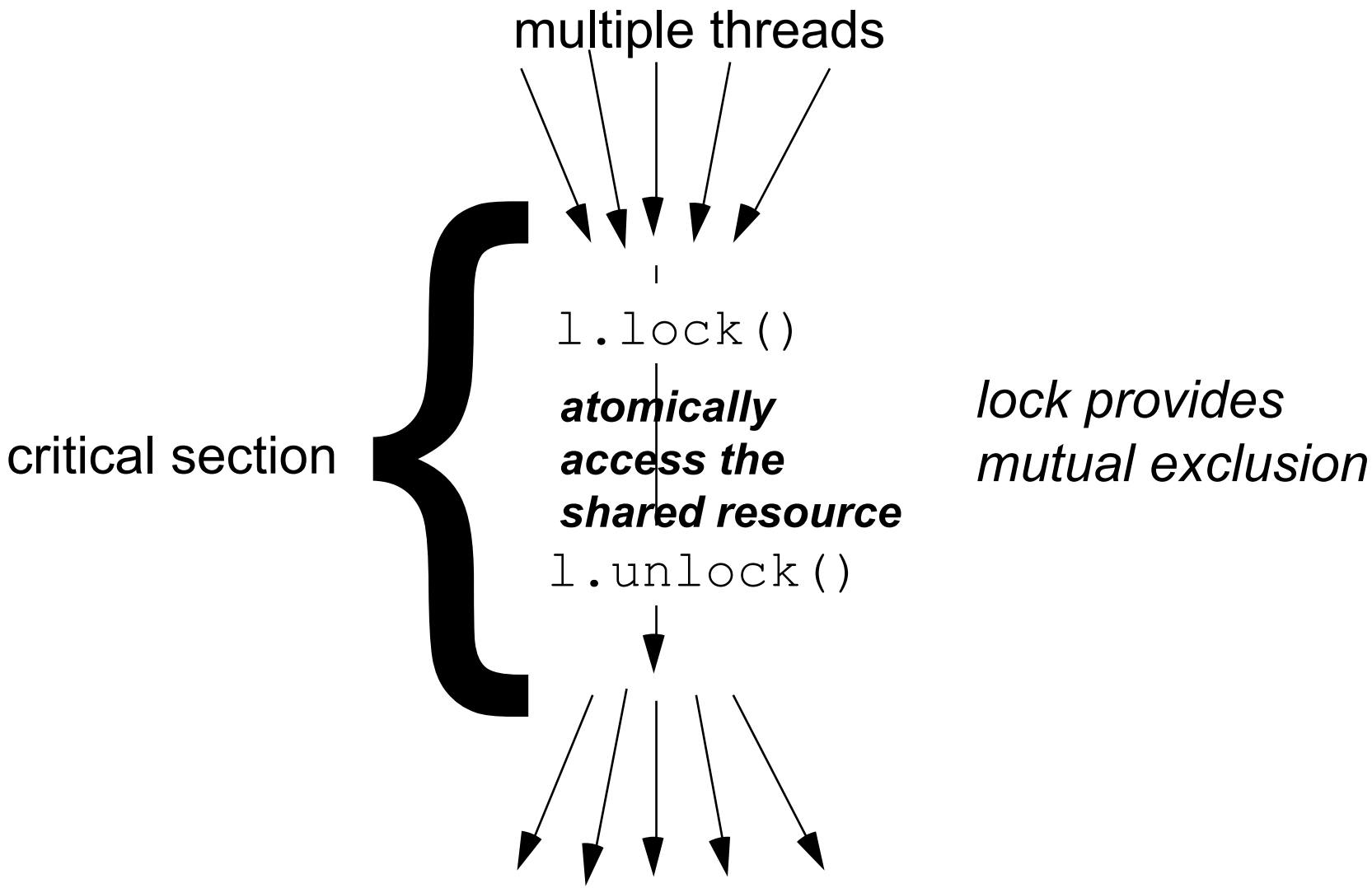
Solution

- Use locks!
- What is a lock?
  - ➔ a runtime structure (in Java an object)
  - ➔ can be in 2 states: “locked” / “unlocked”
  - ➔ state toggled with 2 methods: lock() and unlock()
    - other names are possible: e.g. take() and release()
  - ➔ the last thread to return from lock() “owns” the lock
  - ➔ only one thread at a time can own the lock
    - others attempting to lock it are kept blocked

Locks and Mutual Exclusion

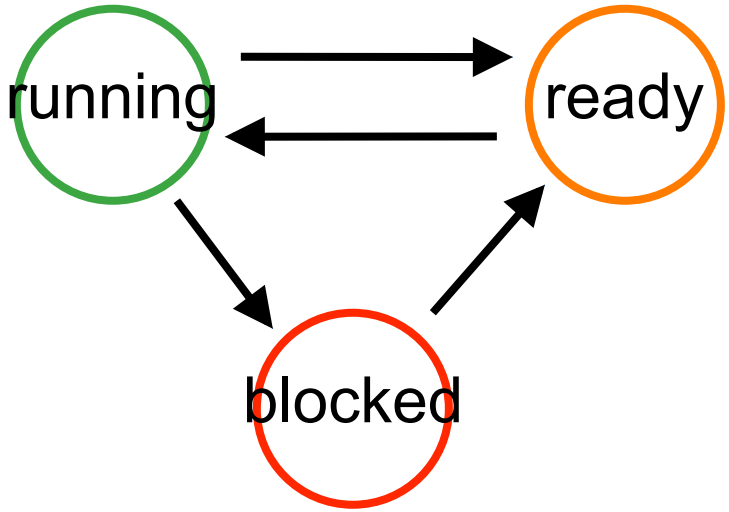
- Locks create “safe regions” of code
  - ➔ invariant:
    “*only one thread at a time in the regions of the lock*”
  - ➔ known as “critical sections”
- If thread t calls l.lock() when l is already locked
  - ➔ t remains blocked in call until lock is unlocked
  - ➔ when lock unlocked: one of the blocked threads unblocked
- Similar to a lock on a bathroom
  - ➔ l.lock() same as: try open the door
    if unlocked, enter and lock; if locked, wait until unlocked
  - ➔ except ≥ 1 regions can be associated with same lock

Locks & Mutual Exclusion



Locks and OS Scheduler

- Locks often implemented at OS level
  - ➔ OS scheduler decides which process / thread runs when
  - ➔ based on heuristics, priorities, etc. (see OS module)
- Each process / thread can be in 3 states
  - ➔ running (has CPU), ready (can run), and blocked



Locks and Scheduler

- Switch running / ready
  - ➔ known as a context switch
  - ➔ goal: share n cores between m >> n threads / processes
  - ➔ more in OS course
- Blocked state
  - ➔ process / thread cannot run: e.g. waiting on I/O or lock
- If lock l already taken
  - ➔ l.lock() puts current thread in blocked state
- When thread that has lock calls l.unlock()
  - ➔ **one** of the threads blocked on lock put in “ready” state

Fairness of Locks

- If multiple threads are waiting for a lock
  - ➔ who gets the lock next is non-deterministic
  - ➔ depends on heuristics of underlying OS scheduler
  - ➔ or result of hardware-level scheduling in multicores

- Lock implementation possible where
  - ➔ some threads get lock preferably when competing (unfair)
  - ➔ opposite property: fairness
  - ➔ important property when looking at lock implementations

Fairness of Locks

- Example
  - ➔ server application, accepts 2 types of requests: increment and decrement

```
shared var  count int = 0
shared lock l

request increment is
begin
  l.lock
  count++
  l.unlock
end

request decrement is
begin
  l.lock
  count--
  l.unlock
end
```

  - ➔ If always mix of 2 requests waiting to take the lock l
  - ➔ both should get lock as often as the other

Locks, Safety & Liveliness

- Locks prevents bad situation from happening
  - ➔ i.e. no more than one thread in critical section
- Known as “safety” property
  - ➔ nothing bad happens
- However one key danger: that nothing happens at all
  - ➔ known as “deadlock”: so many locks, program is blocked
- Avoiding this: known as “liveliness” property
  - ➔ something happens
- Continuous tension between safety and liveliness
  - ➔ locks help with safety, but can hurt liveliness

Locks in Java

- (Caveat: Not standard Java synchronisation
  - ➔ see ‘synchronized’ keyword and monitors in Unit 5)
- Interface

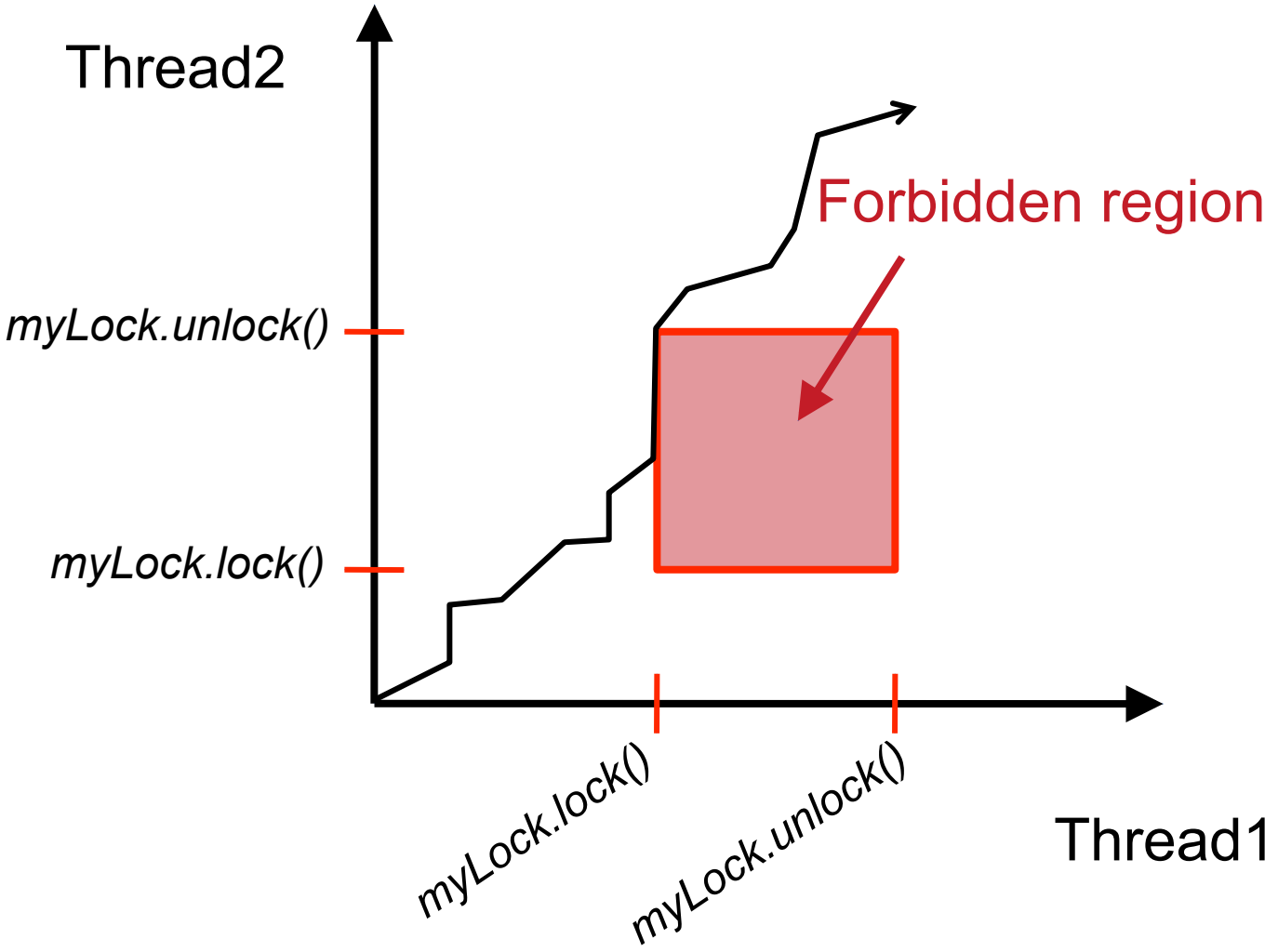
  java.util.concurrent.locks.Lock
- Implementation:

  java.util.concurrent.locks.ReentrantLock
  - ➔ we will see what “Reentrant” mean
- lock and unlock methods + others variants
  - ➔ tryLock(..), lockInterruptibly()

```
import java.util.concurrent.locks.ReentrantLock;

public class MultiThreadedCounterWithLock extends Thread {

  static long count = 0;
  static ReentrantLock myLock = new ReentrantLock();

  public void run() {
    for(int i=1; i<=10000000; i++) {
      myLock.lock();
      MultiThreadedCounterWithLock.count++ ;
      myLock.unlock();
    } // EndFor
  } // EndMethod run

  public static void main(String[] args) {
    Thread t1 = new MultiThreadedCounterWithLock();
    Thread t2 = new MultiThreadedCounterWithLock();
    t1.start();
    t2.start();
  }

} // EndClass MultiThreadedCounterWithLock
```

Revisiting our Example

Graphical Representation



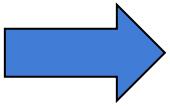
Locks and Performance

```
AdminMacBook ftaiani [SIMPLIFIED] $ time java MTCounter
java MTCounter 0.31s user 0.08s system 117% cpu 0.335 total

AdminMacBook ftaiani [SIMPLIFIED] $ time java MTCounterLock
java MTCounterLock 1.53s user 0.11s system 127% cpu 1.289 total
```

- Locks introduce a performance penalty
  - ➔ bookkeeping, scheduling, blocking, unblocking
- But do we pay this price all the time?

A variant

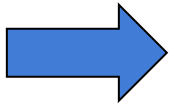
- Removing contention: count private to each thread



```
public class MTCountNoContention extends Thread {

  long count = 0;

  public void run() {
    for(int i=1; i<=10000000; i++) {
      this.count++ ;
    } // EndFor
  } // EndMethod run

  public static void main(String[] args) {
    Thread t1 = new MTCountNoContention();
    Thread t2 = new MTCountNoContention();
    t1.start();
    t2.start();
  }

} // EndClass MTCountNoContention
```

And the lock version

- count and lock private to each thread



```
public class MTCountLockNoContention extends Thread {

  long count = 0;
  ReentrantLock myLock = new ReentrantLock();

  public void run() {
    for(int i=1; i<=10000000; i++) {
      this.myLock.lock();
      this.count++ ;
      this.myLock.unlock();
    } // EndFor
  } // EndMethod run

  public static void main(String[] args) {
    Thread t1 = new MTCountLockNoContention();
    Thread t2 = new MTCountLockNoContention();
    t1.start();
    t2.start();
  }

} // EndClass MTCountLockNoContention
```

But …

- We no longer need locks here:
  - ➔ each thread has its own count variable

- So why would we use locks?
  - ➔ to test the impact of contention of performance
  - ➔ more generally as a form of “defensive programming”

- Defensive Programming
  - ➔ my code (classes) will be used (and abused) by others
  - ➔ hope for the best, plan for the worse

The results

```
AdminMacBook ftaiani [SIMPLIFIED] $ time java MTCntNoCont
java MTCntNoCont 0.33s user 0.08s system 121% cpu 0.340 total

AdminMacBook ftaiani [SIMPLIFIED] $ time java MTCntLockNoCont
java MTCntLockNoCont 1.44s user 0.09s system 166% cpu 0.921 total
```

- Still some substantial cost (+170%)
  - ➔ but not as bad as with contention (+284%)

- Note: these Java locks not particularly optimised
  - ➔ The linux kernel offers fast mutex (futex)
    futex with no contention: almost no impact

Analysing a Code with Locks

- Consider the following program (pseudo-code)

```
shared lock printer
shared lock network

thread t1 is
begin
  printer.lock()
  network.lock()
  network.unlock()
  printer.unlock()
end
```

```
thread t2 is
begin
  network.lock()
  printer.lock()
  printer.unlock()
  network.unlock()
end
```

Java Version

- The two locks as static global variables

```
public class BuggyWithLocks {

  public static ReentrantLock printer = new ReentrantLock();
  public static ReentrantLock network = new ReentrantLock();
```

Java Version

- The 2 threads extending class Thread

```
class Thread1 extends Thread {
  public void run() {
    for(int i=1; i<=10000000; i++) {
      BuggyWithLocks.printer.lock();
      BuggyWithLocks.network.lock();
      // work with printer and network
      BuggyWithLocks.network.unlock();
      BuggyWithLocks.printer.unlock();
    }
  }
}
```

```
class Thread2 extends Thread {
  public void run() {
    for(int i=1; i<=10000000; i++) {
      BuggyWithLocks.network.lock();
      BuggyWithLocks.printer.lock();
      // work with printer and network
      BuggyWithLocks.printer.unlock();
      BuggyWithLocks.network.unlock();
    }
  }
}
```

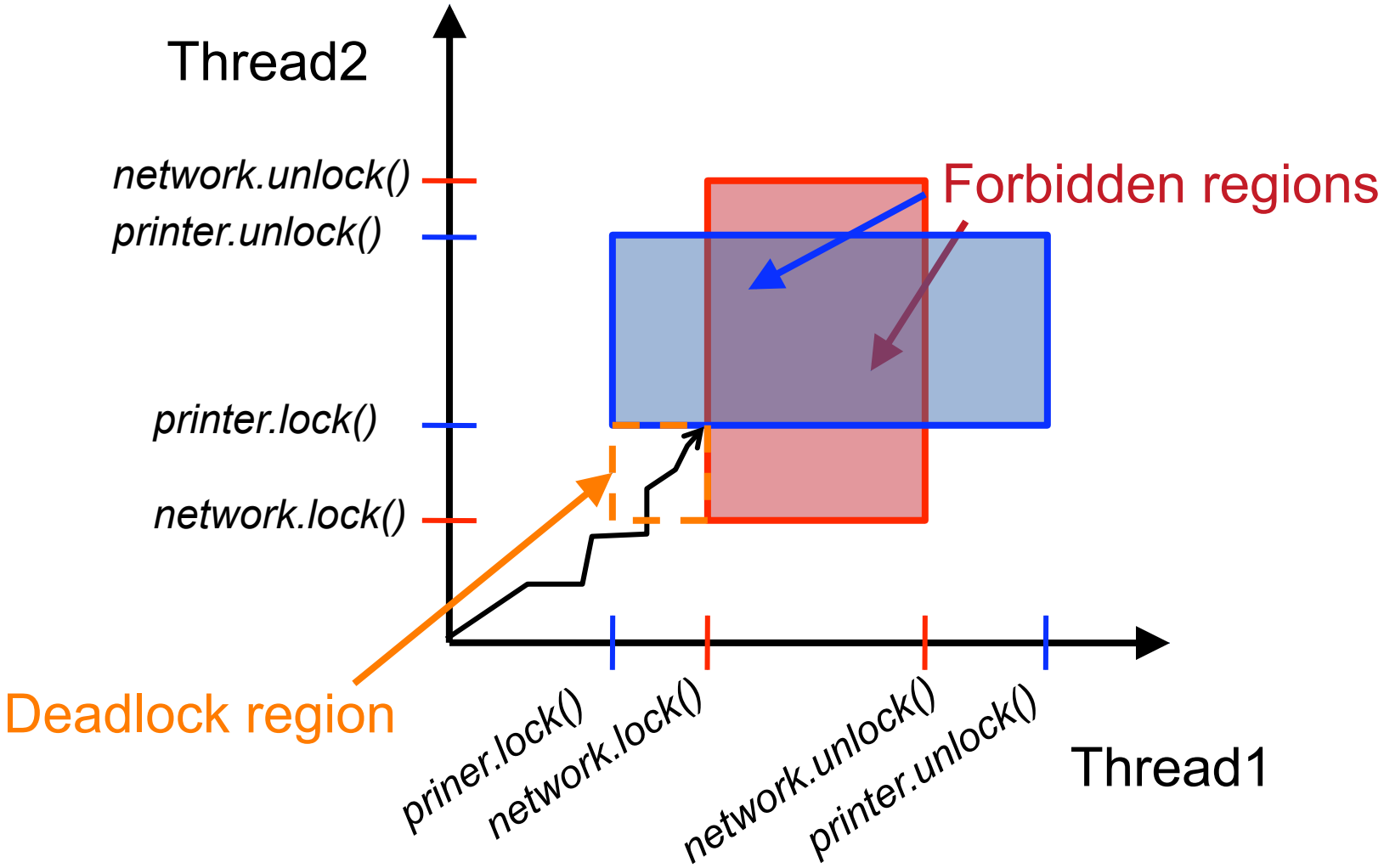
Java Version

- All started in method main

```
public static void main(String[] args) {
   Thread t1 = new Thread1();
   Thread t2 = new Thread2();
   t1.start();
   t2.start();
}
```

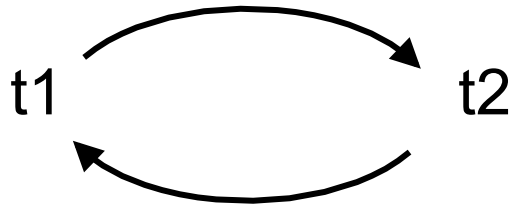
- Demonstration of execution
  - ➔ Quiz: Where is the problem?
  - ➔ Give a graphical representation of the problem.

Graphical Representation



Deadlocks

- t1 is waiting for t2 to release the network lock
- but t2 is waiting for t1 to release the printer lock



- Here only 2 threads
  - ➔ but could involve any number
  - ➔ t1 ➔ t2 ➔ t3 ➔ … ➔ tn ➔ t1
- Possible to detect (cycle detection)
  - ➔ but performance hit + what to do then?

Deadlocks

- Typical case of loss of liveliness
  - ➔ system is safe (does not enter any forbidden state)
  - ➔ but no progress is made (no liveliness)
- Seems easy to avoid
  - ➔ always take locks in same order
- But harder to achieve that it appears
  - ➔ reason: locks hidden in libraries / function calls

Hidden Locks

```
shared lock printer
shared lock network

thread t1 is
begin
  printer.lock()
  foo()
  printer.unlock()
end
```

```
thread t2 is
begin
  network.lock()
  bar()
  network.unlock()
end
```

- Apparently no problem, except
  - ➔ foo() uses the network (e.g. for logging)
  - ➔ and bar connects to the printer (e.g. to get font info)
  - ➔ result: deadlock!!

Advanced locks

- Consider the following code

```
shared lock printer

thread t1 is
begin
  printer.lock()
  bar()
  printer.unlock()
end
```

```
function bar is
begin
  printer.lock()
  // do something
  printer.unlock()
end
```

➔ What will happen?

Reentrant Locks

- Aka recursive locks
- The previous program
  - ➔ blocks with plain (non-recursive) locks
  - ➔ thread t1 is deadlocked with itself!
- Recursive locks: use a counter
  - ➔ incremented each time thread gets lock
  - ➔ decremented each time thread releases lock
  - ➔ lock freed when counter gets to zero
- Are all locks reentrant / recursive?
  - ➔ no: POSIX locks (“mutex”) are not by default
  - ➔ special option to be used

Shared Locks

- Normal locks can be bit blunt at times
- Consider following object
  - ➔ 2 counters: a and b
  - ➔ 2 ops: get sum a+b, or transfer an amount from a to b

```
int a = 1000
int b = 0

method getSum is
begin
  return a+b
end
```

```
method transfer(int x) is
begin
  a = a - x
  b = b + x
end
```

One Solution: Normal Locks

```
int a = 1000
int b = 0
lock l

method getSum is
begin
  l.lock()
  int result = a + b
  l.unlock()
  return result
end
```

```
method transfer(int x) is
begin
  l.lock()
  a = a - x
  b = b + x
  l.unlock()
end
```

- What is the possible downside of this code?

Normal Locks

- Previous solution
  - ➔ prevents transfer() to interfere with getSum()
  - ➔ but also forbids concurrent executions of getSum()
- If getSum() invoked by many threads: performance hit
- Solution: use read/write locks
  - ➔ aka as shared locks
- 2 lock operations
  - ➔ lock_read: locked in reading, multiple threads allowed
  - ➔ lock_write: locked in writing, exclusive access

With Shared Locks

```
int a = 1000
int b = 0
rw_lock l

method getSum is
begin
  l.lock_read()
  int result = a + b
  l.unlock()
  return result
end
```

```
method transfer(int x) is
begin
  l.lock_write()
  a = a - x
  b = b + x
  l.unlock()
end
```

- Concurrent execution of getSum() now allowed

Summary

Introduction to principle and use of locks

- What are locks?
- What property to they provide?
- What is their impact on performance?
- Danger in using locks: Deadlocks
- Advanced locks: Recursive locks, Read/Write locks

Next session: How can locks be implemented?