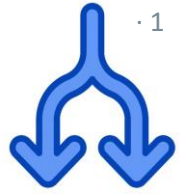
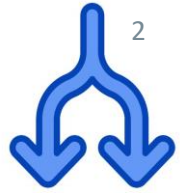


Practical Concurrent and Parallel Programming I

Intro to Concurrency and Mutual Exclusion

Raúl Pardo, Jørgen Staunstrup

Agenda



- Course General Info
- Introduction to Concurrency
- Java Threads
- Mutual Exclusion
- Java Locks

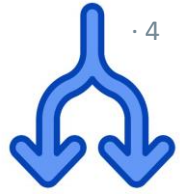
Teachers

- Course manager: Raúl Pardo
  - PhD from Chalmers University 2017
  - Postdoc at Inria 2017 & ITU 2019
  - Research interest: Privacy & Security,
    Formal Methods, Probabilistic Reasoning, Concurrency

- Co-teacher: Jørgen Staunstrup
  - He will introduce himself shortly

- Teaching Assistants
  - Holger Stadel Borum
  - Amund Ranheim Lome
  - You will meet them after the lecture

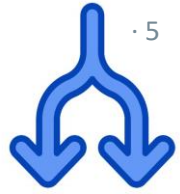
Standard weekly plan



- Lectures
  - Mondays 10-12
  - We publish the readings for the lecture a week before the lecture (approx.)

- Exercise sessions
  - Mondays 12-14
  - Every week we publish a set of exercises covering the material of the lecture

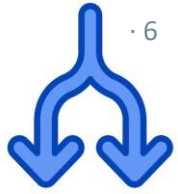
Course Info Online



- The learnIT website (https://learnit.itu.dk/course/view.php?id=3020335) contains information about the course and submission links for the assignments

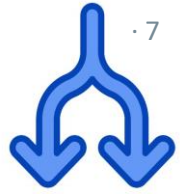
- The material of the course is hosted in our github repository (https://github.itu.dk/jst/PCPP2021-public)
  - Lecture slides
  - Readings
  - Example code
  - Exercises
  - Accompanying code for the exercises

Exercises



- You are expected to work on **groups of 2/3 people**
  - Today in the exercise session we will start by forming groups with the help of TAs
- **Solutions are submitted bi-weekly**
  - Every two weeks you submit the solution to the two exercise sets of the two previous weeks
- Assignments submissions **on Mondays before the lecture**
  - We made sure to minimize clashing with other courses in the CS programme
  - An assignment is hand-in including solutions to two exercise sets
- **Oral Feedback and Assessment**
  - Feedback and assessment of assignments if provided in oral sessions
  - You must **book an oral feedback slot with a TA/teacher using the scheduler in learnIT**
  - Today in the exercise session you can already book a slot (https://learnit.itu.dk/mod/scheduler/view.php?id=143279)
- **In total there are 12 exercise sets, you must pass 10 to be entitled to take the exam**
- Exercises are divided into mandatory and challenging
  - Doing the challenging exercises will increase your chances of getting high marks in the exam
  - Challenging exercises are optional

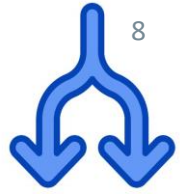
Questions and Answers Forum



- The main channel of communication is the Questions and Answers Forum in LearnIT
  - https://learnit.itu.dk/mod/hsuforum/view.php?id=143401
  - **We strongly encourage you to use the forum!**
  - We will check the forum regularly
  - But, we also encourage you to answers questions yourself!
    - The forum is meant to be a discussion platform to boost learning and share knowledge

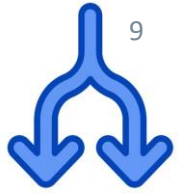
- The only constraint: *do not directly post solutions to exercises*

Exam



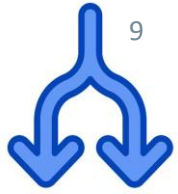
- Oral exam
  - We will ask questions covering all the content in the readings and lecture slides

About Jørgen



Datalog/Computer scientist Aarhus University: 1975

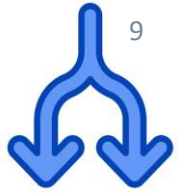
About Jørgen



Datalog/Computer scientist Aarhus University: 1975

Taught first course on Concurrency: 1978 (USC, Los Angeles)

About Jørgen

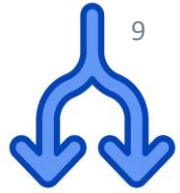


Datalog/Computer scientist Aarhus University: 1975

Taught first course on Concurrency: 1978 (USC, Los Angeles)

It was an on-line course !!

About Jørgen



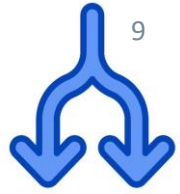
Datalog/Computer scientist Aarhus University: 1975

Taught first course on Concurrency: 1978 (USC, Los Angeles)

It was an on-line course !!

Joined ITU in 2001, retired in 2014

About Jørgen



Datalog/Computer scientist Aarhus University: 1975

Taught first course on Concurrency: 1978 (USC, Los Angeles)

It was an on-line course !!

Joined ITU in 2001, retired in 2014

Teaching MSc course Mobile App Development at ITU since 2016

Concurrency

It takes one person 2 hours to dig 2 meters of ditch.
How long will it take 2 people to dig 4 meters of ditch?

What if they only have one shovel?

What if they have to dig a hole (and not a ditch)?

How fast can a 100 persons dig 1 meter of ditch?

…

The first computers were sequential

In my first years at university, we used the Danish Gier (1965)

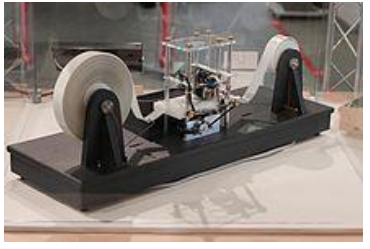
The first computers were sequential

The Turing machine (1936) - a mathematical model

In my first years at university, we used the Danish Gier (1965)

The first computers were sequential

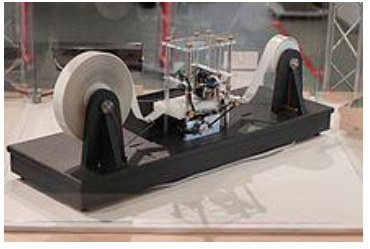
The Turing machine (1936) - a mathematical model



In my first years at university, we used the Danish Gier (1965)

The first computers were sequential

The Turing machine (1936) - a mathematical model

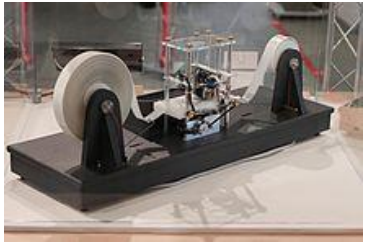


Eniac (1945)

In my first years at university, we used the Danish Gier (1965)

The first computers were sequential

The Turing machine (1936) - a mathematical model



Eniac (1945)

In my first years at university, we used the Danish Gier (1965)

Transistormaskinen GIER, 1961

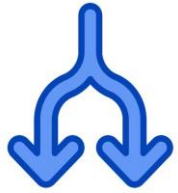
Timesharing - my first programs (1969)

Via a number of terminals several users shared the computer

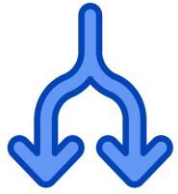
My first (and best) question:

What happens if two users print simultaneously?

Concurrency

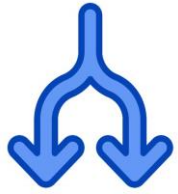


Concurrency



```
public class RemoveDuplicateInArrayExample{
  public static int removeDuplicateElements(int arr[], int n){
    if (n==0 || n==1){  return n; }
    int[] temp = new int[n];
    int j = 0;
    for (int i=0; i<n-1; i++){
      if (arr[i] != arr[i+1]){
        temp[j++] = arr[i];
      }
     }
    temp[j++] = arr[n-1];
    // Changing original array
    for (int i=0; i<j; i++){
      arr[i] = temp[i];
    }
    return j;
  }

  public static void main (String[] args) {
    int arr[] = {10,20,20,30,30,40,50,50};
    int length = arr.length;
    length = removeDuplicateElements(arr, length);
    //printing array elements
    for (int i=0; i<length; i++)   System.out.print(arr[i]+" ");
  }
}
```

Concurrency

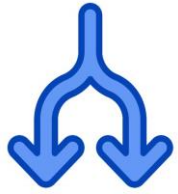


```
public class RemoveDuplicateInArrayExample{
  public static int removeDuplicateElements(int arr[], int n){
    if (n==0 || n==1){  return n; }
    int[] temp = new int[n];
    int j = 0;
    for (int i=0; i<n-1; i++){
      if (arr[i] != arr[i+1]){
        temp[j++] = arr[i];
      }
     }
    temp[j++] = arr[n-1];
    // Changing original array
    for (int i=0; i<j; i++){
      arr[i] = temp[i];
    }
    return j;
  }

  public static void main (String[] args) {
    int arr[] = {10,20,20,30,30,40,50,50};
    int length = arr.length;
    length = removeDuplicateElements(arr, length);
    //printing array elements
    for (int i=0; i<length; i++)   System.out.print(arr[i]+" ");
  }
}
```

Single stream

Concurrency

```
public class RemoveDuplicateInArrayExample{
  public static int removeDuplicateElements(int arr[], int n){
    if (n==0 || n==1){  return n; }
    int[] temp = new int[n];
    int j = 0;
    for (int i=0; i<n-1; i++){
      if (arr[i] != arr[i+1]){
        temp[j++] = arr[i];
      }
     }
    temp[j++] = arr[n-1];
    // Changing original array
    for (int i=0; i<j; i++){
      arr[i] = temp[i];
    }
    return j;
  }

  public static void main (String[] args) {
    int arr[] = {10,20,20,30,30,40,50,50};
    int length = arr.length;
    length = removeDuplicateElements(arr, length);
    //printing array elements
    for (int i=0; i<length; i++)   System.out.print(arr[i]+" ");
  }
}
```



Single stream

Concurrency

```
public class RemoveDuplicateInArrayExample{
  public static int removeDuplicateElements(int arr[], int n){
    if (n==0 || n==1){  return n; }
    int[] temp = new int[n];
    int j = 0;
    for (int i=0; i<n-1; i++){
      if (arr[i] != arr[i+1]){
        temp[j++] = arr[i];
      }
     }
    temp[j++] = arr[n-1];
    // Changing original array
    for (int i=0; i<j; i++){
      arr[i] = temp[i];
    }
    return j;
  }

  public static void main (String[] args) {
    int arr[] = {10,20,20,30,30,40,50,50};
    int length = arr.length;
    length = removeDuplicateElements(arr, length);
    //printing array elements
    for (int i=0; i<length; i++)   System.out.print(arr[i]+" ");
  }
}
```

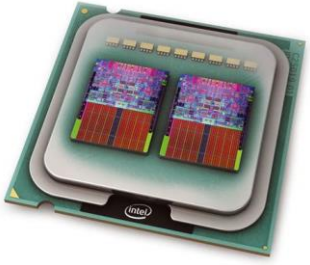
Single stream

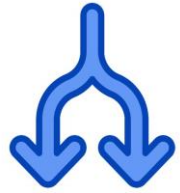
```
------------
----
--------
-----
---
-------
```

Concurrent (multiple streams)

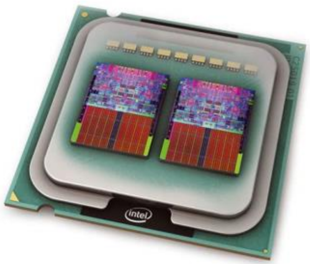
```
----------    ----    ------------
----          ---------    ----
--------      -----    --------
-----         --------    -----
---           ---------    ---
-------       --------    -------
              ---------
              -----
```

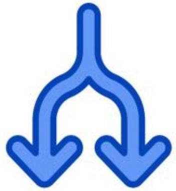
Multiple streams

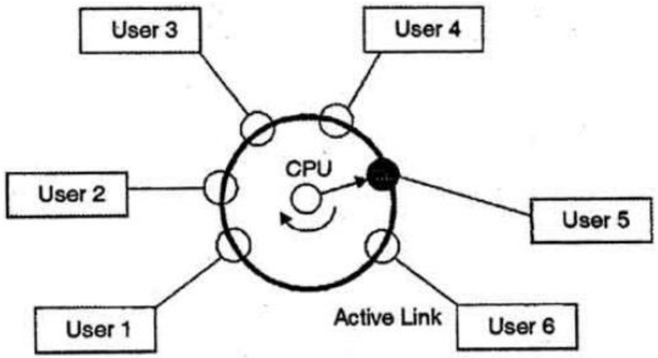


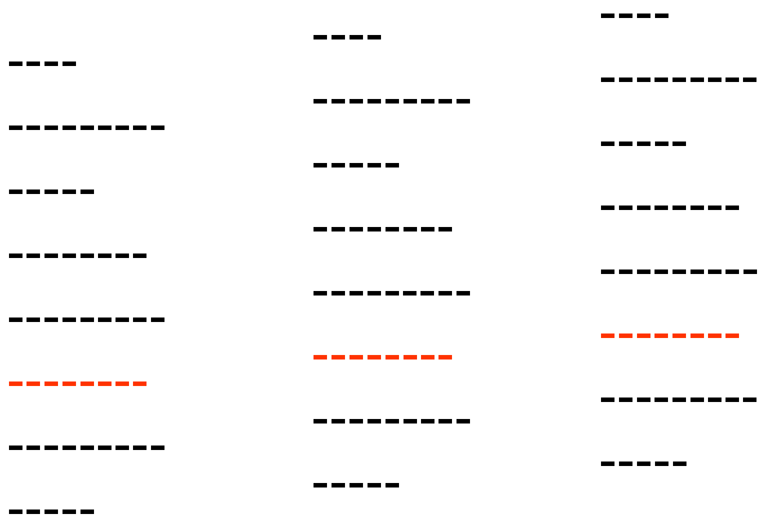


Multiple streams



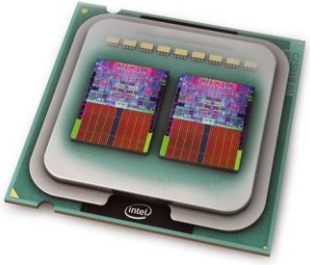


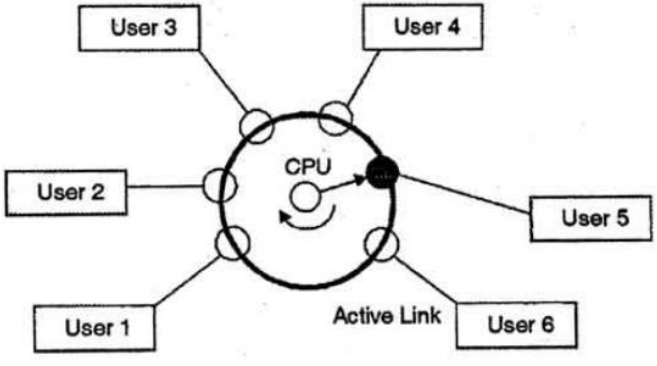




Multiple streams

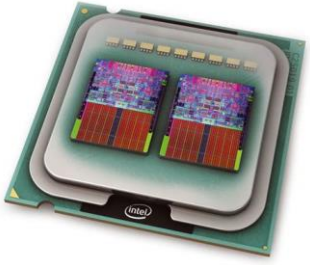
Exploitation



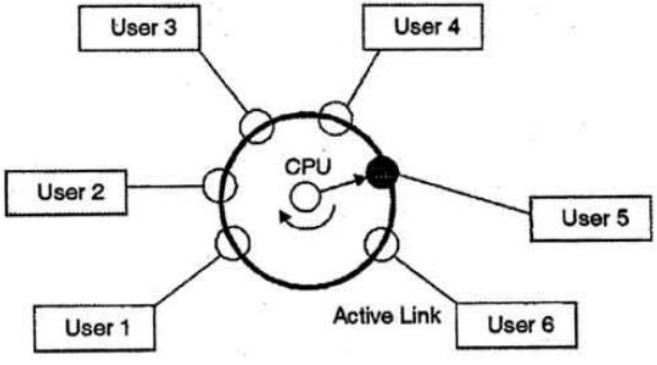


Multiple streams

Exploitation

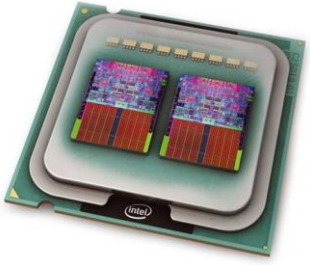


Inherent

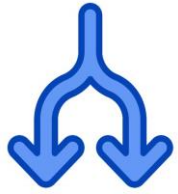


Multiple streams

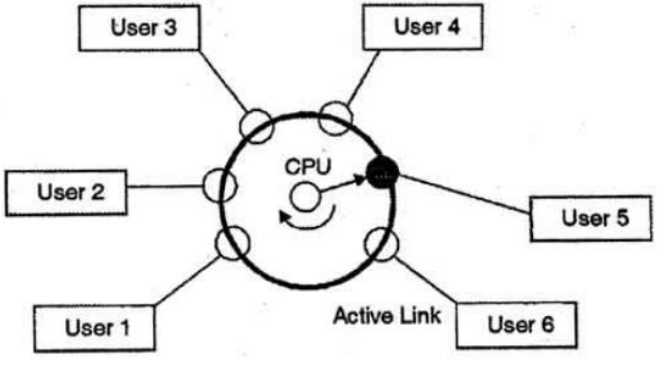
Exploitation



Inherent

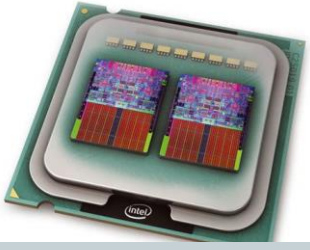


Hidden

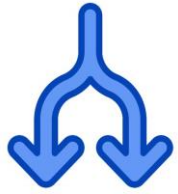


Multiple streams

Exploitation



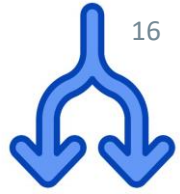
Inherent



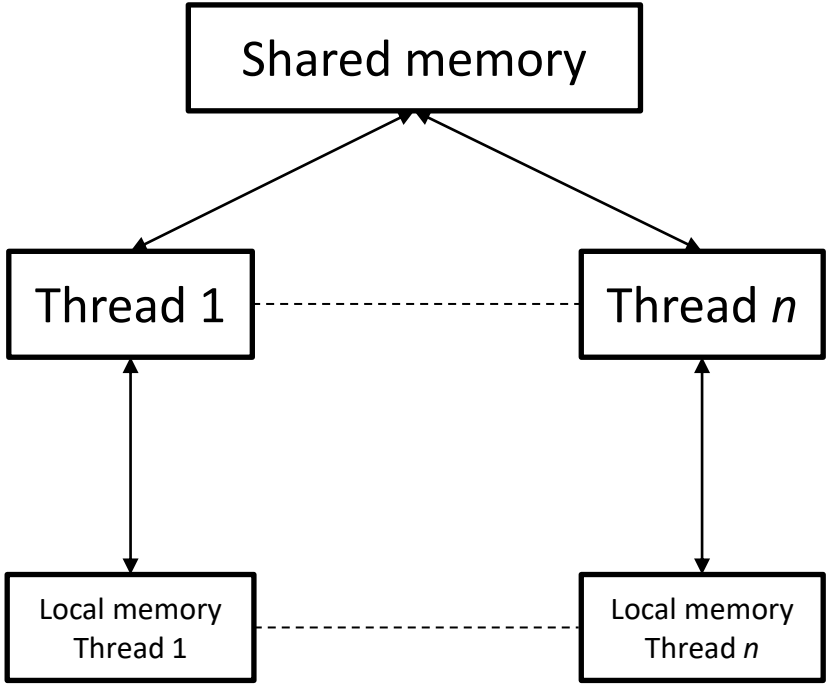
Concurrency is an abstraction for all of these

(and more)

Threads in Java



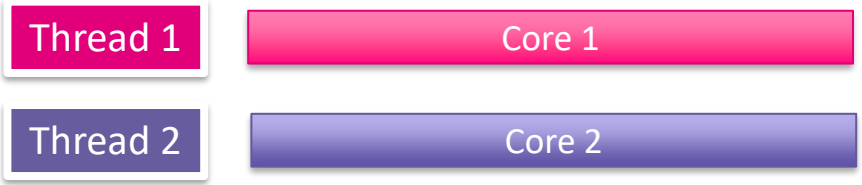
- A *thread* is a stream of program statements executed sequentially
- Several threads can be executed at the same time, i.e., *concurrently*
- Each threads works at its own speed
- Each thread has its own local memory
- Threads can communicate via shared memory



Sequential vs Parallel vs Concurrent

- **Sequential** execution (in 1 processing core)

- **Parallel** execution (in 2 processing cores)

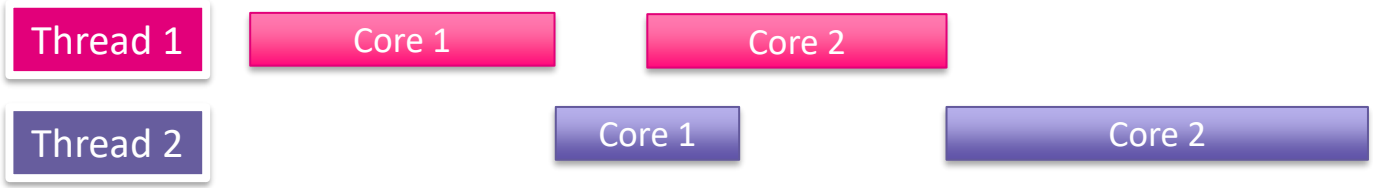


Sequential vs Parallel vs Concurrent

- **Sequential** execution (in 1 processing core)

- **Parallel** execution (in 2 processing cores)

- **Concurrent** execution (in 2 processing cores)



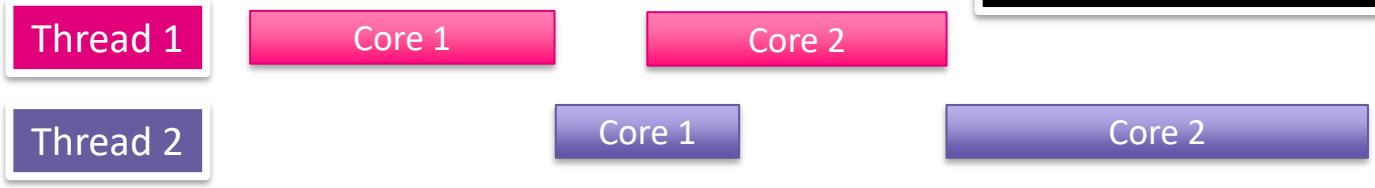
Sequential vs Parallel vs Concurrent

- **Sequential** execution (in 1 processing core)

- **Parallel** execution (in 2 processing cores)

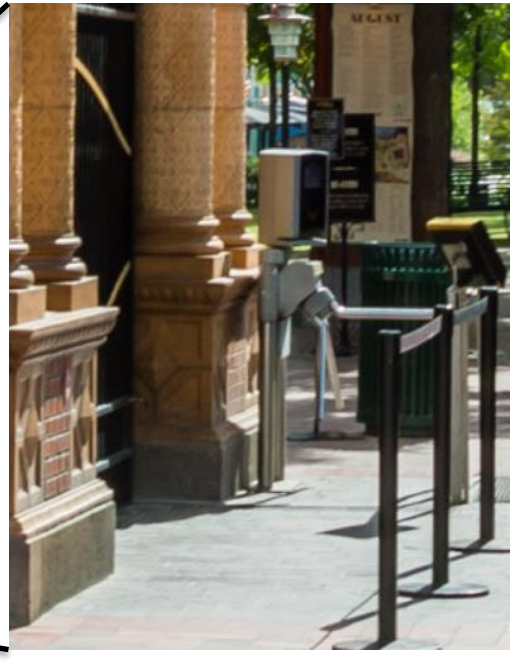
- Computation is split in arbitrary blocks (as opposed to sequential)
- Concurrent execution can happen with one core (as opposed to parallel)
- When there are more than two cores different threads may run in parallel

- **Concurrent** execution (in 2 processing cores)



Threads in Java – Example

Tivoli entrance turnstile



Threads in Java - Example

- Java threads can be created either by implementing **`Runnable`** or extending from **`Thread`**
- The behaviour of the thread is in the **`run()`** method (override)

```
final long PEOPLE = 10_000;
long counter = 0;
…

public class Turnstile extends Thread {
   public void run() {
        for (int i = 0; i < PEOPLE; i++) {
            counter++;
        }
   }
}
```

This thread simulates 10000 people entering to Tivoli

Threads in Java - Example

- Java threads can be created either by implementing **`Runnable`** or extending from **`Thread`**
- The behaviour of the thread is in the **`run()`** method (override)

```
final long PEOPLE = 10_000;
long counter = 0;
…

public class Turnstile extends Thread {
   public void run() {
        for (int i = 0; i < PEOPLE; i++) {
            counter++;
        }
   }
}
```

Shared memory

This thread simulates 10000 people entering to Tivoli

Threads in Java - Example

- Java threads can be created either by implementing **`Runnable`** or extending from **`Thread`**
- The behaviour of the thread is in the **`run()`** method (override)

```
final long PEOPLE = 10_000;
long counter = 0;
...

public class Turnsti                Thread {
   public void run()
        for (int i = 0; i < PEOPLE; i++) {
            counter++;
        }
   }
}
```

This thread simulates 10000 people entering to Tivoli

Shared memory

Local memory

Threads in Java - Example

- Java threads can be created either by implementing **`Runnable`** or extending from **`Thread`**
- The behaviour of the thread is in the **`run()`** method (override)

```
final long PEOPLE = 10_000;
long counter = 0;
...

public class Turnsti          Thread {
   public void run()
        for (int i = 0; i < PEOPLE; i++) {
            counter++;
        }
   }
}
```

This thread simulates 10000 people entering to Tivoli

Shared memory

Local memory

Behaviour of the thread

Threads in Java - Example

- The function **`start()`** starts the execution of the thread
- The function **`join()`** waits for the thread to terminate

```
Turnstile turnstile = new Turnstile();

turnstile.start();

turnstile.join();

System.out.println(counter+" people entered");
```

Create an instance of the thread

Start execution of the thread

Wait until the thread terminates

Print the value of **`counter`**

Threads in Java - Example

- Altogether (not executable, see **`CounterThreads.java`** for the executable program)

```
class CounterThreads {
    long counter = 0;
    final long PEOPLE  = 10_000;


    // main thread behaviour
    Turnstile turnstile = new Turnstile();
    turnstile.start();
    turnstile.join();
    System.out.println(counter+" people entered");


    // inner class for accessing shared variables
    public class Turnstile extends Thread {
       public void run() {
           for (int i = 0; i < PEOPLE; i++) {
                counter++;
           }
       }
    }
}
```

Threads in Java - Example

- Altogether (not executable, see **`CounterThreads.java`** for the executable program)

```
class CounterThreads {
    long counter = 0;
    final long PEOPLE  = 10_000;


    // main thread behaviour
    Turnstile turnstile = new Turnstile();
    turnstile.start();
    turnstile.join();
    System.out.println(counter+" people entered");


    // inner class for accessing shared variables
    public class Turnstile extends Thread {
       public void run() {
           for (int i = 0; i < PEOPLE; i++) {
                counter++;
           }
       }
    }
}
```

Shared memory

Create, start, wait till termination and print results

Definition of the thread´s behaviour

Threads in Java - Example

What value of **`counter`** will this program print?

ecutable, see **`CounterThreads.java`** for
gram)

Shared memory

Create, start, wait till termination and print results

Definition of the thread´s behaviour

```
class CounterThreads {
    long counter = 0;
    final long PEOPLE  = 10_000;


    // main thread behaviour
    Turnstile turnstile = new Turnstile();
    turnstile.start();
    turnstile.join();
    System.out.println(counter+" people entered");


    // inner class for accessing shared variables
    public class Turnstile extends Thread {
        public void run() {
            for (int i = 0; i < PEOPLE; i++) {
                counter++;
            }
        }
    }
}
```

Threads in Java – Alternative Definition

Other ways to define threads

Runnable object in the thread constructor

```
long counter = 0;
final long PEOPLE  = 10_000;


Thread t = new Thread(new Runnable(){
    public void run() {
           for (int i=0; i<PEOPLE; i++){
                counter++;
           }
    }
});
t.start();
t.join();
System.out.println(counter+" people entered");
```

Using Java lambda expressions

```
long counter = 0;
final long PEOPLE  = 10_000;


Thread t = new Thread(() -> {
    for (int i=0; i<PEOPLE; i++){
        counter++;
    }
});
t.start();
t.join();
System.out.println(counter+" people entered");
```

Threads in Java – Alternative Definition

Other ways to define threads

Runnable object in the thread constructor

```
long counter = 0;
final long PEOPLE  = 10_000;


Thread t = new Thread(new Runnable(){
    public void run() {
           for (int i=0; i<PEOPLE; i++){
                counter++;
           }
    }
});
t.start();
t.join();
System.out.println(counter+" people entered");
```

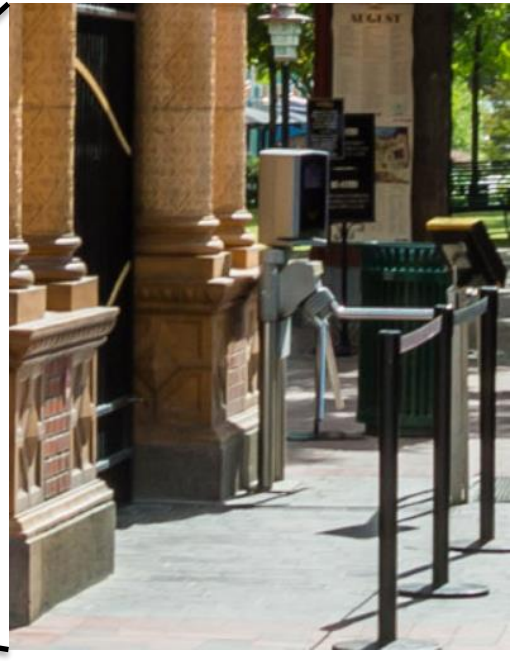
Using Java lambda expressions

```
long counter = 0;
final long PEOPLE  = 10_000;


Thread t = new Thread(() -> {
    for (int i=0; i<PEOPLE; i++){
        counter++;
    }
});
t.start();
t.join();
System.out.println(counter+" people entered");
```

I would only recommend these when the thread´s code is small, e.g., without several methods. And when the local state of the thread is minimal. WARNING: Possibly bias opinion!

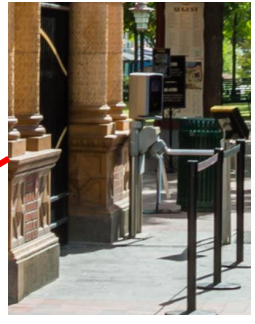
Threads in Java – Example

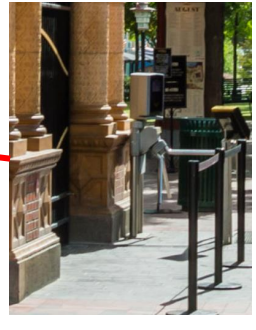
Tivoli entrance turnstile



Threads in Java – Example II

Tivoli entrance turnstile





Threads in Java – Example II

- Altogether (not executable, see **`CounterThreads2.java`** for the executable program)

```
long counter = 0;
final long PEOPLE  = 10_000;


Turnstile turnstile1 = new Turnstile();
Turnstile turnstile2 = new Turnstile();
turnstile1.start();turnstile2.start();
turnstile2.join();turnstile2.join();
System.out.println(counter+" people entered");



public class Turnstile extends Thread {
    public void run() {
        for (int i = 0; i < PEOPLE; i++) {
          counter++;
        }
    }
}
```

We simply add another Turnstile thread

Threads in Java – Example II

What value of **counter** will this program print?

ecutable, see **CounterThreads2.java** for
gram)

```
long counter = 0;
final long PEOPLE  = 10_000;


Turnstile turnstile1 = new Turnstile();
Turnstile turnstile2 = new Turnstile();
turnstile1.start();turnstile2.start();
turnstile2.join();turnstile2.join();
System.out.println(counter+" people entered");



public class Turnstile extends Thread {
    public void run() {
        for (int i = 0; i < PEOPLE; i++) {
          counter++;
        }
    }
}
```

We simply add another Turnstile thread

Threads in Java – Example II

What value of **counter** will this program print?

ecutable, see **CounterThreads2.java** for
gram)

```
long counter = 0;
final long PEOPLE  = 10_000;


Turnstile turnstile1 = new Turnstile();
Turnstile turnstile2 = new Turnstile();
turnstile1.start();turnstile2.start();
turnstile2.join();turnstile2.join();
System.out.println(counter+" people entered");



public class Turnstile extends Thread {
    public void run() {
        for (int i = 0; i < PEOPLE; i++) {
          counter++;
        }
    }
}
```

We simply add another Turnstile thread

Threads in Java – Example II

What value of **`counter`** will this program print?

ecutable, see **`CounterThreads2.java`** for
gram)

```
long counter = 0;
final long PEOPLE  = 10_000;


Turnstile turnstile1 = new Turnstile();
Turnstile turnstile2 = new Turnstile();
turnstile1.start();turnstile2.start();
turnstile2.join();turnstile2.join();
System.out.println(counter+" people entered");



public class Turnstile extends Thread {
    public void run() {
                                          i++) {
}
```

We simply add another Turnstile thread

VERY HARD: What is the minimum value of **`counter`** that this program can print?

Threads in Java – Example II

- What was is the problem in the previous program?

- To answer this question we need to understand
  - Atomicity
  - States of a thread
  - Non-determinism
  - Interleavings

Atomicity

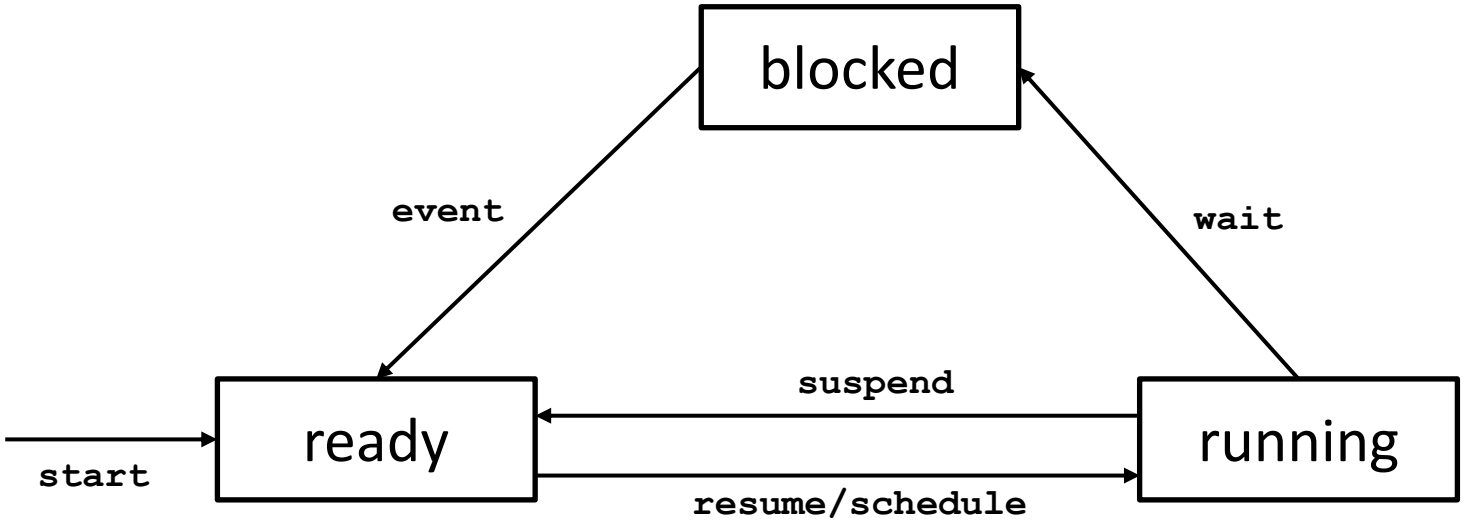
- The program statement **`counter++`** is not *atomic*
- *Atomic statements are executed as a single (indivisible) operation*

```
public class Turnstile extends Thread {
   public void run() {
        for (int i = 0; i < PEOPLE; i++) {
            counter++;
        }
   }
}
```

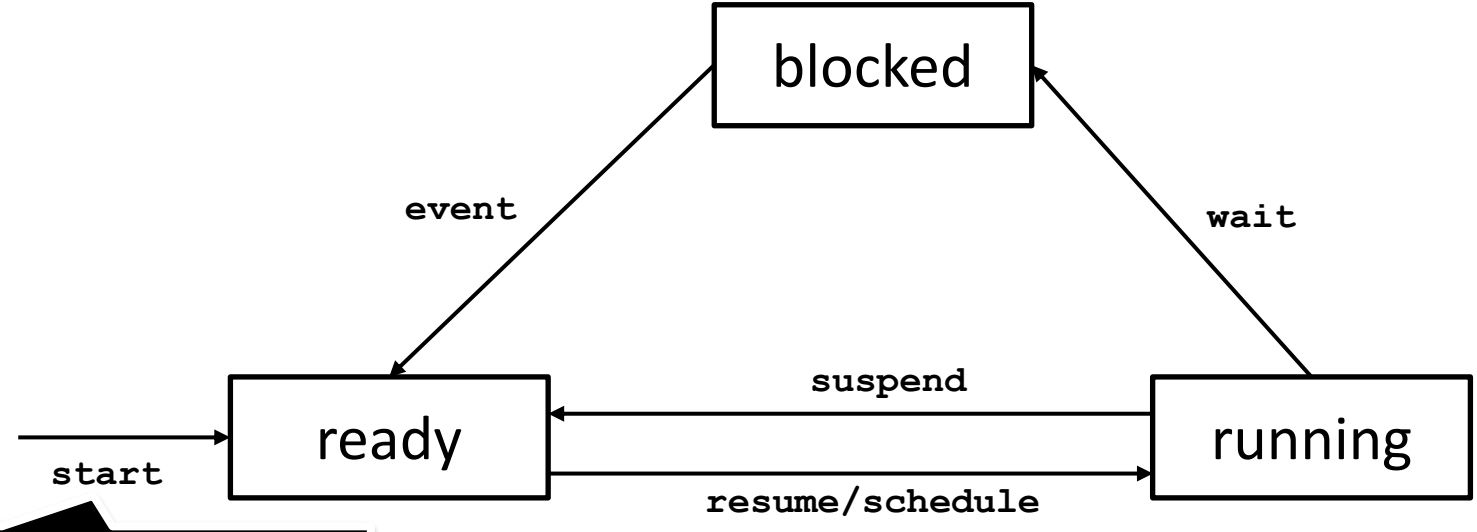
```
int temp = counter;
counter = temp + 1;
```

Watchout: Just because a program statement is a one-liner, it doesn´t mean that it is atomic

States of a thread (simplified)

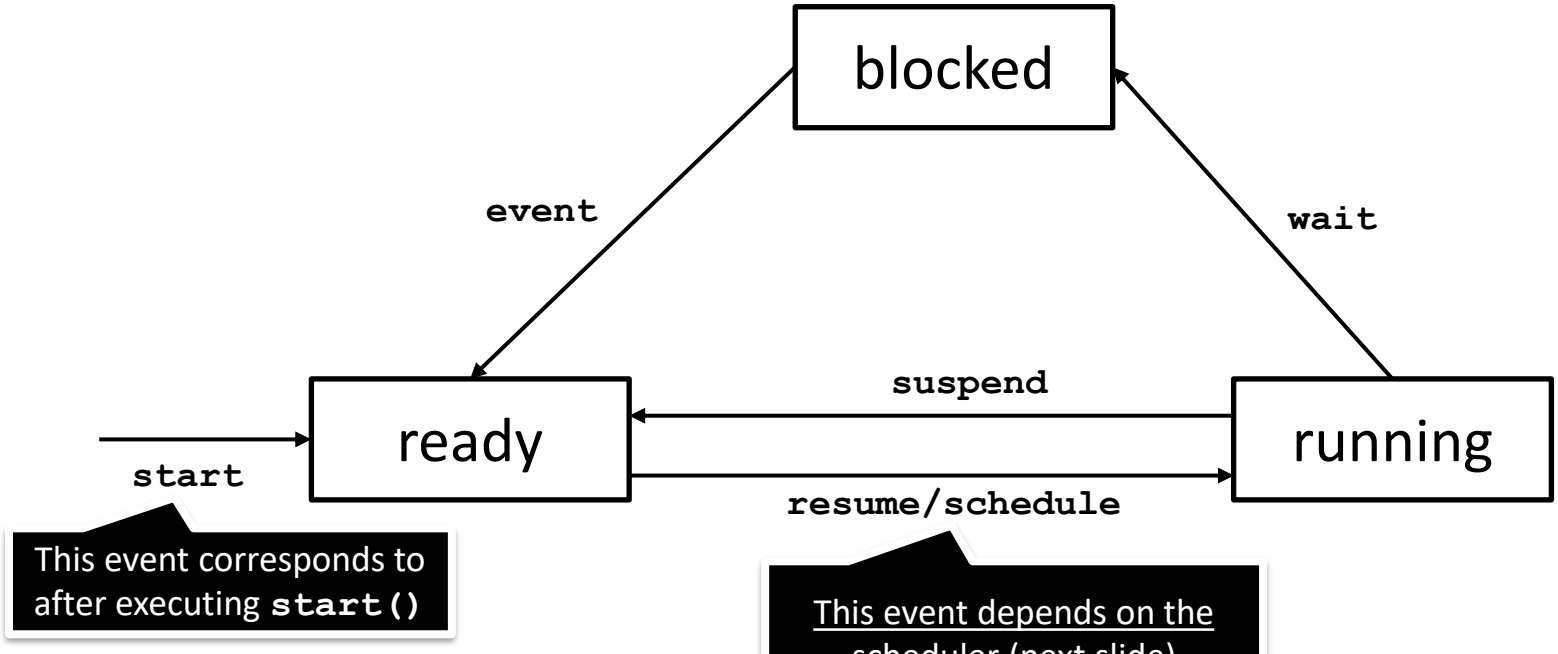


States of a thread (simplified)

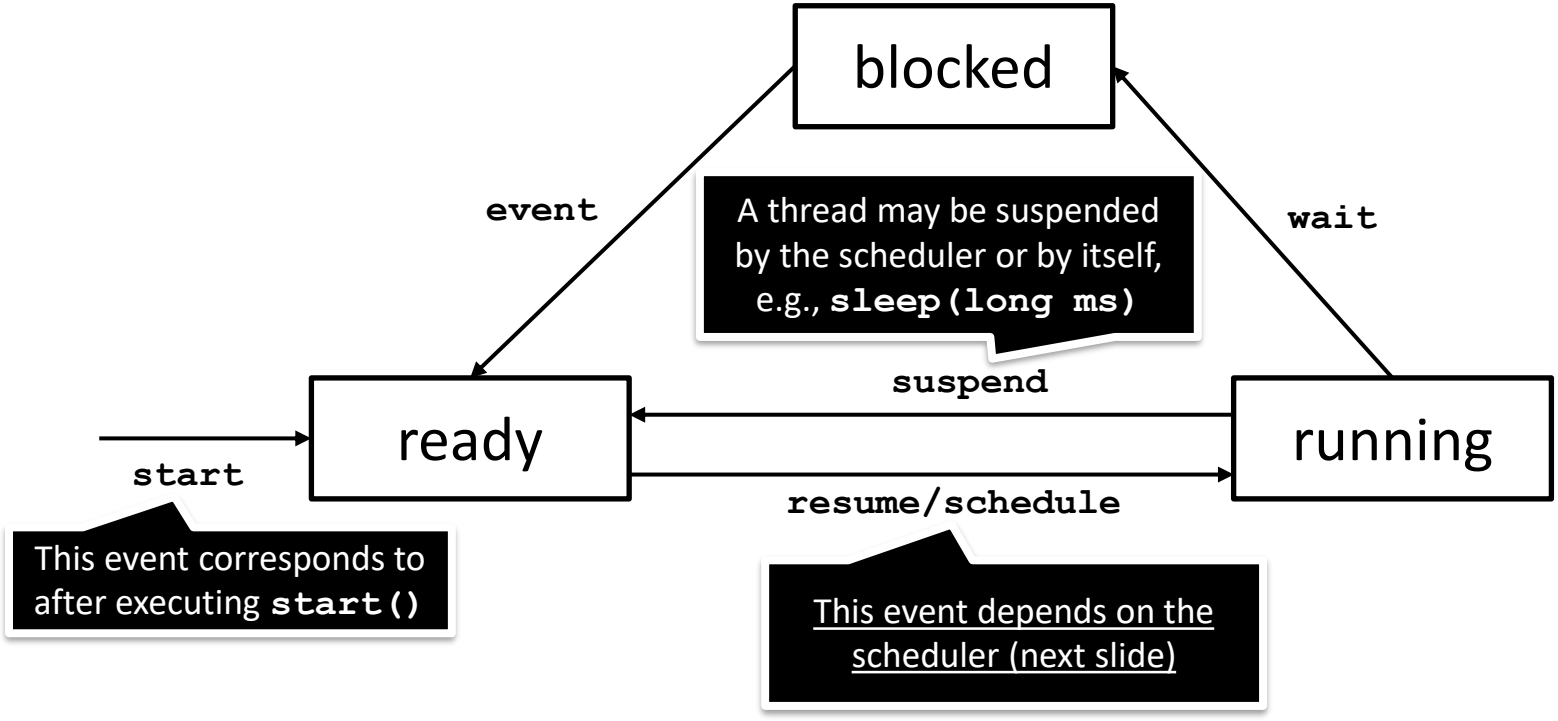


This event corresponds to after executing `start()`

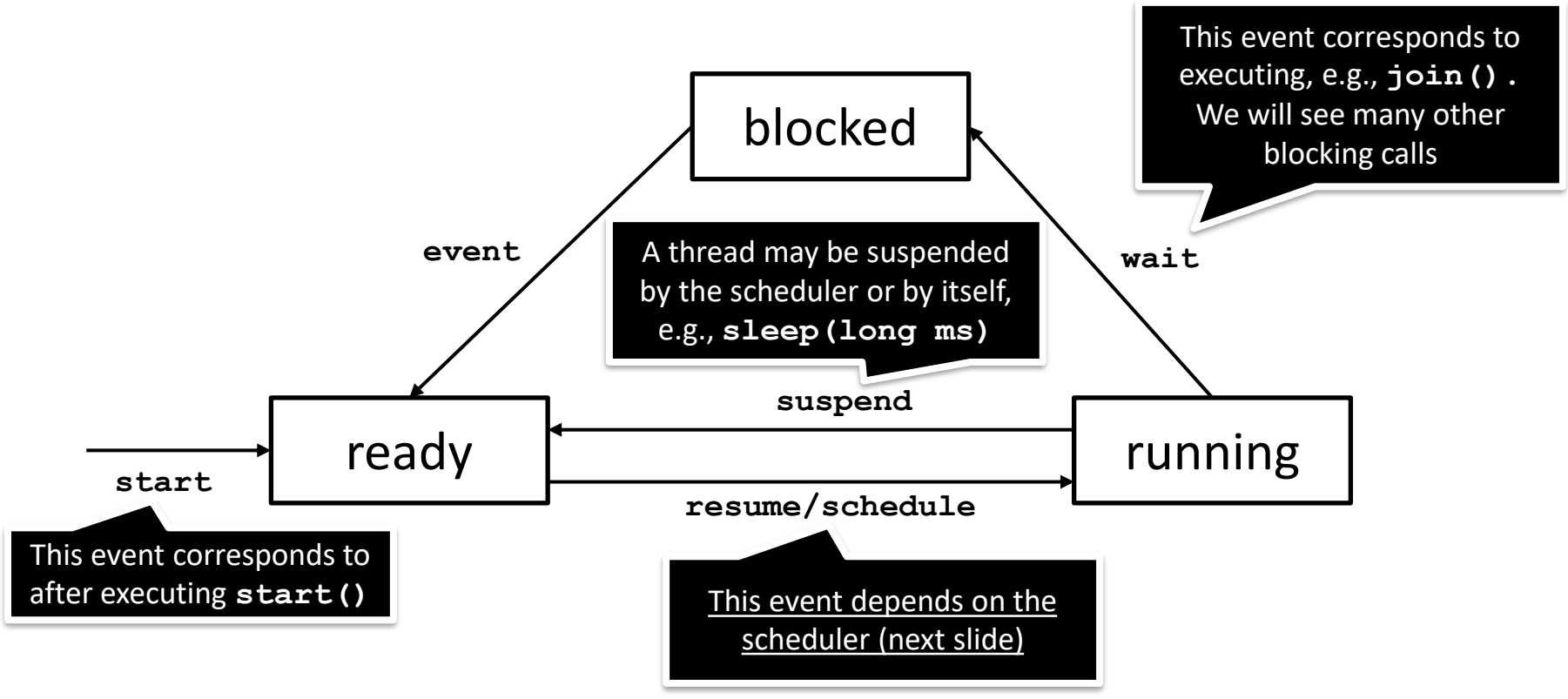
States of a thread (simplified)



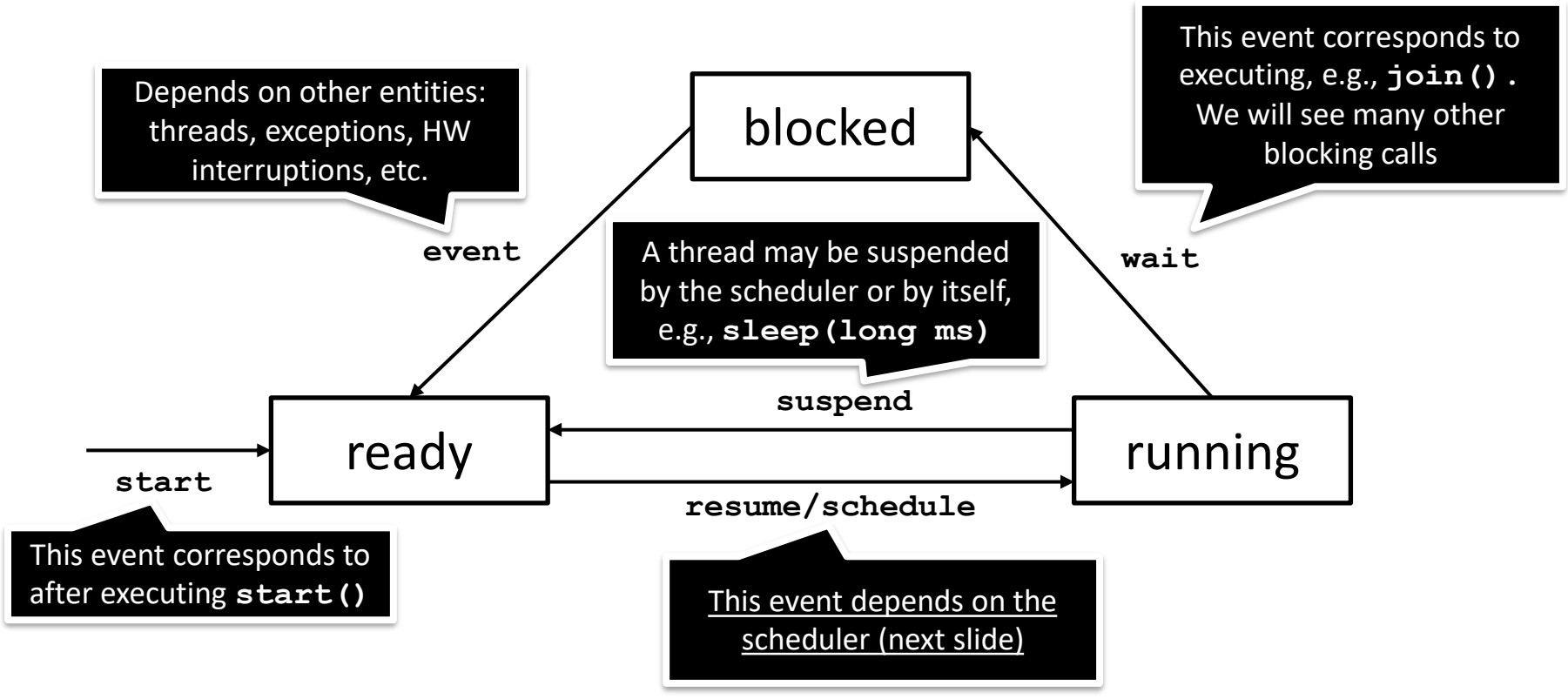
States of a thread (simplified)



States of a thread (simplified)



States of a thread (simplified)



Non-determinism | Scheduler

- In all operating systems/executing environments a *scheduler* selects the processes/threads under execution
  - Threads are selected *non-deterministically*, i.e., no assumptions can be made about what thread will be executed next

- Consider two threads t1 and t2 in the ready state; *t1(ready)* and *t2(ready)*
  1. *t1(running) -> t1(ready) -> t1(running) -> t1(ready) -> …*
  2. *t2(running) -> t2(ready) -> t2(running) -> t2(ready) -> …*
  3. *t1(running) -> t1(ready) -> t2(running) -> t2(ready) -> …*
  4. Infinitely many different executions!

Interleaving

- The statements in a thread are executed when the thread is in its “running” state

- An *interleaving* is a possible sequence of operations for a concurrent program
  - Note this: a sequence of operations for a concurrent program, not for a thread. Concurrent programs are composed by 2 or more threads.

Interleaving – Example I

main

```
long counter = 0;
final long PEOPLE  = 10_000;

Turnstile turnstile1 = new Turnstile();
Turnstile turnstile2 = new Turnstile();
turnstile1.start();turnstile1.start();
```

time

turnstile1

```
// first iteration loop
int i = 0
if (i < PEOPLE)
int temp = counter // counter == 0
counter = temp + 1 // counter == 1
```

time

turnstile2

```
// first iteration loop
int i = 0
if (i < PEOPLE)
int temp = counter // counter == 1
counter = temp + 1 // counter == 2
…
```

time

Interleaving – Example II

main

```
long counter = 0;
final long PEOPLE  = 10_000;

Turnstile turnstile1 = new Turnstile();
Turnstile turnstile2 = new Turnstile();
turnstile1.start();turnstile1.start();
```

time

turnstile1

```
// first iteration loop
int i = 0
if (i < PEOPLE)
int temp = counter // counter == 0
```

```
counter = temp + 1 // counter == 1
```

time

turnstile2

```
// first iteration loop
int i = 0
if (i < PEOPLE)
int temp = counter // counter==0
```

```
counter = temp + 1 // counter == 1
i++
…
```

time

Interleaving – Example II

main

```
long counter = 0;
final long PEOPLE  = 10_000;

Turnstile turnstile1 = new Turnstile();
Turnstile turnstile2 = new Turnstile();
turnstile1.start();turnstile1.start();
```

Are there any other interleavings?

time

turnstile1

```
// first iteration loop
int i = 0
if (i < PEOPLE)
int temp = counter // counter == 0
```

```
counter = temp + 1 // counter == 1
```

time

turnstile2

```
// first iteration loop
int i = 0
if (i < PEOPLE)
int temp = counter // counter==0
```

```
counter = temp + 1 // counter == 1
i++
…
```

time

Interleaving – Example II

main

```
long counter = 0;
final long PEOPLE  = 10_000;

Turnstile turnstile1 = new Turnstile();
Turnstile turnstile2 = new Turnstile();
turnstile1.start();turnstile1.start();
```

Are there any other interleavings?

There are as many interleavings as possible ways to order the operations in the program

turnstile1

```
// first iteration loop
int i = 0
if (i < PEOPLE)
int temp = counter // counter == 0
```

```
counter = temp + 1 // counter == 1
```

time

turnstile2

```
// first iteration loop
int i = 0
if (i < PEOPLE)
int temp = counter // counter==0
```

```
counter = temp + 1 // counter == 1
i++
…
```

time

Race Conditions

Concurrency Humour…

- Knock, knock

- Race condition

- Who's there?

Race Conditions

- *A **race condition** occurs when the result of the computation depends on the interleavings of the operations*

Data Races

- *A **data race** occurs when two concurrent threads:*
  - *Access a shared memory location*
  - *At least one access is a write*

Race Conditions vs Data Races

Not all race conditions are data races

- Threads may not access shared memory
- Threads may not write on shared memory

Not all data races result in race conditions

- The result of the program may not change based on the writes of threads

Threads in Java – Example II

- What was is the problem in the previous program?
  - The statement **`counter++`** is not atomic
  - Some interleavings result in threads reading stale (outdated) data
  - Consequently, the program has race conditions
- In what follows, we will see how to tackle this type of problems

Critical sections

- A *critical section* is a part of the program that only one thread can execute at the same time
  - Useful to avoid race conditions in concurrent programs

```
public class Turnstile extends Thread {
   public void run() {
        for (int i = 0; i < PEOPLE; i++) {
            // start critical section
            int temp = counter;
            counter = temp + 1;
            // end critical section
        }
   }
}
```

Critical sections

- A *critical section* is a part of the program that only one thread can execute at the same time
  - Useful to avoid race conditions in concurrent programs

```
public class Turnstile extends Thread {
   public void run() {
        for (int i = 0; i < PEOPLE; i++) {
            // start critical section
            int temp = counter;
            counter = temp + 1;
            // end critical section
        }
   }
}
```

Critical sections should cover the parts of the code handling shared memory

Critical sections

- A *critical section* is a part of the program that only one thread can execute at the same time
- ... race conditions in concurrent programs

Shouldn't the critical section start before the **`for`**?

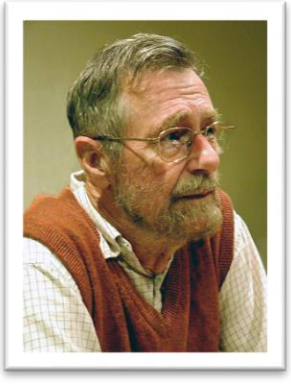
```
...stile extends Thread {
    public void run() {
        for (int i = 0; i < PEOPLE; i++) {
            // start critical section
            int temp = counter;
            counter = temp + 1;
            // end critical section
        }
    }
}
```

Critical sections should cover the parts of the code handling shared memory

Mutual Exclusion

- The *mutual exclusion* property states that
  - *Two or more threads cannot be executing their critical section at the same time*

- Mutual exclusion was first formulated by EW Dijkstra (see optional readings)
  - He devised a protocol to ensure mutual exclusion (*solving the mutual exclusion problem*)
  - He laid down the properties for a satisfactory solution to ensuring mutual exclusion



Mutual Exclusion

- An ideal solution to the mutual exclusion problem must ensure:
  - Mutual exclusion: at most one thread executing the critical section at the same time
  - Absence of *deadlock*: threads eventually exit the critical section allowing other threads to enter
  - Absence of *starvation*: if a thread is ready to enter the critical section, it must eventually do so

Mutual Exclusion

- An ideal solution to the mutual exclusion problem must ensure:
  - Mutual exclusion: at most one thread executing the critical section at the same time
  - Absence of *deadlock*: threads eventually exit the critical section allowing other threads to enter
  - Absence of *starvation*: if a thread is ready to enter the critical section, it must eventually do so

In practice, we will see that it is not always possible to achieve absence of starvation

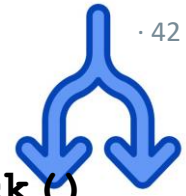
Mutual Exclusion | Java `Locks`

- In Java, mutual exclusion can be achieved using the **`Lock`** interface in the **`java.util.concurrent.locks`** package

  - **`lock()`**
    - Acquires the lock if available, otherwise it blocks
    - It is blocking

  - **`unlock()`**
    - Releases the lock, if there are other threads waiting for the lock it signals one of them
    - It is not blocking

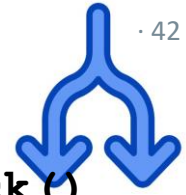
Java **Locks** | Critical sections



- Simple protocol: call **`lock()`** before entering the critical section, and **`unlock()`** after exiting
- *Each critical section must have a lock associated to it, but many critical sections may use the same lock.*
- Simplified, see **`CounterThreadsLock.java`**

```
Lock l = new Lock();

public class Turnstile extends Thread {
   public void run() {
        for (int i = 0; i < PEOPLE; i++) {
            l.lock()          // start critical section
            int temp = counter;
            counter = temp + 1;
            l.unlock()        // end critical section
        }
   }
}
```

Java **Locks** | Critical sections



- Simple protocol: call **`lock()`** before entering the critical section, and **`unlock()`** after exiting
- *Each critical section must have a lock associated to it, but critical sections may use the same lock.*
- Simplified, see **`CounterThreadsLock.java`**

```
Lock l = new Lock();

public class Turnstile extends Threa
   public void run() {
        for (int i = 0; i < PEOPLE; i
            l.lock()          // start
            int temp = counter;
            counter = temp + 1;
            l.unlock()        // end cr
        }
   }
}
```

- We do not focus on the implementation of **`lock()`**/**`unlock()`**, but in their use to solve concurrency problems.
- See Sections 2.3, 2.5, 2.7 and Section 7.3 onwards in Herlihy for implementation details of **`lock()`**/**`unlock()`**

Java `Locks` | Interleavings

main

```
long counter = 0;
final long PEOPLE  = 10_000;
Lock l = new Lock()

Turnstile turnstile1 = new Turnstile();
Turnstile turnstile2 = new Turnstile();
turnstile1.start();turnstile1.start();
```

turnstile1

```
// first iteration loop
int i = 0
```

```
if (i < PEOPLE)
```

```
l.lock() // begin critical section
int temp = counter // counter == 0
counter = temp + 1 // counter == 1
l.unlock() // end critical section
```

turnstile2

```
// first iteration loop
int i = 0
```

```
if (i < PEOPLE)
```

```
l.lock() // begin critical section
int temp = counter // counter == 1
counter = temp + 1 // counter == 2
l.unlock() // end critical section
```

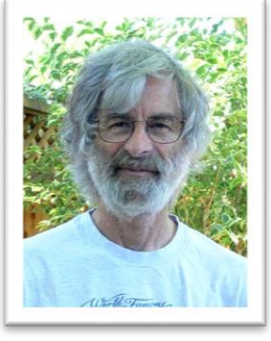
Operations in the critical section are always executed sequentially by the same thread

Happens-before

- In fact, we can now characterize an order of execution between some of the operations of a program
- We say that an operation a *happens-before* than operation b, denoted as $a \rightarrow b$, iff
  - a and b belong to the same thread and a appears before b in the thread definition
  - a is an `unlock()` and b is a `lock()` on the same lock
- In the absence of *happens-before* relation between operations, the JVM is free to choose any execution order
  - In that case we say that operations are executed *concurrently*
  - Sometimes denoted as $a \mid\mid b$

Happens-before

- In fact, we can now characterize an order of execution between some of the operations of a program
- We say that an operation a *happens-before* than operation b, denoted as $a \rightarrow b$, iff
  - a and b belong to the same thread and a appears before b in the thread definition
  - a is an `unlock()` and b is a `lock()` on the same lock
- In the absence of *happens-before* relation between operations, the JVM is free to choose any execution order
  - In that case we say that operations are executed *concurrently*
  - Sometimes denoted as $a \mid\mid b$
- *Happens-before* is a *pre-order* over operations of concurrent programs
  - Transitive
  - Anti-symmetric
- "Happened-before" was first introduced by Leslie Lamport for distributed systems
  - See optional readings



Happens-before

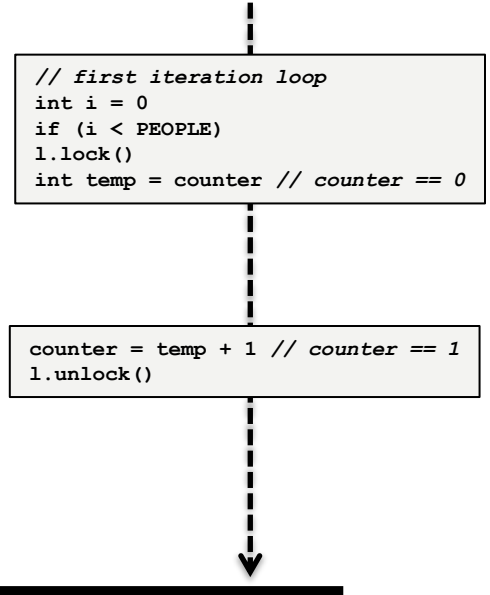
- In fact, we can now characterize an order of execution between some of the operations of a program
- We say that an operation a *happens-before* than operation b, denoted as $a \rightarrow b$, iff
  - a and b belong to the same thread and a appears before b in the thread definition
  - a is an `unlock()` and b is a `lock()` on the same lock
- In the absence of *happens-before* relation between operations, the JVM is free to choose any execution order
  - In that case we say that operations are executed *concurrently*
  - Sometimes denoted as $a \mid\mid b$
- *Happens-before* is a *pre-order* over operations of concurrent programs
  - Transitive
  - Anti-symmetric
- "Happened-befo
  - See optional r

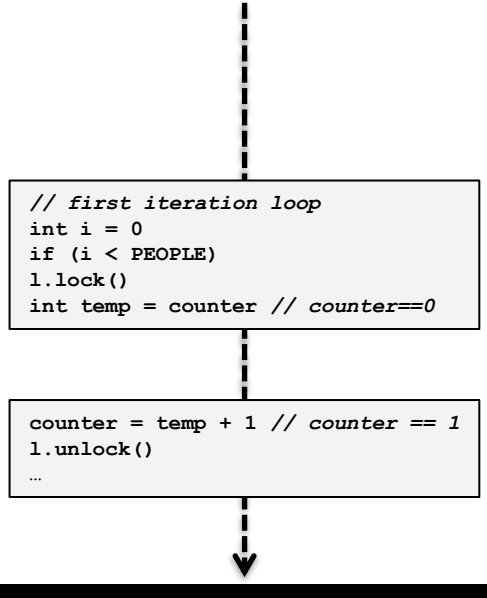
"Don't be brainwashed by programming languages. Free your mind with mathematics." Time for one question before we go straight to the next talk. #HLF18

Happens-before | Interleavings

- *We use locks to remove undesired interlavings*, and happens-before can help us reasoning about correctness

Is this a valid interleaving?

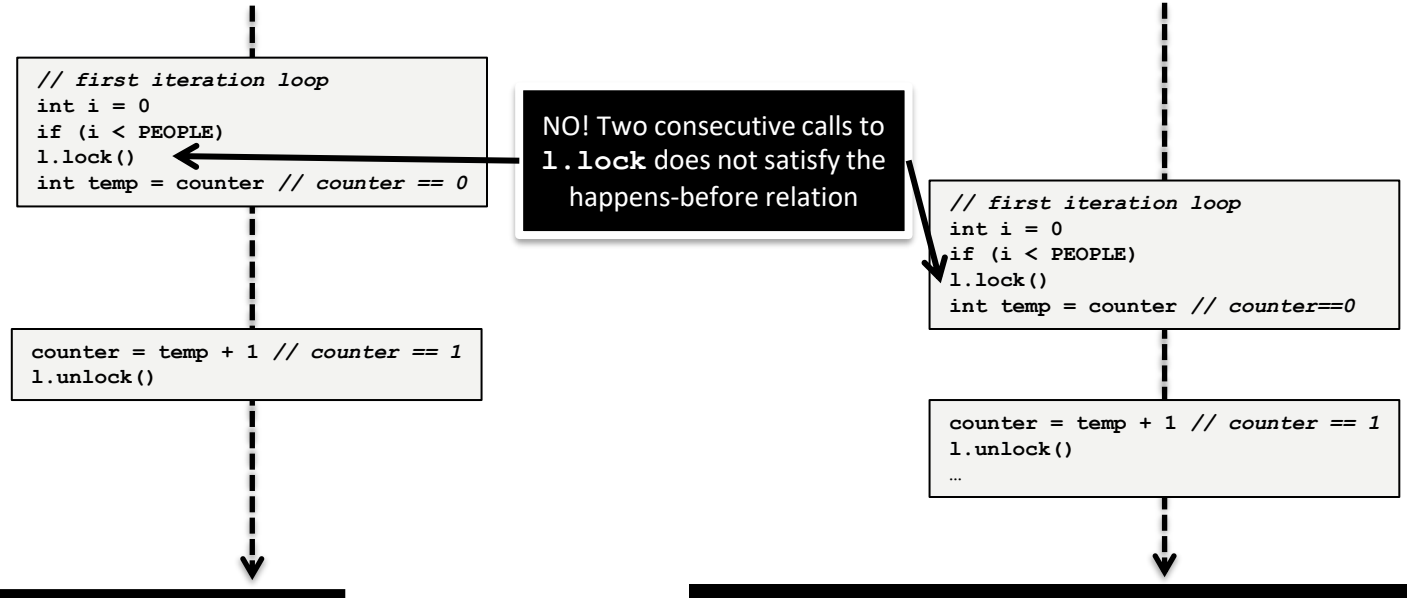




Happens-before | Interleavings

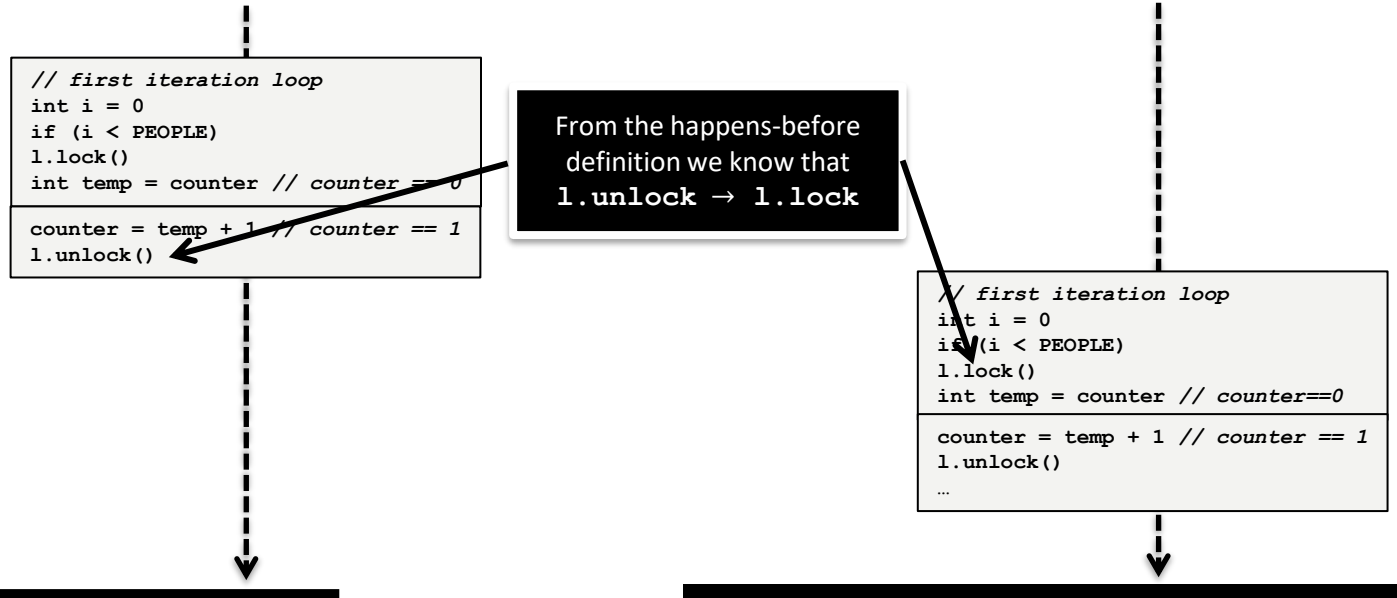
- *We use locks to remove undesired interlavings*, and happens-before can help us reasoning about correctness

Is this a valid interleaving?



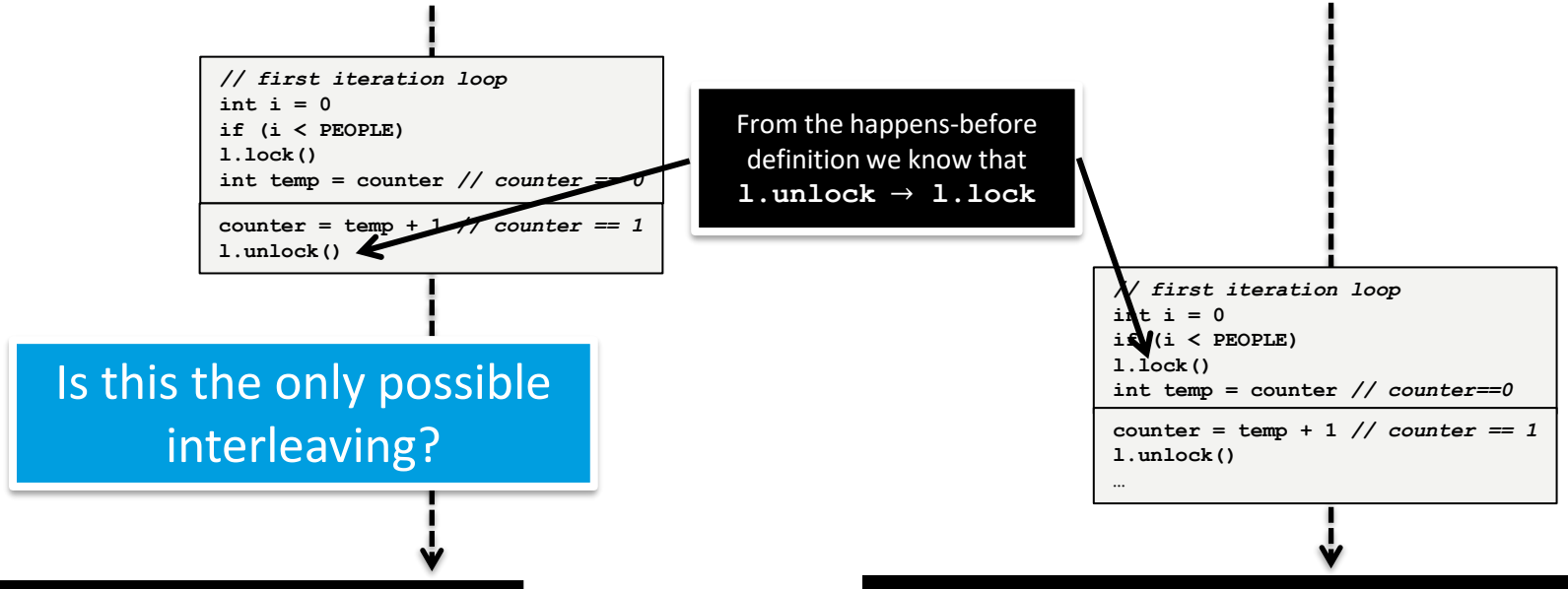
Happens-before | Interleavings

- *We use locks to remove undesired interlavings*, and happens-before can help us reasoning about correctness



Happens-before | Interleavings

- *We use locks to remove undesired interlavings*, and happens-before can help us reasoning about correctness



Deadlocks

- The solution to the turnstile problem ensures mutual exclusion but does it ensure absence of deadlock?

Absence of *deadlock*: threads eventually exit the critical section allowing other threads to enter

```
Lock l = new Lock();

public class Turnstile extends Thread {
   public void run() {
        for (int i = 0; i < PEOPLE; i++) {
            l.lock()          // start critical section
            int temp = counter;
            counter = temp + 1;
            l.unlock()        // end critical section
        }
   }
}
```

(Naïve) examples of deadlocks

- Locking twice within the thread

```
for (int i = 0; i < PEOPLE; i++) {
     l.lock()
     int temp = counter;
     counter = temp + 1;
     l.lock()   // blocks forever
                // or forgetting lot unlock()
}
```

Thread 1

This is not unrealistic.
For instance, see these bug repots in the Linux kernel

[1] https://github.com/torvalds/linux/commit/e1db4ce
[2] https://github.com/torvalds/linux/commit/2904207

- Exception during execution

```
for (int i = 0; i < PEOPLE; i++) {
     l.lock()
     l2.lock()
     int temp = counter;
     throw new Exception();
     // If exception handling doesn't
     unlock, blocks forever
}
```

Thread 1

```
for (int i = 0; i < PEOPLE; i++) {
     l2.lock() // blocks forever
     …
}
```

Thread 2

(Naïve) examples of deadlocks

- Locking twice within the thread

```
for (int i = 0; i < PEOPLE; i++) {
     l.lock()
     int temp = counter;
     counter = temp + 1;
     l.lock()   // blocks forever
                // or forgetting lot unlock()
}
```

Thread 1

This is not unrealistic.
For instance, see these bug repots in the Linux kernel

[1] https://github.com/torvalds/linux/commit/e1db4ce
[2] https://github.com/torvalds/linux/commit/2904207

- Exceptio

Why did I write `1` and `12` in this example?

```
for (int i = 0 ... < PEOPLE; i++) {
     l.lock()
     l2.lock()
     int temp = counter;
     throw new Exception();
     // If exception handling doesn't
     unlock, blocks forever

}
```

Thread 1

```
for (int i = 0; i < PEOPLE; i++) {
     l2.lock() // blocks forever
     …
}
```

Thread 2

Java `Lock` Idiom

- When using Java locks, it is recommended to use the idiom on the right
- This prevents deadlocks problems if the thread finish unexpectedly due to exception
  - Solutions not following this idiom cannot be deemed as free from deadlocks (note for assignments)
- We use this idiom in **`CounterThreadLocks.java`**

```
Lock l = new Lock();

l.lock()
try {
   // critical section code
} finally {
   l.unlock()
}
```

Java ReentratLock

- Java **Lock** is an interface, so it cannot be used as we showed in the examples today
- We use an implementation of the **Lock** interface, namely **ReentrantLock**
  - Reentrant locks act like a regular lock, except that they allow locks to be locked more than once by the same thread

```
Lock l = new ReentrantLock();

for (int i = 0; i < PEOPLE; i++) {
    l.lock()
    int temp = counter;
    counter = temp + 1;
    l.lock();     // it doesn't block
    l.unlock();   // still holds the lock
    l.unlock();   // now the lock is free
}
```