Practical Concurrent and Parallel Programming 5

Thomas Dybdahl Ahle
IT University of Copenhagen

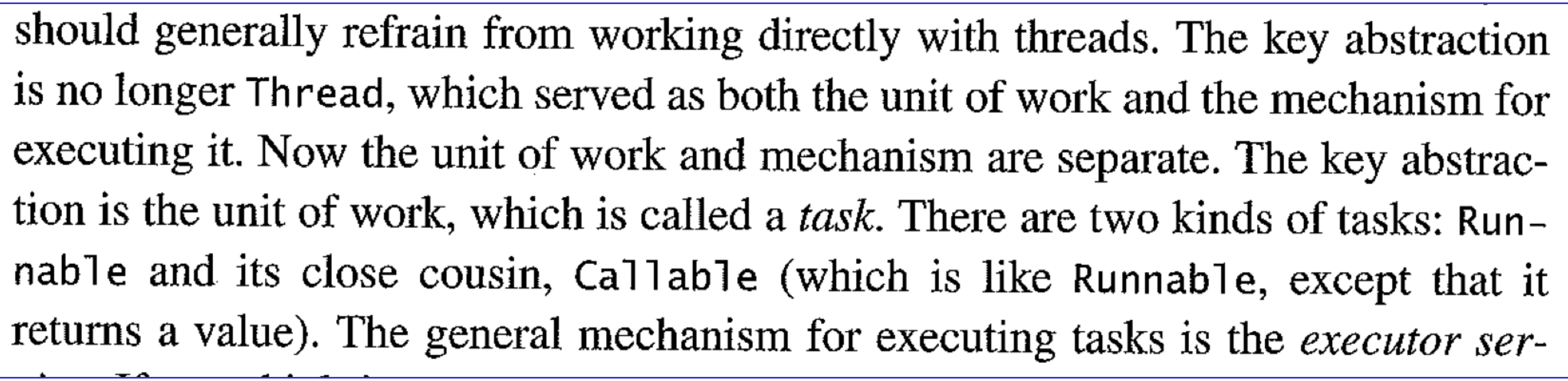
Friday 2019-09-25

Plan for today

- Tasks and the Java executor framework
  - Executors, Runnables, Callables, Futures
- The states of a task
- Task creation overhead
- Using tasks to count prime numbers
- Java versus the .NET Task Parallel Library
- Producer-consumer pipelines
- Bounded queues, thread wait() and notify()
- The states of a thread
- Java 8 stream implementation

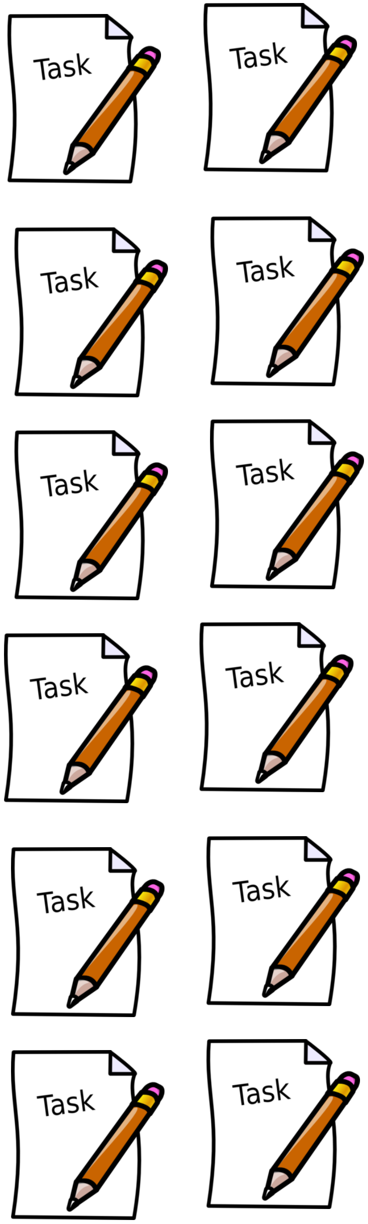
Prefer executors and tasks to threads

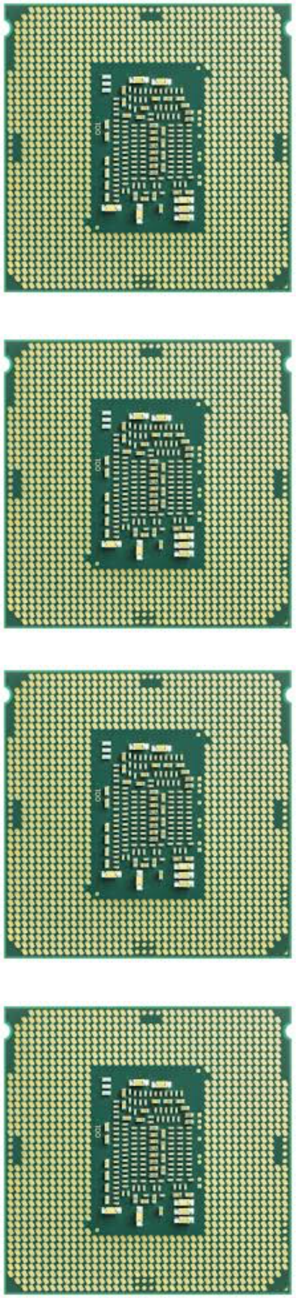
- We have used *threads* to parallelize work
  - But creating many threads takes time and memory
- Better divide work into (many small) *tasks*
  - Then submit the tasks to an executor
  - This uses a pool of (few) threads to run the tasks
- Goetz chapters 6, 8 and Bloch item 68



should generally refrain from working directly with threads. The key abstraction is no longer `Thread`, which served as both the unit of work and the mechanism for executing it. Now the unit of work and mechanism are separate. The key abstraction is the unit of work, which is called a *task*. There are two kinds of tasks: `Runnable` and its close cousin, `Callable` (which is like `Runnable`, except that it returns a value). The general mechanism for executing tasks is the *executor ser-*

Bloch item 68


Task
Task
Task
Task
Task
Task
Task
Task
Task
Task
Task
Task



Executors and tasks

- A *task* is just a Runnable or Callable<T>
- Submitting it to an *executor* gives a *Future*

```
Future<?> fut
  = executor.submit(new Runnable() { public void run() {
      System.out.println("Task ran!");
}});
```

TestTaskSubmit.java

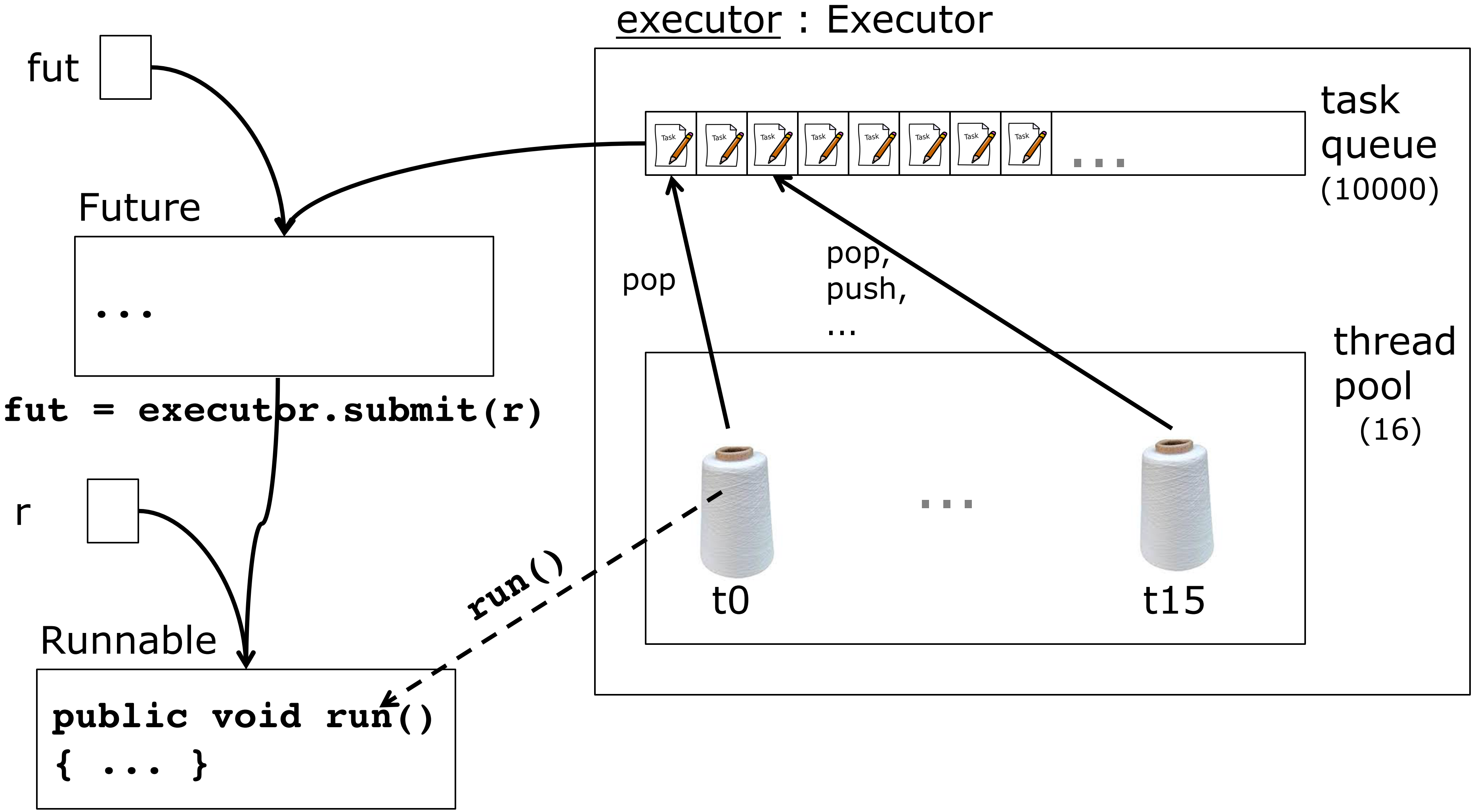
Same, using a lambda

```
Future<?> fut
  = executor.submit(() -> System.out.println("Task ran!"));
```

- The executor has a bunch of threads and uses one of them to run the task
- Use Future's **`get()`** to wait for task completion

```
try { fut.get(); }
catch (InterruptedException exn) { System.out.println(exn); }
catch (ExecutionException exn) { throw new RuntimeException(exn);
```

Dynamics of the executor framework



executor : Executor

fut

Future

...

`fut = executor.submit(r)`

r

Runnable

```
public void run()
{ ... }
```

`run()`

task
queue
(10000)

...

pop

pop,
push,
...

thread
pool
(16)

t0

...

t15

A task that produces a result

- Make the task from a Callable<T>

Future's result type

... same as Callable's

```
Future<String> fut
  = executor.submit(new Callable<String>() {
      public String call() throws IOException {
        return getPage("http://www.wikipedia.org", 10);
}});
```

TestTaskSubmit.java

- Use the Future to get the task's result:

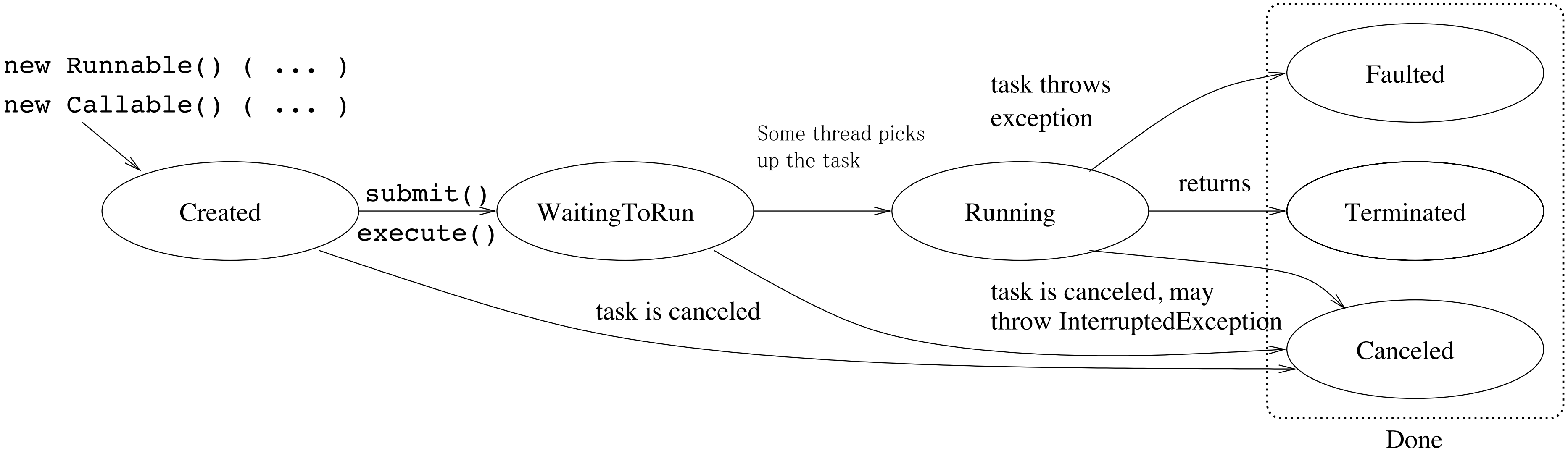
```
String webpage = fut.get();
System.out.println(webpage);
```

Task rules

- Different tasks may run on different threads
  - So objects accessed by tasks must be thread-safe
- A thread running a task can be interrupted
  - So a task can be interrupted
  - So **`fut.get()`** can throw InterruptedException

- Creating a task is fast, takes little memory
- Creating a thread is slow, takes much mem.

The states of a task



- After **`submit`** or **`execute`**
  - a task may be running immediately or much later
  - depending on the executor and available threads

Thread creation vs task creation

- Task creation is faster than thread creation

	Thread	Task
Work	6581 ns	6612 ns
Create	1030 ns	77 ns
Create+start/(submit+cancel)	48929 ns	835 ns
Create+(start/submit)+complete	72759 ns	21226 ns

Intel i7 2.4 GHz JVM 1.8

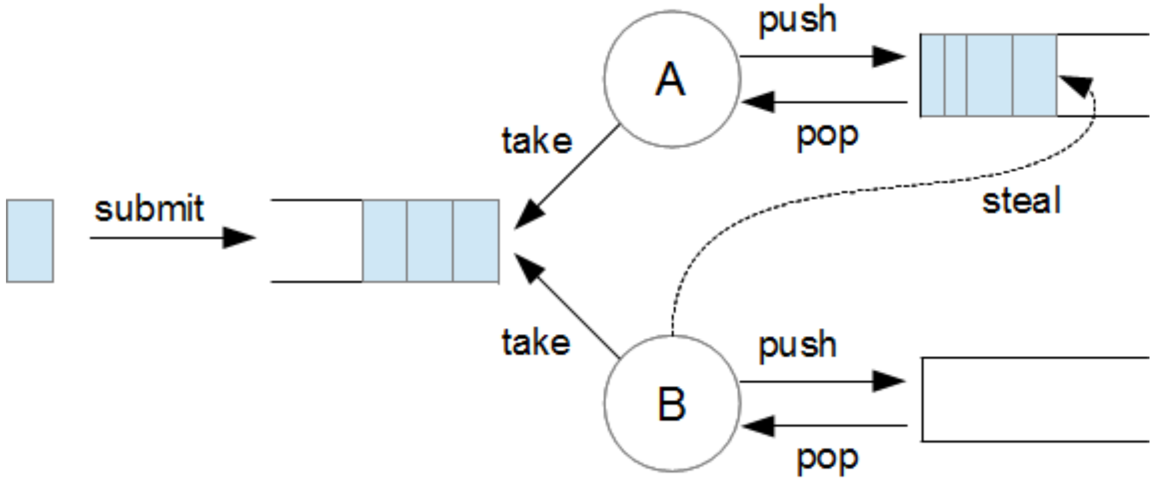
- A task also uses much less memory

Various Java executors

- In class java.util.concurrent.Executors:
- newFixedThreadPool(n)
  - Fixed number n of threads; automatic restart
- newCachedThreadPool()
  - Dynamically adapted number of threads, no bound
- newSingleThreadExecutor()
  - A single thread; so tasks need not be thread-safe
- newScheduledThreadPool()
  - Delayed and periodic tasks; eg clean-up, reporting
- newWorkStealingPool()
  - Adapts thread pool to number of processors, uses multiple queues; therefore better scalability

New in Java 8. Use it

Work Stealing Pool!



Plan for today

- Tasks and the Java executor framework
  - Executors, Runnables, Callables, Futures
- The states of a task
- Task creation overhead
- **Using tasks to count prime numbers**
- Java versus the .NET Task Parallel Library
- Producer-consumer pipelines
- Bounded queues, thread wait and notify
- The states of a thread

Week 1 flashback: counting primes in multiple threads

```
final LongCounter lc = new LongCounter();
Thread[] threads = new Thread[threadCount];
for (int t=0; t<threadCount; t++) {
   final int from = perTask * t,
               to = perTask * (t+1)
  threads[t] = new Thread(() -> {
    for (int i=from; i<to; i++)
      if (isPrime(i))
        lc.increment();
  });
}
for (int t=0; t<threadCount; t++)
  threads[t].start();
```

Thread processes segment [from,to)

- Creates one thread for each segment

Counting primes in multiple tasks

```
List<Future<?>> futures = new ArrayList<>();
for (int t=0; t<taskCount; t++) {
  final int from = perTask * t, to = perTask * (t+1);
  futures.add(executor.submit(() -> {
      for (int i=from; i<to; i++)
        if (isPrime(i))
          lc.increment();
  }));
}
try {
  for (Future<?> fut : futures)
    fut.get();
} catch (...) { ... }
```

Create task, submit to executor, save a future

Add to shared

Wait for all tasks to complete

TestCountPrimesTasks.java

- Creates a task for each segment
- The tasks execute on a thread pool

T2

Tasks with task-local counts

```
List<Callable<Long>> tasks = new ArrayList<>();
for (int t=0; t<taskCount; t++) {
  final int from = perTask * t, to = perTask * (t+1);
  tasks.add(() -> {
    long count = 0;
    for (int i=from; i<to; i++)
      if (isPrime(i))
        count++;
    return count;
  });
}
long result = 0;
try {
  List<Future<Long>> futures = executor.invokeAll(tasks);
  for (Future<Long> fut : futures)
    result += fut.get();
} catch (...) { ... }
```

Create task

Add to local

Submit tasks, wait for all to complete, get futures

Add local task results

TestCountPrimesTasks.java

Callable<Void> is like Runnable

T3

```
List<Callable<Void>> tasks = new ArrayList<>();
for (int t=0; t<taskCount; t++) {
  final int from = perTask * t, to = perTask * (t+1);
  tasks.add(() -> {
      for (int i=from; i<to; i++)
        if (isPrime(i))
          lc.increment();
      return null;
  });
}
try {
  executor.invokeAll(tasks);
} catch (...) { ... }
```

Create task

Add to shared

Submit tasks, wait for all to complete

TestCountPrimesTasks.java

Type parameters <Void> and <?>

- The type java.lang.Void contains only **`null`**
- Callable<Void> requires **`Void call() {...}`**
  - Similar to Runnable's **`void run() { ... }`**
  - With Future<Void> the **`get()`** returns null

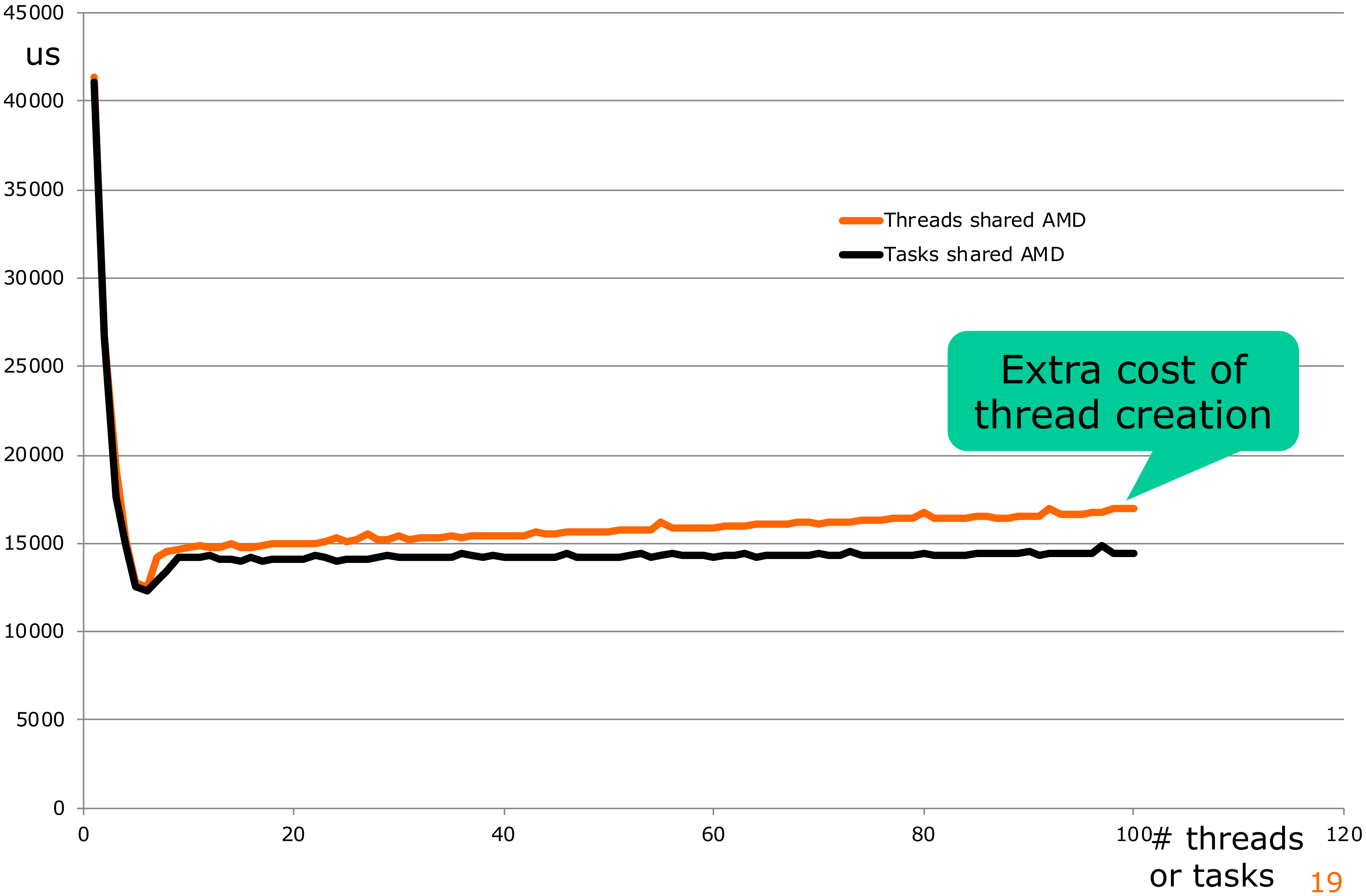
Not same

- Future<?> has an unknown type of value
  - With Future<?> the **`get()`** returns null also
- Java's type system is somewhat muddled
  - Forbids this assignment, so need Callable<Void>:

Type Future<?>

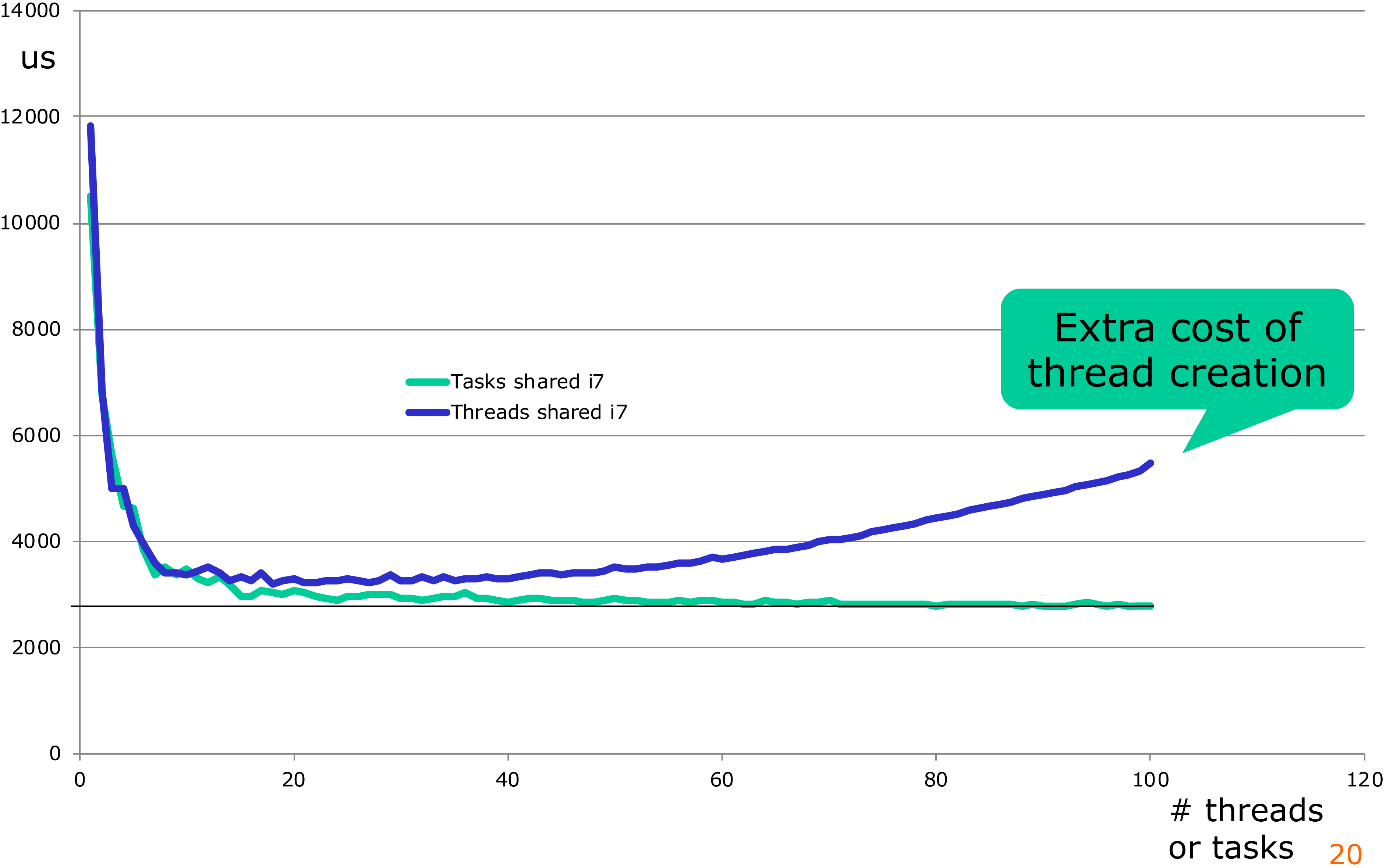
```
Future<Void> future;
future = executor.submit(new Runnable() { ... });
```

Overhead of creating threads (AMD)



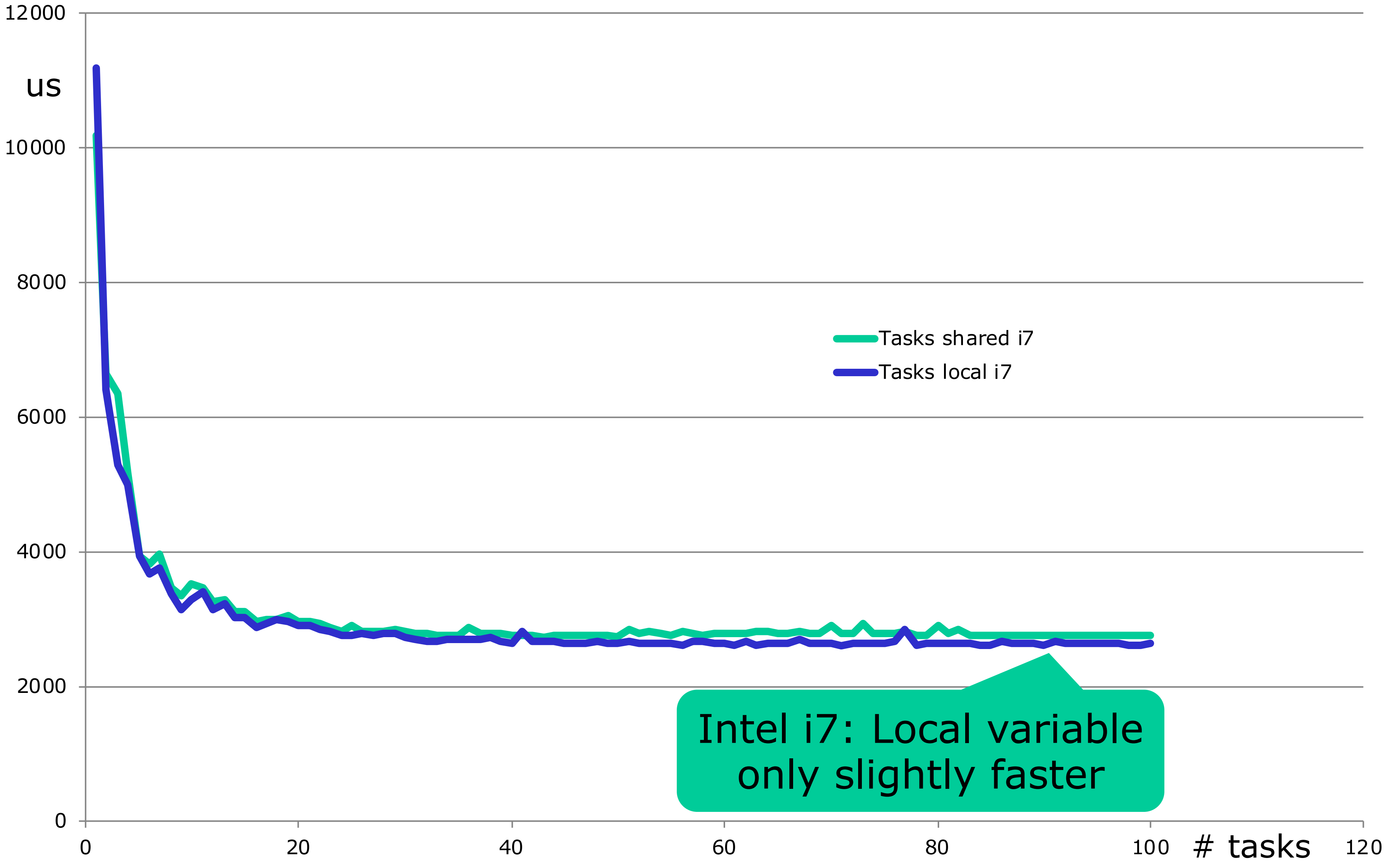
Extra cost of thread creation

Overhead of creating threads (i7)



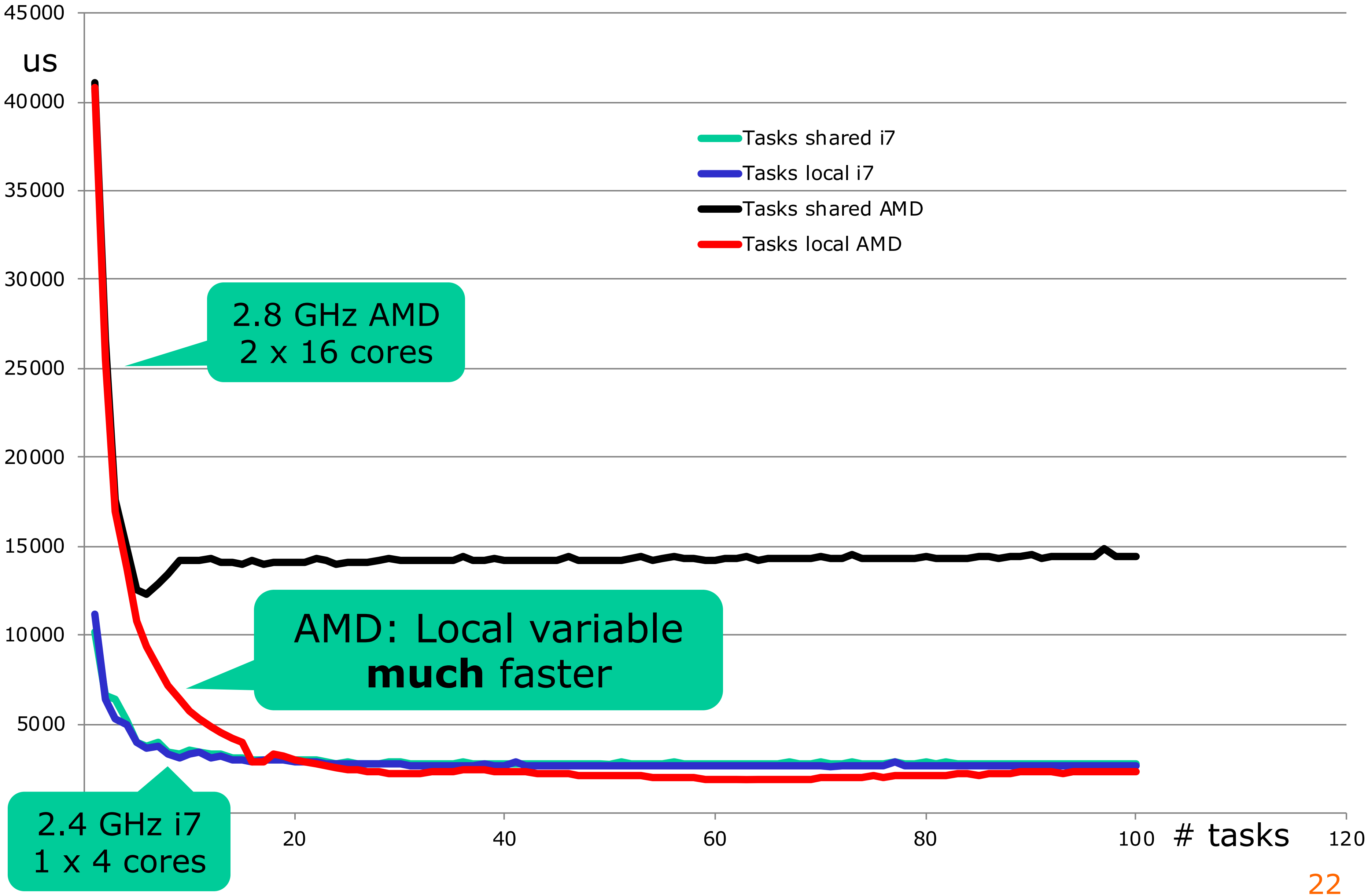
Extra cost of thread creation

Shared counter vs local counter



Intel i7: Local variable only slightly faster

Computers differ a lot



2.8 GHz AMD
2 x 16 cores

AMD: Local variable
much faster

2.4 GHz i7
1 x 4 cores

Plan for today

- Tasks and the Java executor framework
  - Executors, Runnables, Callables, Futures
- The states of a task
- Task creation overhead
- Using tasks to count prime numbers
- **Java versus the .NET Task Parallel Library**
- Producer-consumer pipelines
- Bounded queues, thread wait and notify
- The states of a thread

The .NET Task Parallel Library

- Since C#/.NET 4.0, 2010
- Easier to use and better language integration
  - **`async`** and **`await`** keywords in C#
  - .NET class library has more non-blocking methods
  - Java may get non-blocking methods … sometime
- Namespace System.Threading.Tasks
- Class Task combines Runnable & Future<?>
- Class Task<T> combines Callable<T> and Future<T>

- See *C# Precisely* chapters 22 and 23

C#T1

Parallel prime counts in C#, shared

```
int perTask = range / taskCount;
LongCounter lc = new LongCounter();
Parallel.For(0, taskCount, t => {
    int from = perTask * t, to = perTask * (t+1);
    for (int i=from; i<to; i++)
      if (isPrime(i))
        lc.increment();
  });
return lc.get();
```

Create tasks, submit to standard executor, run

Create task t

TestCountPrimesTasks.cs

- Same concepts as in Java
  - much leaner notation
  - easier to use out of the box
- The tasks are executed on a thread pool
  - in an unknown order

C#T2

Parallel prime counts in C#, local

```
long[] results = new long[taskCount];
Parallel.For(0, taskCount, t => {
    int from = perTask * t, to = perTask * (t+1);
    long count = 0;
    for (int i=from; i<to; i++)
      if (isPrime(i))
        count++;
    results[t] = count;
  });
return results.Sum();
```

TestCountPrimesTasks.cs

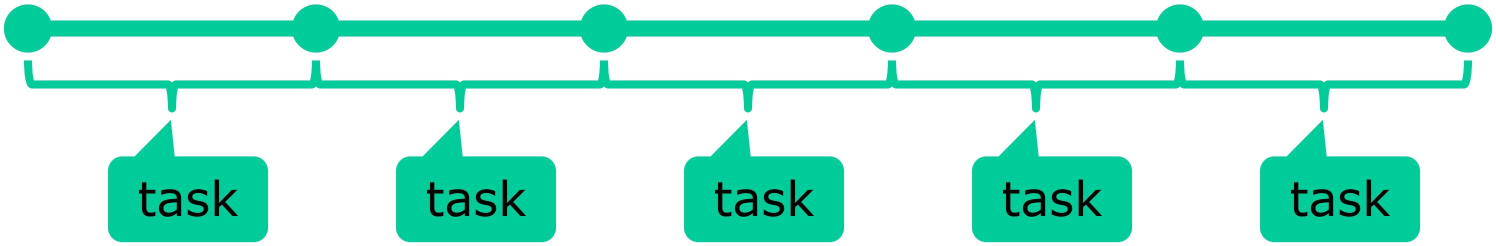
- Q: Why safe to write to **`results`** array?

Plan for today

- Tasks and the Java executor framework
  - Executors, Runnables, Callables, Futures
- The states of a task
- Task creation overhead
- Using tasks to count prime numbers
- Java versus the .NET Task Parallel Library
- **Producer-consumer pipelines**
- Bounded queues, thread wait and notify
- The states of a thread

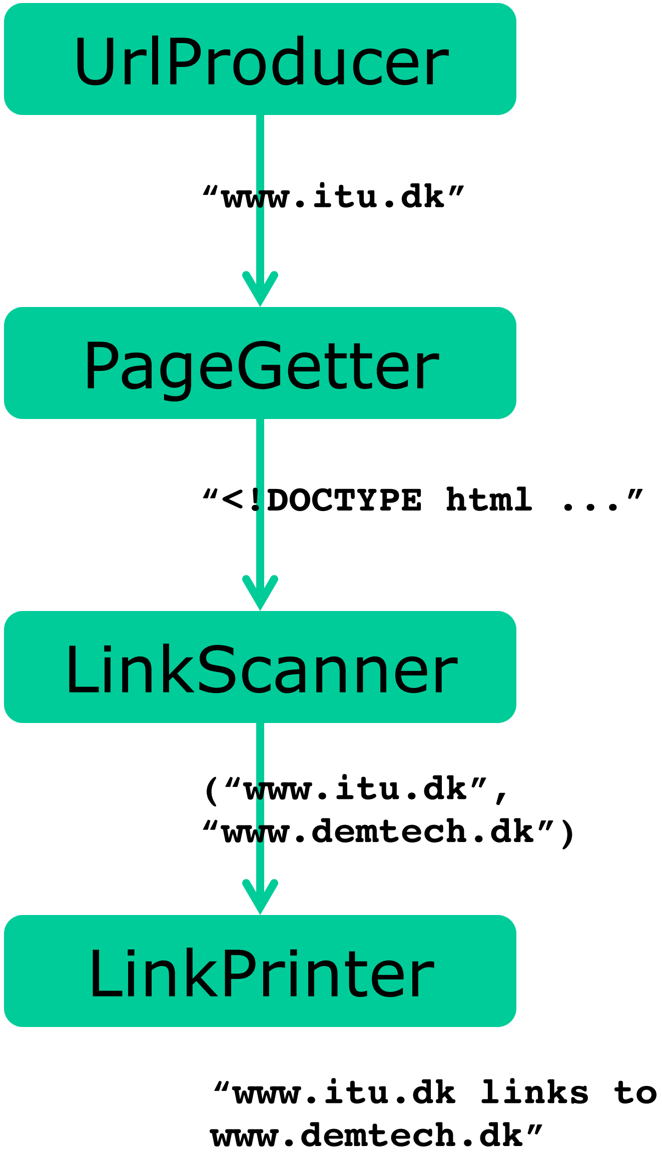
Concurrent pipelines (Goetz § 5.3)

- We parallelized prime counting by splitting the work into chunks:



- A different way is to create a *pipeline*
- Example problem: Given long list of URLs,
  - For each URL,
  - download the webpage at that URL
  - scan the webpage for links **`<a href="link">`** …
  - for each link, print "**`url`** links to **`link`**"

Pipeline to produce URL, get webpage, scan for links, and print them



UrlProducer

`“www.itu.dk”`

PageGetter

`“<!DOCTYPE html ...”`

LinkScanner

`(“www.itu.dk”, “www.demtech.dk”)`

LinkPrinter

`“www.itu.dk links to www.demtech.dk”`

- There are four stages
- They can run in parallel
  - On four threads
  - Or as four tasks
- Each does a simple job
- Two stages communicate via a blocking queue
  - **`queue.put(item)`** sends data item to next stage; blocks until room for data
  - **`queue.take()`** gets data item from previous stage; blocks until data available

Sketch of a one-item queue

TestPipeline.java

```
interface BlockingQueue<T> {
  void put(T item);
  T take();
}
```

```
class OneItemQueue<T> implements BlockingQueue<T> {
  private T item;
  private boolean full = false;
  public void put(T item) {
    synchronized (this) {
      full = true;
      this.item = item;
    }
  }
  public T take() {
    synchronized (this) {
      full = false;
      return item;
    }
  }
}
```

Java monitor pattern, good

But: what if already full?

If queue full, we must wait for **another** thread to **`take()`** first

But: What if queue empty?

Other thread can **`take()`** only if we release lock first

Useless

Using wait() and notifyAll()

TestPipeline.java

```
public void put(T item) {
  synchronized (this) {
    while (full) {
      try { this.wait(); }
      catch (InterruptedException exn) { }
    }
    full = true;
    this.item = item;
    this.notifyAll();
  }
}
```

If queue full, wait for notify from other thread

When non-full, save item, notify all waiting threads

- **`this.wait()`**: release lock on **`this`**; do nothing until notified, then acquire lock and continue
  - Must hold lock on **`this`** before call
- **`this.notifyAll()`**: tell all threads **`wait()`**ing on **`this`** to wake up
  - Must hold lock on **`this`**, and keeps holding it

The take() method is similar

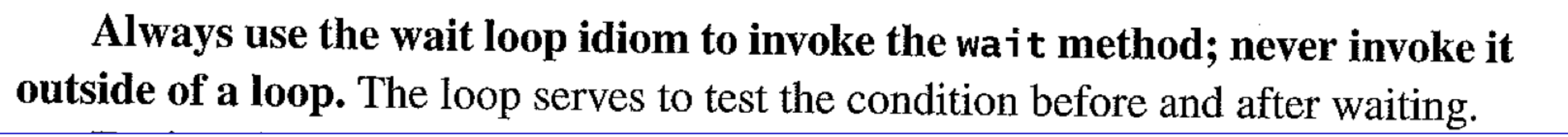
TestPipeline.java

```
public T take() {
  synchronized (this) {
    while (!full) {
      try { this.wait(); }
      catch (InterruptedException exn) { }
    }
    full = false;
    this.notifyAll();
    return item;
  }
}
```

If queue empty, wait for notify from other thread

When non-empty, take item, notify all waiting threads

- Only works if **all** methods locking on the queue are written correctly
- MUST do the wait() in a while loop; Q: Why?

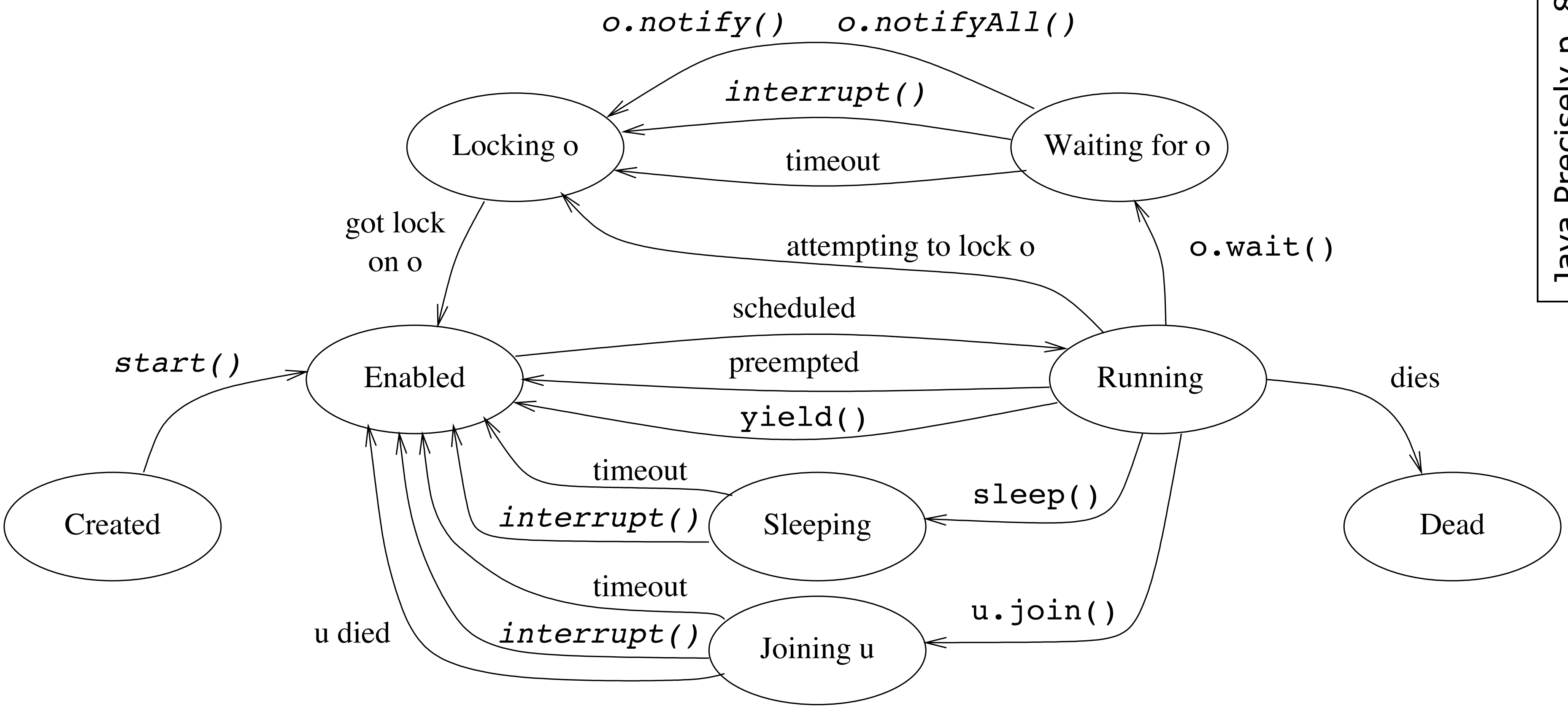


Always use the wait loop idiom to invoke the `wait` method; never invoke it outside of a loop. The loop serves to test the condition before and after waiting.

Bloch p. 276

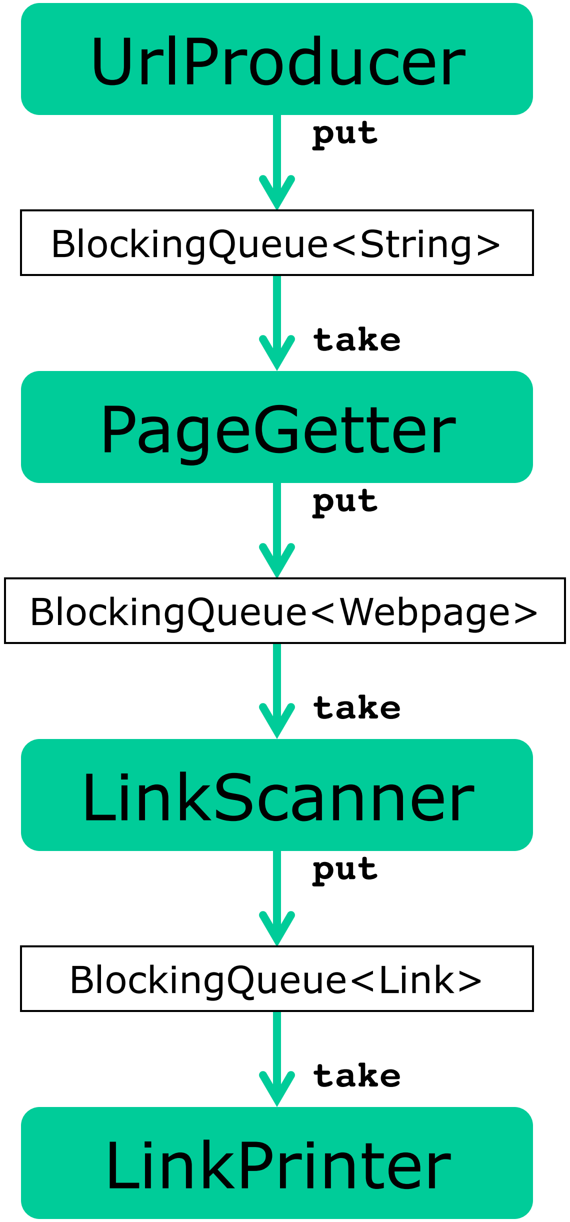
Java Thread states

Java Precisely p. 81



- **`o.wait()`** is an action of the running thread itself
- ***`o.notify()`*** is an action by another thread, on the waiting one
- scheduled, preempted, ... are actions of the system

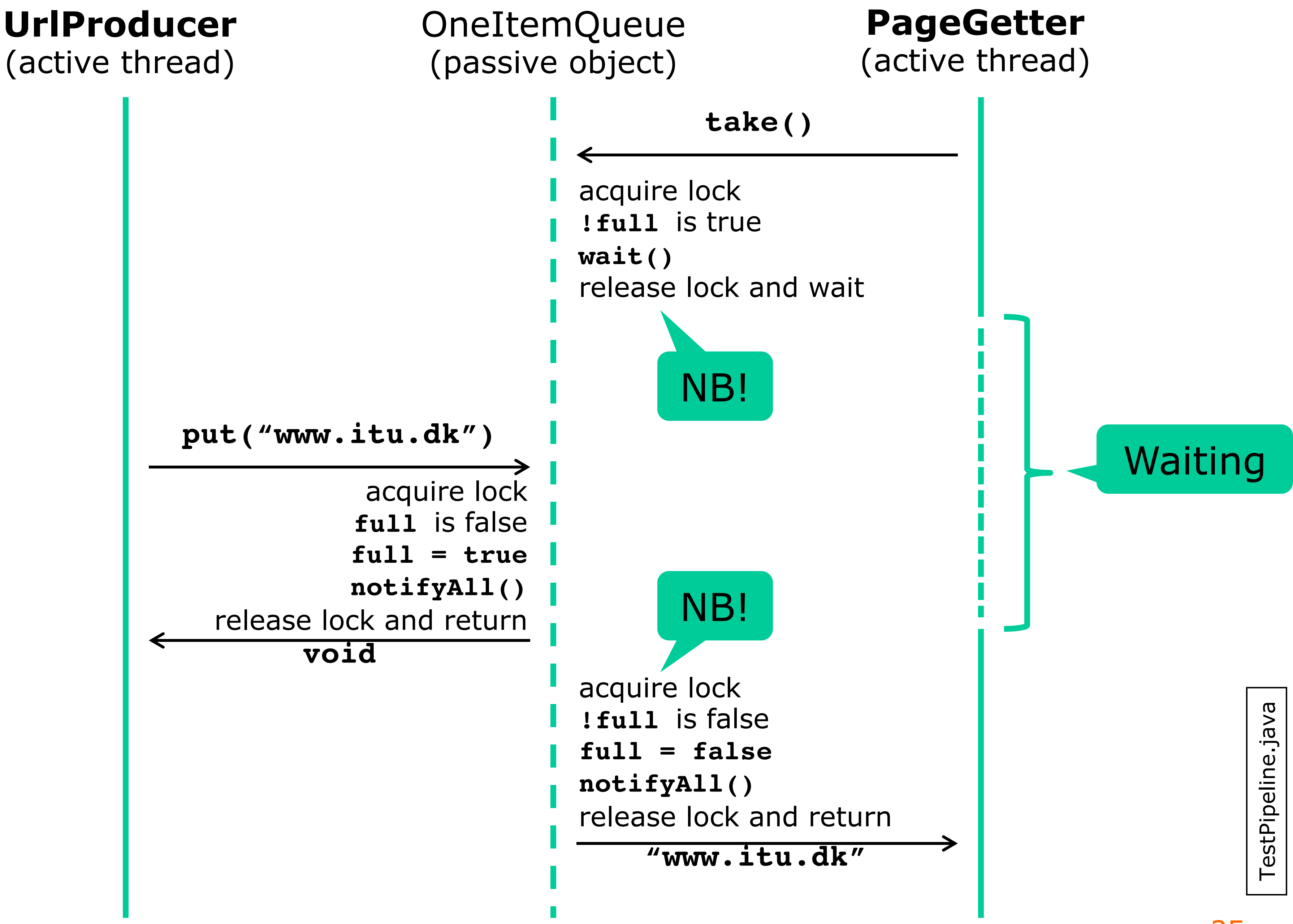
Producer-consumer pattern: Pipeline stages and connecting queues



- The first stage is a producer only
- The middle stages are both consumers and producers
- The last stage is only a consumer

- A queue connects producer(s) to consumer(s) in a thread-safe way

How wait and notifyAll collaborate



UrlProducer
(active thread)

OneItemQueue
(passive object)

PageGetter
(active thread)

`take()`

acquire lock
`!full` is true
`wait()`
release lock and wait

NB!

Waiting

`put("www.itu.dk")`

acquire lock
`full` is false
`full = true`
`notifyAll()`
release lock and return
`void`

NB!

acquire lock
`!full` is false
`full = false`
`notifyAll()`
release lock and return
`"www.itu.dk"`

TestPipeline.java

Stages 1 and 2

TestPipeline.java

```
class UrlProducer implements Runnable {
  private final BlockingQueue<String> output;
  public UrlProducer(BlockingQueue<String> output) {
    this.output = output;
  }
  public void run() {
    for (int i=0; i<urls.length; i++)
      output.put(urls[i]);
  }
}
```

Produce URLs

```
class PageGetter implements Runnable {
  ...
  public void run() {
    while (true) {
      String url = input.take();
      try {
        String contents = getPage(url, 200);
        output.put(new Webpage(url, contents));
      } catch (IOException exn) { System.out.println(exn); }
    }
  }
```

Transform URL to webpage

Stages 3 and 4

```
class LinkScanner implements Runnable {
  ...
  private final static Pattern urlPattern
    = Pattern.compile("a href=\"(\\p{Graph}*)\"");
  public void run() {
    while (true) {
      Webpage page = input.take();
      Matcher urlMatcher = urlPattern.matcher(page.contents);
      while (urlMatcher.find()) {
        String link = urlMatcher.group(1);
        output.put(new Link(page.url, link));
} } } }
```

Transform web page to link stream

```
class LinkPrinter implements Runnable {
  ...
  public void run() {
    while (true) {
      Link p = input.take();
      System.out.printf("%s links to %s%n", p.from, p.to);
} } }
```

Consume links and print them

Putting stages and queues together

```
final BlockingQueue<String> urls = new OneItemQueue<String>();
final BlockingQueue<Webpage> pages = new OneItemQueue<Webpage>();
final BlockingQueue<Link> refPairs = new OneItemQueue<Link>();
Thread t1 = new Thread(new UrlProducer(urls));
Thread t2 = new Thread(new PageGetter(urls, pages));
Thread t3 = new Thread(new LinkScanner(pages, refPairs));
Thread t4 = new Thread(new LinkPrinter(refPairs));
t1.start(); t2.start(); t3.start(); t4.start();
```

TestPipeline.java

- Each stage does *one* job
  - Simple to implement and easy to modify
  - Separation of concerns, simple control flow
- Easy to add new stages
  - For instance, discard duplicate links
- Can achieve high throughput
  - May run multiple copies of a slow stage

"Prefer concurrency utilities to wait and notify"

Bloch item 69

- It's instructive to use **`wait`** and **`notify`**
- … but easy to do it wrong
- Package java.util.concurrent has
  - BlockingQueue<T> interface
  - ArrayBlockingQueue<T> class and much more
- Better use those in practice

- … or make a pipeline with Java 8 streams
  - Simpler, and *very* easy to parallelize

Using Java 8 streams instead

- Package java.util.stream
- A Stream<T> is a source of T values
  - Lazily generated
  - Can be transformed with **`map(f)`** and **`flatMap(f)`**
  - Can be filtered with **`filter(p)`**
  - Can be consumed by **`forEach(action)`**
- Generally simpler than concurrent pipeline

```
Stream<String> urlStream
  = Stream.of(urls);
Stream<Webpage> pageStream
  = urlStream.flatMap(url -> makeWebPageOrNone(url, 200));
Stream<Link> linkStream
  = pageStream.flatMap(page -> makeLinks(page));
linkStream.forEach(link ->
    System.out.printf("%s links to %s%n", link.from, link.to));
```

TestStreams.java

Making the stages run in parallel

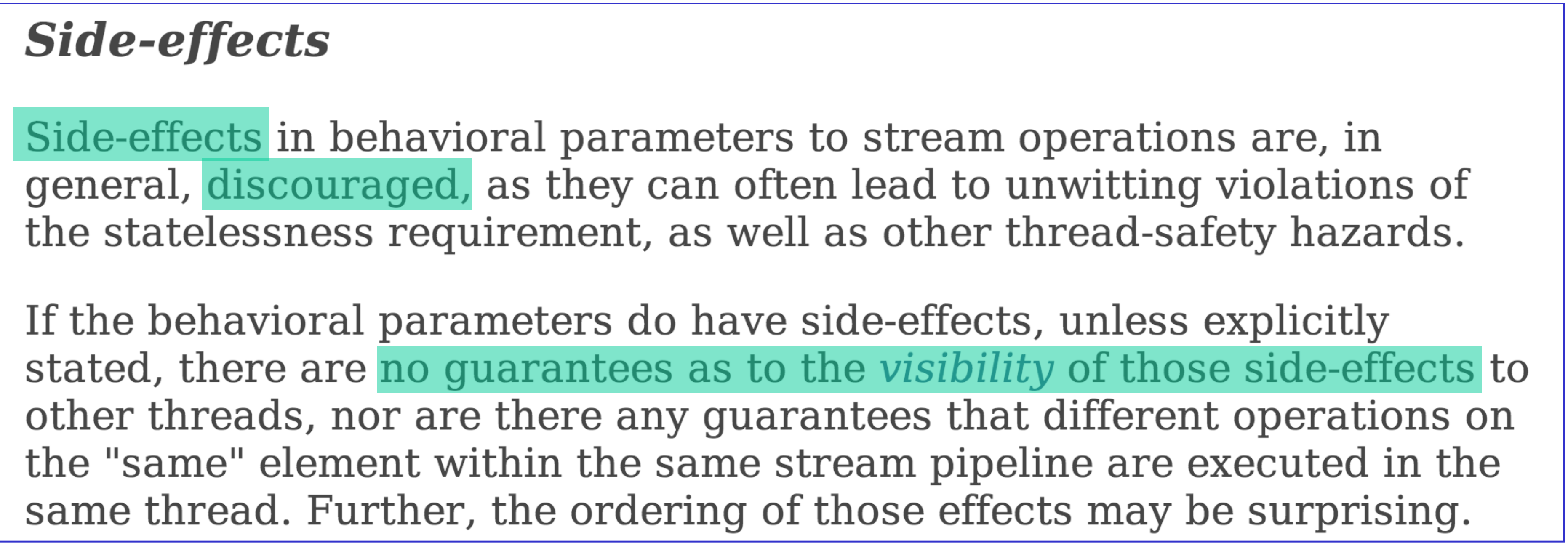
```
Stream<String> urlStream
  = Stream.of(urls).parallel();
Stream<Webpage> pageStream
  = urlStream.flatMap(url -> makeWebPageOrNone(url, 200));
Stream<Link> linkStream
  = pageStream.flatMap(page -> makeLinks(page));
linkStream.forEach(link ->
    System.out.printf("%s links to %s%n", link.from, link.to));
```

TestStreams.java

- Magic? No!
- Divides streams into substream chunks
- Evaluates the chunks in tasks
- Runs tasks on an executor called ForkJoinPool
  - Using a thread pool and work stealing queues
  - More precisely ForkJoinPool.commonPool()

So easy. Why learn about threads?

- Parallel streams use tasks, run on threads
- Should be **side effect free** and take no locks
- Otherwise all the usual thread problems:
  - updates must be made atomic (by locking)
  - updates must be made visible (by locking, volatile)
  - deadlock risk if locks are taken



Side-effects

Side-effects in behavioral parameters to stream operations are, in general, discouraged, as they can often lead to unwitting violations of the statelessness requirement, as well as other thread-safety hazards.

If the behavioral parameters do have side-effects, unless explicitly stated, there are no guarantees as to the *visibility* of those side-effects to other threads, nor are there any guarantees that different operations on the "same" element within the same stream pipeline are executed in the same thread. Further, the ordering of those effects may be surprising.

Java 8 class library documentation

How are Java streams implemented?

- Spliterator = splittable iterator

```
interface Spliterator<T> {
  void forEachRemaining(Consumer<T> action);
  boolean tryAdvance(Consumer<T> action);
  void Spliterator<T> trySplit();
}
```

  - Many method calls (well inlined/fused by the JIT)
- Parallelization
  - Divide stream into chunks
  - Process each chunk in a task
  - Run on thread pool using queues

Example: array spliterator def.

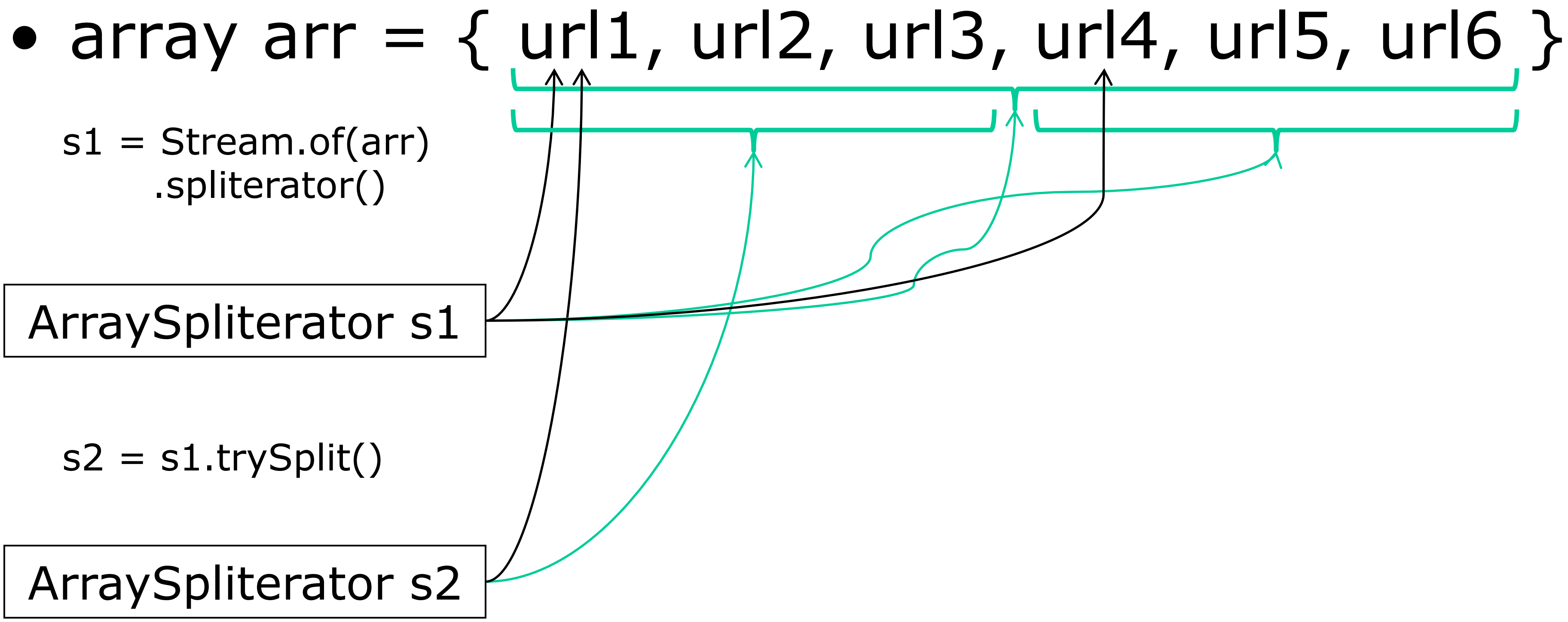
```
class ArraySpliterator<T> implements Spliterator<T> {
  private final Object[] array; // underlying array
  private int index;            // next index, modified on advance/split
  private final int fence;      // one past last index

  public boolean tryAdvance(Consumer<? super T> action) {
    if (index >= 0 && index < fence) {
      action.accept((T) array[index++]);
      return true;
    } else
      return false;
  }

  public Spliterator<T> trySplit() {
    int lo = index, mid = (lo + fence) >>> 1;
    return (lo >= mid)
      ? null
      : new ArraySpliterator<>(array, lo, index = mid, characteristics);
  }

  public void forEachRemaining(Consumer<? super T> action) { ... }
}
```

Consume one element

Split into two Spliterators

http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/8-b132/java/util/Spliterators.java

Example: array spliterator



- array arr = { url1, url2, url3, url4, url5, url6 }

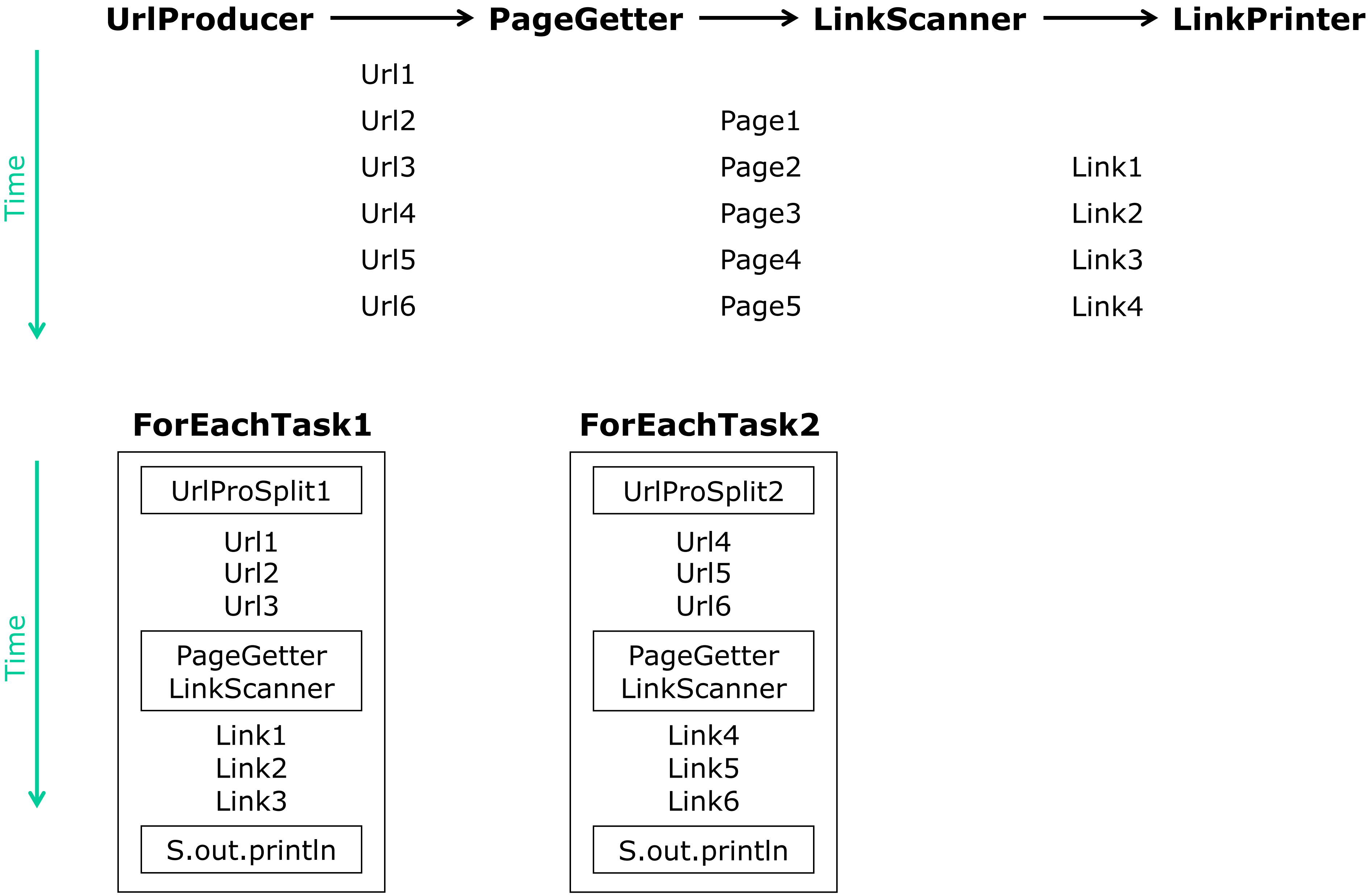
s1 = Stream.of(arr)
.spliterator()

ArraySpliterator s1

s2 = s1.trySplit()

ArraySpliterator s2

Thread vs Streams parallel pipelines



UrlProducer → **PageGetter** → **LinkScanner** → **LinkPrinter**

Time

Url1		
Url2	Page1	
Url3	Page2	Link1
Url4	Page3	Link2
Url5	Page4	Link3
Url6	Page5	Link4

Time

ForEachTask1

UrlProSplit1

Url1
Url2
Url3

PageGetter
LinkScanner

Link1
Link2
Link3

S.out.println

ForEachTask2

UrlProSplit2

Url4
Url5
Url6

PageGetter
LinkScanner

Link4
Link5
Link6

S.out.println

This week

- Reading
  - Goetz et al chapters 5.3, 6 and 8
  - Bloch items 68, 69
- Exercises week 5
  - Show that you can use tasks and the executor framework, and modify a concurrent pipeline