1 Threads Scheduling

With simpler tools…

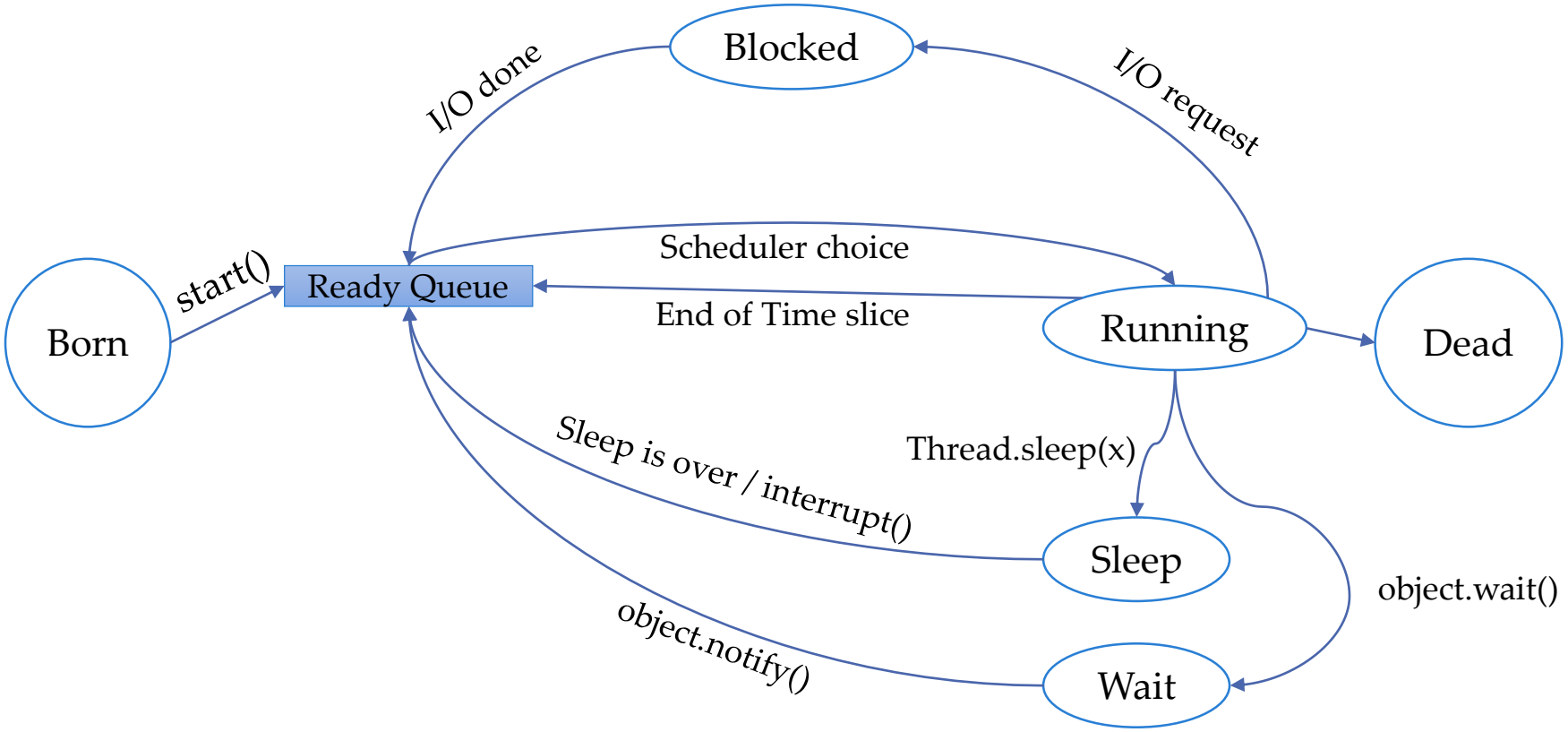
Dr. Eliahu Khalastchi

Introduction

- By now, you know how to program threads
- But in a very basic level that matches *low-level* implementations
- For *higher-level* code, we need advanced tools
  - Tools that will hide the threads logic from us
  - Makes it easier for us to control threads
  - Mostly are from ***java.util.concurrent***
  - introduced in java 1.5

Scheduling

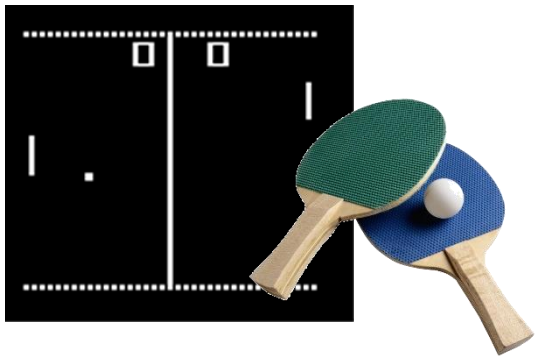
- We can use sleep() & wait() to influence thread scheduling



Scheduling Tasks

```
public class Ping implements Runnable{
 public void run(){while(true)System.out.println("ping");}
}
public class Pong implements Runnable{
 public void run(){while(true)System.out.println("pong");}
}
 public static void main(String[] args) {
  Ping ping=new Ping();
  Pong pong=new Pong();
  Thread t=new Thread(ping,"thread 1");
  Thread t1=new Thread(pong,"thread 2");
  t.start();
  t1.start();
 }
```

- We want an ordered "ping-pong" sequence
- Half a second apart
- Does this code meet the demands?



Scheduling Tasks

- Here is a solution using *sleep()*

```
public class Ping implements Runnable{
  public void run(){
   while(true){
    System.out.println("ping");
    try {Thread.sleep(1000);}
    catch (InterruptedException e) {}
   }
  }
}
// pong is the same…
```

```
public static void main(String[] args) throws InterruptedException {
  Ping ping=new Ping();
  Pong pong=new Pong();
  Thread t=new Thread(ping,"thread 1");
  Thread t1=new Thread(pong,"thread 2");
  t.start();
  Thread.sleep(500); // the main sleeps 0.5 sec
  t1.start();
}
```

Scheduling Tasks – with a simple Timer!

```
import java.util.Timer;
import java.util.TimerTask;
public class ThreadTest {
 private static class Ping extends TimerTask{
  public void run(){System.out.println("ping");}
 }
 private static class Pong extends TimerTask{
  public void run(){System.out.println("pong");}
 }
 public static void main(String[] args){
  Ping ping=new Ping();
  Pong pong=new Pong();
  Timer t=new Timer();
  t.scheduleAtFixedRate(ping, 0, 1000);
  t.scheduleAtFixedRate(pong, 500, 1000);
 }
}
```

Canceling tasks:

```
int i;
while((i=System.in.read())!=13);
ping.cancel(); // canceled task
pong.cancel(); // t continues…
t.cancel(); // t is cancled
```

2 Replacing Synchronized

With faster / none-blocking locks

Dr. Eliahu Khalastchi

Using Synchronized is slow…

```
public class Count {
 private int count;
 public void setCount(int x){count=x;}
 public int getCount(){return count;}
 public synchronized void update(){count++;}
}
```

```
public class CountUpdater implements Runnable{
 Count c;
 public CountUpdater(Count c){this.c=c;}
 public void run(){
  for(int i=0;i<100000000; c.update(),i++);
 }
}
```

```
public static void main(String[] args) {
  Count c=new Count();
  c.setCount(0);
  CountUpdater ca=new CountUpdater(c);
  Thread t=new Thread(ca);
  Thread t1=new Thread(ca);
  long time=System.currentTimeMillis();
  t.start();
  t1.start();
  t.join();
  t1.join();
  System.out.println(c.getCount());
  long duration=(System.currentTimeMillis()-time)/1000;
  System.out.println(duration);
 }
```

46 seconds!!!

Using Atomic Variables

```
import java.util.concurrent.atomic.AtomicInteger;
public class Count {
 AtomicInteger count = new AtomicInteger(0);
 public  void setCount(int x){count.set(x);}
 public int getCount(){return count.get();}
 public void update(){
  count.incrementAndGet();// ++count
 }
}
```

```
public class CountUpdater implements Runnable{
 Count c;
 public CountUpdater(Count c){this.c=c;}
 public void run(){
  for(int i=0;i<100000000; c.update(),i++);
 }
}
```

```
public static void main(String[] args) {
  Count c=new Count();
  c.setCount(0);
  CountUpdater ca=new CountUpdater(c);
  Thread t=new Thread(ca);
  Thread t1=new Thread(ca);
  long time=System.currentTimeMillis();
  t.start();
  t1.start();
  t.join();
  t1.join();
  System.out.println(c.getCount());
  long duration=(System.currentTimeMillis()-time)/1000;
  System.out.println(duration);
 }
```

6 seconds!!!

3 Deadlock Example

And how to avoid it…

Dr. Eliahu Khalastchi

Deadlock Example

```
Object R=new Object();// readers lock
Object W=new Object();// writers lock
```

```
new Thread(new Runnable() {
    @Override
    public void run() {

        synchronized (W) {
            // do the writing...
            synchronized (R) {
                // do some reading...
            }
            // do more writing...
        }
    }
}).start();
```

```
new Thread(new Runnable() {
    @Override
    public void run() {

        synchronized (R) {
            // do the reading...
            synchronized (W) {
                // do some writing...
            }
            // do more reading...
        }
    }
}).start();
```

Deadlock avoidance with tryLock()

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

ReentrantLock W=new ReentrantLock();
ReentrantLock R=new ReentrantLock();
```

- reentrantLock allows to use **tryLock();**
- It returns **true / false**
  - **Instead of just blocking like synchronized…**

- Only if we manage to lock both locks
  - We do the reading & writing
- Else, we try again later

- Finally, we unlock whatever lock we may have locked

```
public void run() {
    boolean w=W.tryLock();
    boolean r=R.tryLock();
    try{
        if(w && r){
            // do the writing...
            // do some reading...
            // do more writing...
        } else{
            // try again later...
        }
    }finally{
        if(w) W.unlock();
        if(r) R.unlock();
    }
}
```

4 Thread-safe containers

Dr. Eliahu Khalastchi

Thread Safe Containers

- Most of java.util containers are not thread safe
  - Because synchronize slows performance
- They could be wrapped with synchronized decorators
  - Only when we must, we'll pay with performance

```
Map<String,Integer> hm =
        Collections.synchronizedMap(
                new HashMap<String,Integer>());
```

- Decorator Pattern!
- Every method is implemented with synchronized

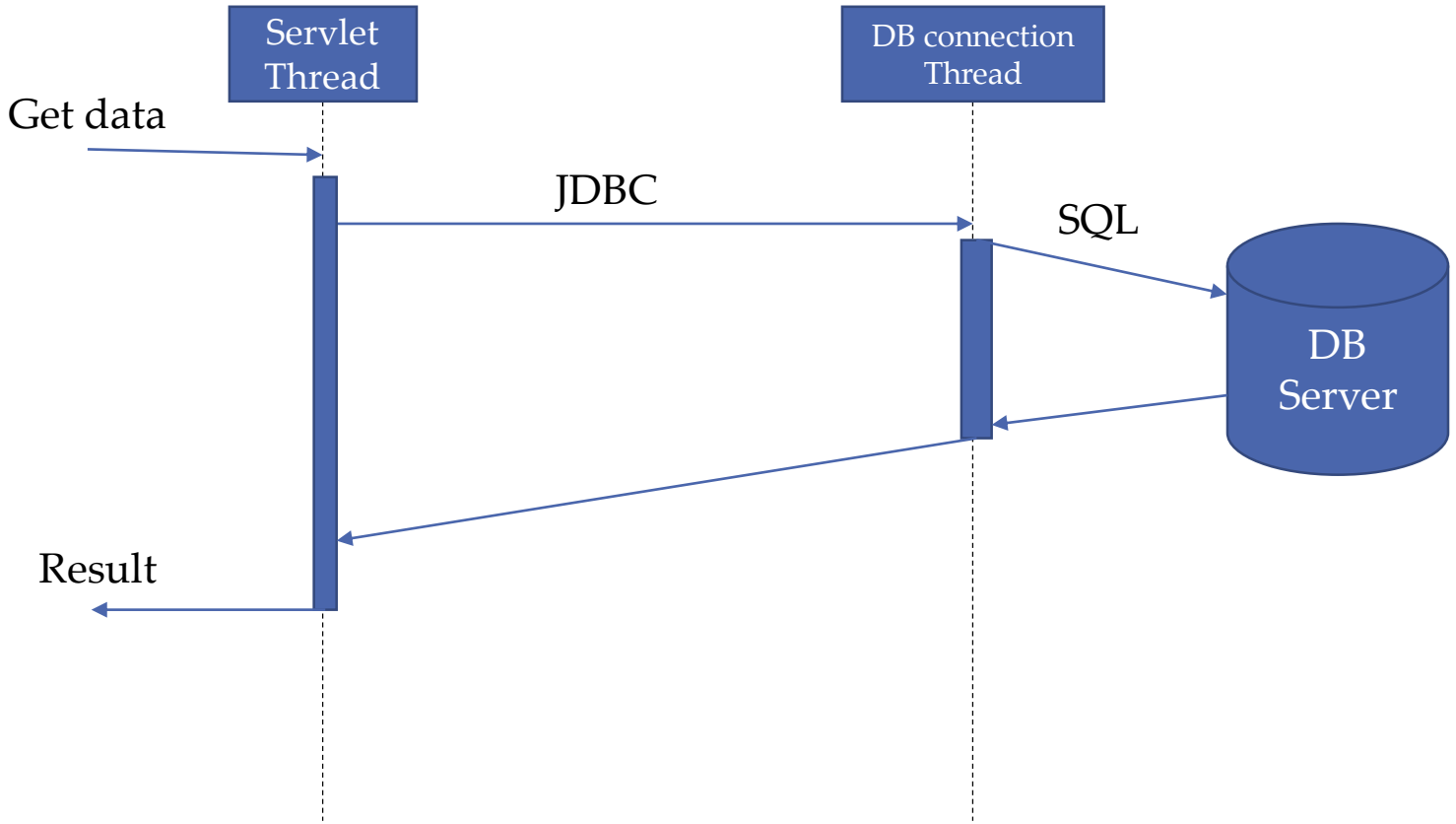
Thread Safe Containers

- java.util.concurrent introduced **Thread Safe** containers,
- that also provides good performance!
  - **ArrayBlockingQueue<E>**
  - **ConcurrentHashMap<K,V>**
  - **ConcurrentLinkedQueue<E>**
  - etc…

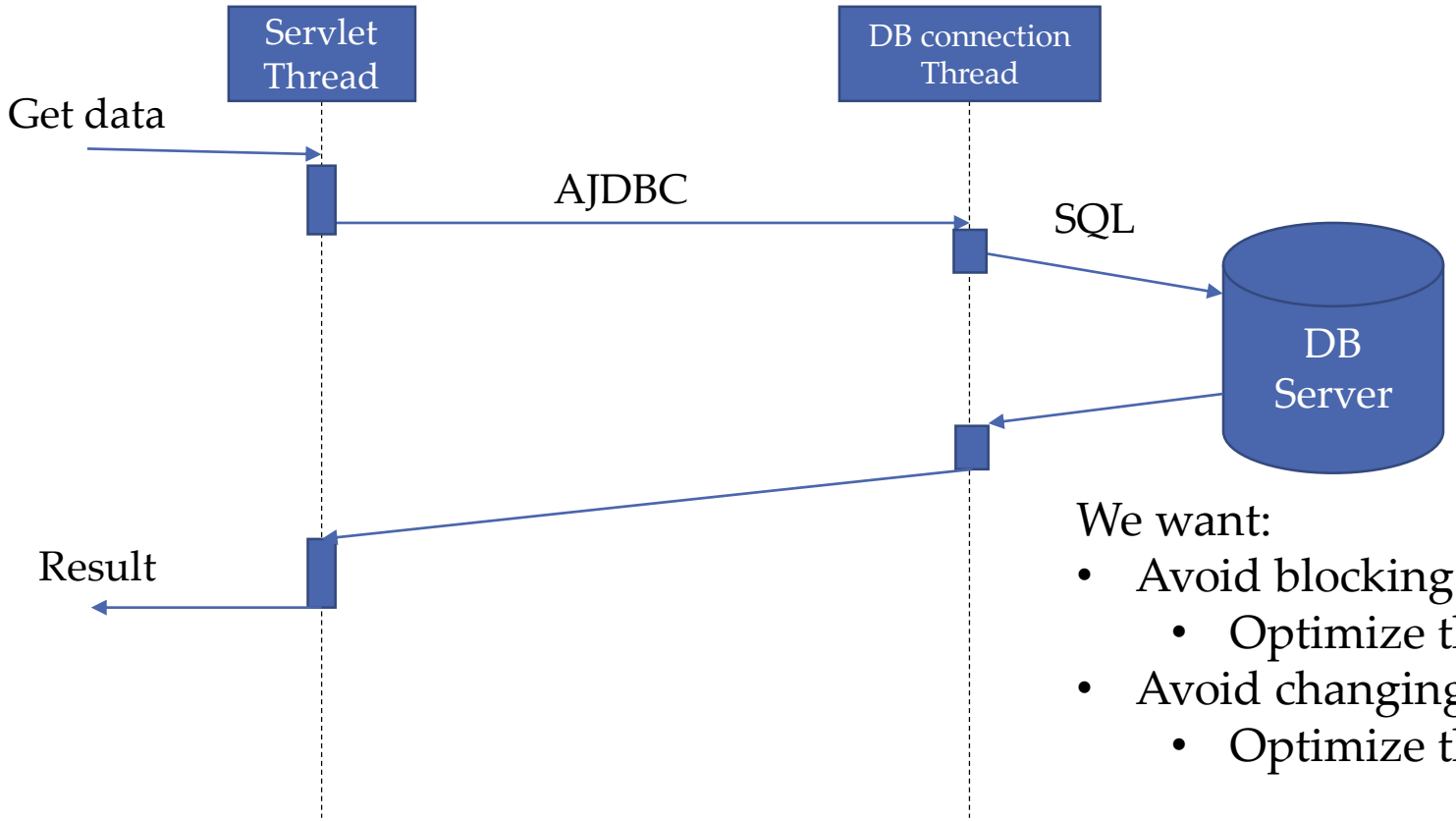
5 Java8 Additions

Dr. Eliahu Khalastchi

Blocking (yet asynchronous)



Non-Blocking



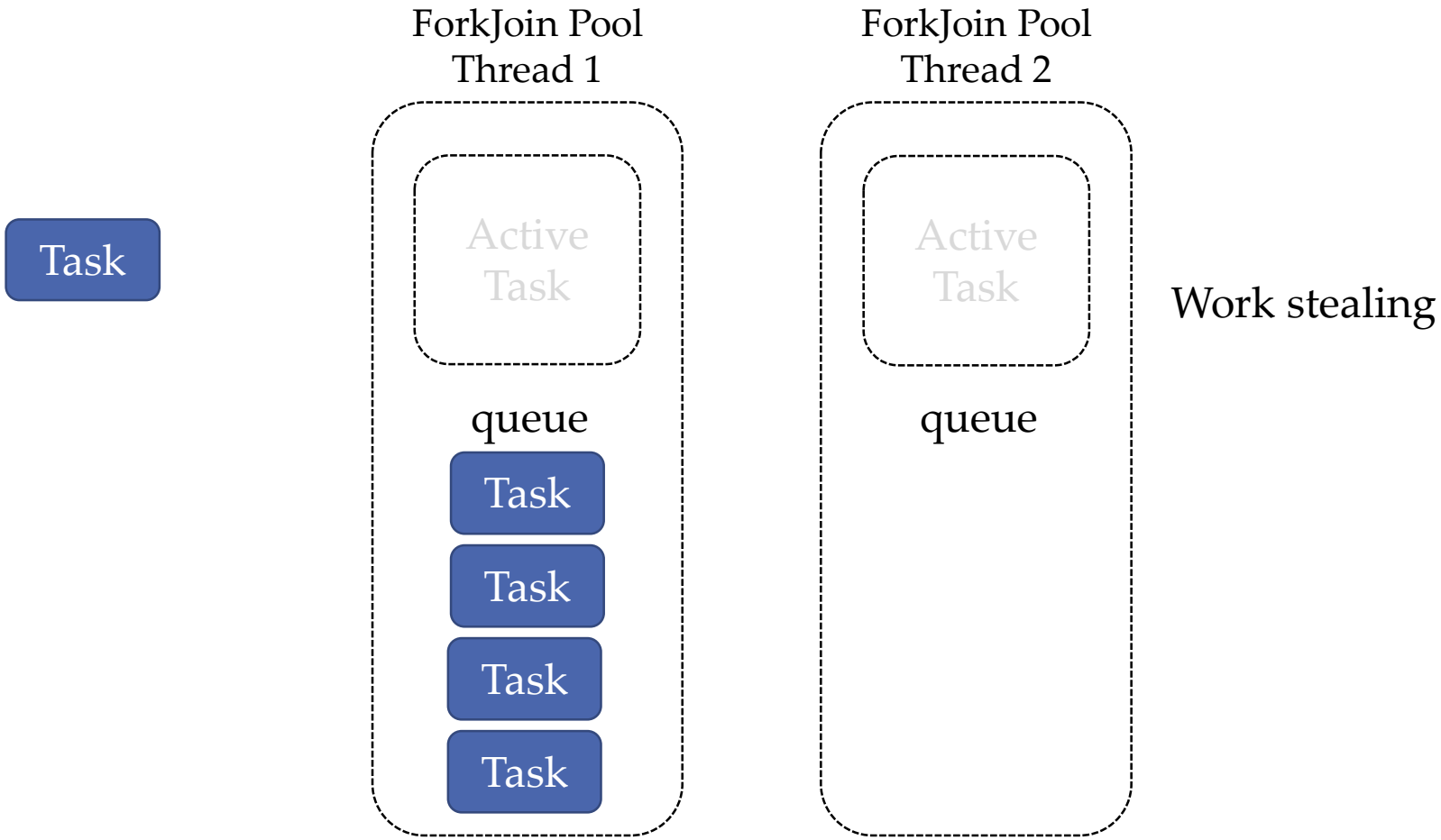
We want:

- Avoid blocking
  - Optimize the use of a multicore
- Avoid changing threads
  - Optimize the use of cache

Fork-Join Pool

Java 7

ForkJoin Pool (JDK 7)

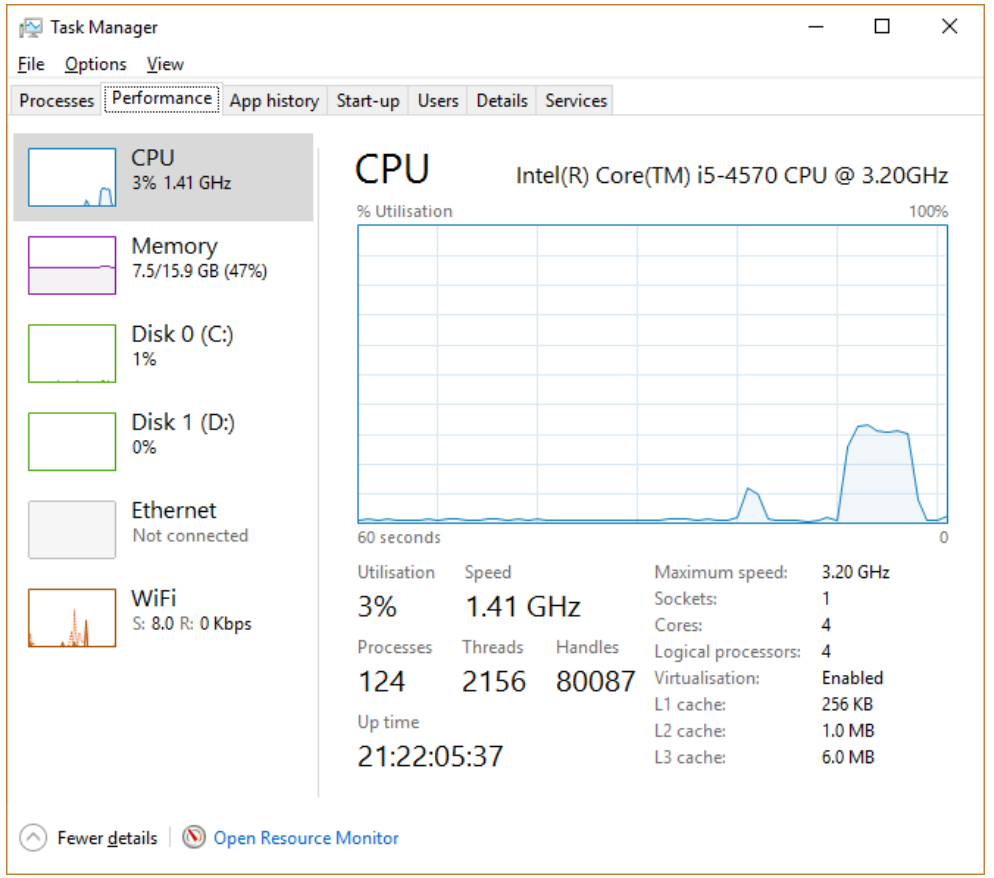


Fibonacci Example

```
public class Fib {

 int num;
 public Fib(int num) {
  this.num=num;
 }

 public int compute(){
  if(num<=1)
   return num;
  Fib fib1= new Fib(num-1);
  Fib fib2= new Fib(num-2);
  return fib2.compute()+fib1.compute();
 }
 public static void main(String[] args) {
  System.out.println(new Fib(45).compute());
 }
}
```

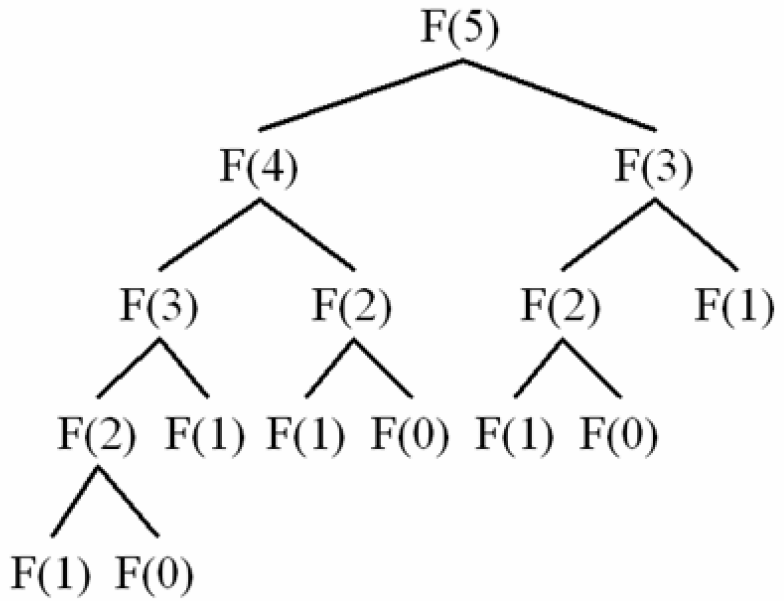


Fibonacci + Dynamic Programming

```
public class Fib_DP { // without concurrency
                      // but with dynamic programming
 static HashMap<Integer,Integer> fibs=new HashMap<>();

 int num;
 public Fib_DP(int num) { this.num=num;}

 public int compute(){ // a recursive task
  if(num<=1)
   return num;
  if(fibs.get(num)!=null)
   return fibs.get(num);
  Fib_DP fib1= new Fib_DP(num-1);
  Fib_DP fib2= new Fib_DP(num-2);
  int result=fib2.compute()+fib1.compute();
  fibs.put(num,result);
  return result;
 }
 public static void main(String[] args) {
  System.out.println(new Fib_DP(2048).compute());
 }
}
```



However, we wish to simulate a multithreaded task

Fibonacci + Thread Pool (JDK 6)

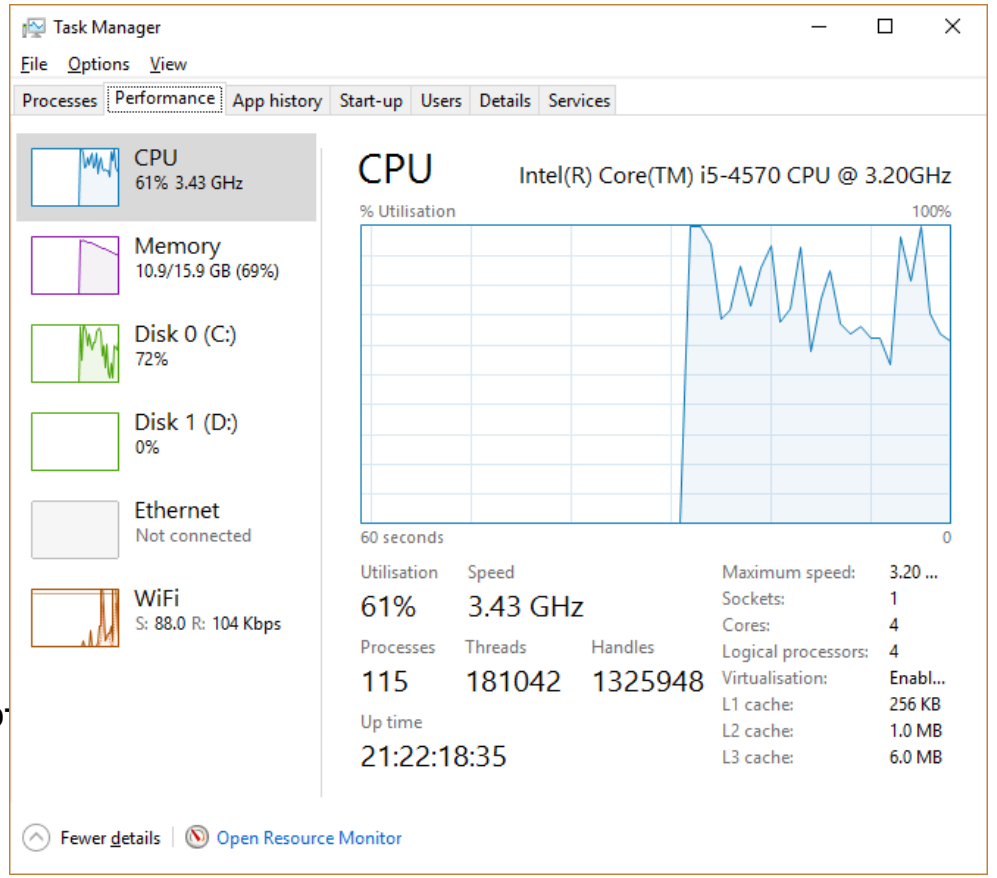
```
public class Fib_TP implements Callable<Integer>{

 static ExecutorService es=Executors.newCachedThreadPool();

 int num;
 public Fib_TP(int num) {this.num=num;}

 @Override
 public Integer call() throws Exception {
  if(num<=1)
   return num;
  Future<Integer> fib1 = es.submit(new Fib_TP(num-1));
  Future<Integer> fib2 = es.submit(new Fib_TP(num-2));
  return fib2.get()+fib1.get();
 }

 public static void main(String[] args) throws InterruptedExcep
  Future<Integer> f=es.submit(new Fib_TP(45));
  System.out.println(f.get());
 }
}
```

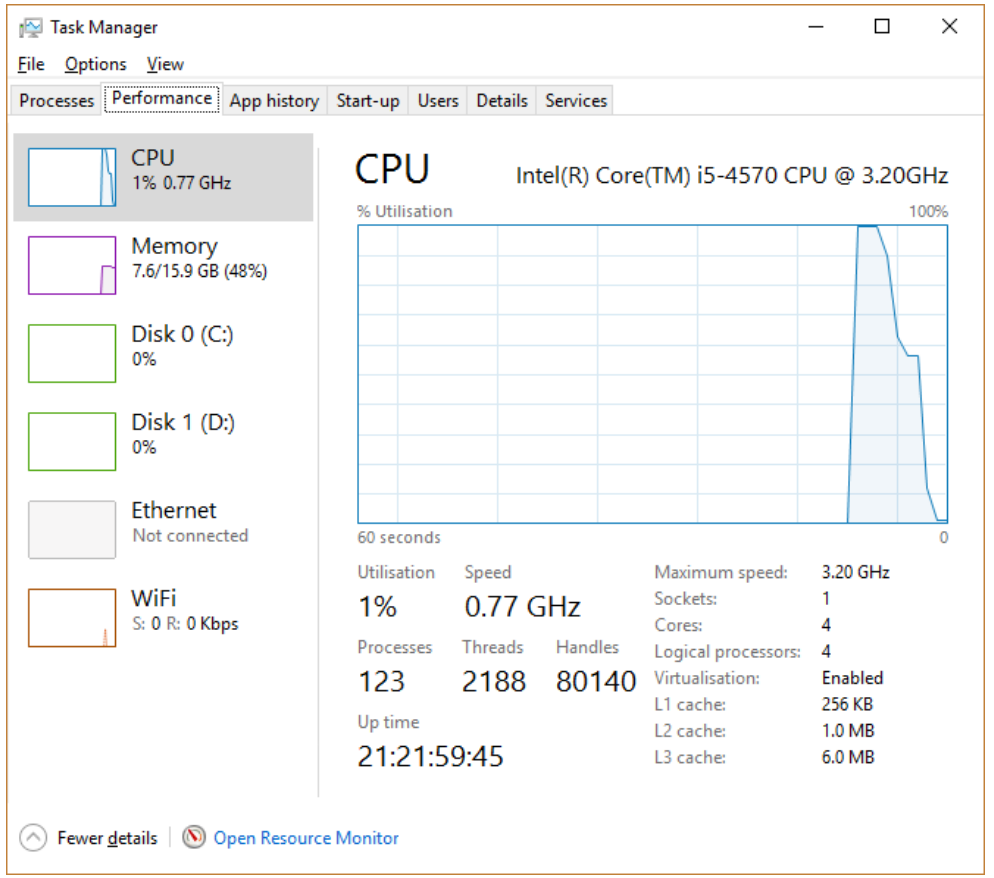


Fibonacci + Fork-Join Pool (JDK 7)

```
public class Fib_FJ extends RecursiveTask<Integer>{
 // with fork-join pool
 int num;
 public Fib_FJ(int num) { this.num=num; }

 @Override
 public Integer compute(){ // a recursive task
  if(num<=1)
   return num;
  Fib_FJ fib1= new Fib_FJ(num-1);
  fib1.fork();
  Fib_FJ fib2= new Fib_FJ(num-2);
  return fib2.compute()+fib1.join();
 }

 public static void main(String[] args) {
  Fib_FJ fib=new Fib_FJ(45);
  ForkJoinPool pool = new ForkJoinPool();
  System.out.println(pool.invoke(fib));
 }
}
```



6 CompletableFuture

Java 8

Since JDK 5 – Callable & Future

```
public String deepThought(){
 // takes a really really long time...
 return "42";
}
```

```
ExecutorService executor=Executors.newCachedThreadPool();

Future<String> f = executor.submit(new Callable<String>() {
        @Override
        public String call() throws Exception {
                return deepThought();
        }
});
```

```
//...
System.out.println(f.get()); // blocks until an answer is given
```

Since JDK 8 – lambda expressions

```
public String deepThought(){
 // takes a really really long time...
 return "42";
}
```

```
ExecutorService executor=Executors.newCachedThreadPool();

Future<String> f = executor.submit( ()-> {
        return deepThought();
});
```

Still, resources are wasted because of the blocking get() call

```
//...
System.out.println(f.get()); // blocks until an answer is given
```

Using CompletableFuture

```
public String deepThought(){
 // takes a really really long time...
 return "42";
}
```

```
ExecutorService executor=Executors.newCachedThreadPool();

// an asynchronous call
CompletableFuture.supplyAsync( ()->{
        return deepThought();
},executor);
```

Using CompletableFuture

```
public String deepThought(){
 // takes a really really long time...
 return "42";
}
```

```
// an asynchronous call
CompletableFuture.supplyAsync( ()->{
        return deepThought();
});
```

Uses the default ForkJoin Pool

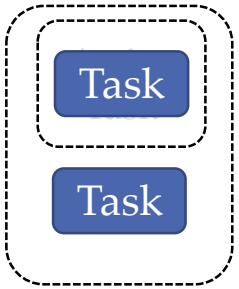
Using CompletableFuture

```
public String deepThought(){
 // takes a really really long time...
 return "42";
}
```

```
CompletableFuture<String> fc = CompletableFuture.supplyAsync( ()->{
        return deepThought();
});

fc.thenAccept( (String answer)->{System.out.println("answer: "+answer);});
```

Reactive pattern: This action will be taken right after deep thought is finished



Using CompletableFuture

```
public String deepThought(){
 // takes a really really long time...
 return "42";
}
```

```
CompletableFuture<String> fc = CompletableFuture.supplyAsync( ()->{
        return deepThought();
});

fc.thenAccept( (String answer)->{System.out.println("answer: "+answer);});
```

Reactive pattern: This action will be taken right after deep thought is finished

Fluent Programming: each method returns its object, allowing chained calls

Returns
CompletableFuture<String>

Using CompletableFuture

```
public String deepThought(){
 // takes a really really long time...
 return "42";
}
```

```
CompletableFuture.supplyAsync( ()->{return deepThought();})
        .thenApply(answer->Integer.parseInt(answer))
        .thenApply(x->x*2)
        .thenAccept(answer->System.out.println("answer: "+answer));
```

Using CompletableFuture

```
public String deepThought(){
 // takes a really really long time...
 return "42";
}
```

```
CompletableFuture.supplyAsync( ()->{return deepThought();},executor)
.thenApply(answer->Integer.parseInt(answer))
.the
```

- thenAccept(Consumer<? super Void> action) : CompletableFuture<Void> - CompletableFuture
- thenAcceptAsync(Consumer<? super Void> action) : CompletableFuture<Void> - CompletableFuture
- thenAcceptAsync(Consumer<? super Void> action, Executor executor) : CompletableFuture<Void> - CompletableFuture
- thenAcceptBoth(CompletionStage<? extends U> other, BiConsumer<? super Void,? super U> action) : CompletableFuture<Void> - CompletableFuture
- thenAcceptBothAsync(CompletionStage<? extends U> other, BiConsumer<? super Void,? super U> action) : CompletableFuture<Void> - CompletableFuture
- thenAcceptBothAsync(CompletionStage<? extends U> other, BiConsumer<? super Void,? super U> action, Executor executor) : CompletableFuture<Void> - CompletableFuture
- thenApply(Function<? super Void,? extends U> fn) : CompletableFuture<U> - CompletableFuture
- thenApplyAsync(Function<? super Void,? extends U> fn) : CompletableFuture<U> - CompletableFuture
- thenApplyAsync(Function<? super Void,? extends U> fn, Executor executor) : CompletableFuture<U> - CompletableFuture
- thenCombine(CompletionStage<? extends U> other, BiFunction<? super Void,? super U,? extends V> fn) : CompletableFuture<V> - CompletableFuture
- thenCombineAsync(CompletionStage<? extends U> other, BiFunction<? super Void,? super U,? extends V> fn) : CompletableFuture<V> - CompletableFuture
- thenCombineAsync(CompletionStage<? extends U> other, BiFunction<? super Void,? super U,? extends V> fn, Executor executor) : CompletableFuture<V> - CompletableFuture
- thenCompose(Function<? super Void,? extends CompletionStage<U>> fn) : CompletableFuture<U> - CompletableFuture
- thenComposeAsync(Function<? super Void,? extends CompletionStage<U>> fn) : CompletableFuture<U> - CompletableFuture
- thenComposeAsync(Function<? super Void,? extends CompletionStage<U>> fn, Executor executor) : CompletableFuture<U> - CompletableFuture
- thenRun(Runnable action) : CompletableFuture<Void> - CompletableFuture
- thenRunAsync(Runnable action) : CompletableFuture<Void> - CompletableFuture

Exercise

- Write an Active Object using the fork-join pool

Please look at

- New Concurrency Utilities in Java 8
  - https://www.youtube.com/watch?v=Q_0_1mKTlnY
- How to use CompletableFuture
  - https://www.youtube.com/watch?v=HdnHmbFg_hw
- Reactive Programming patterns
  - https://www.youtube.com/watch?v=tiJEL3oiHIY
- Disruptor Pattern
  - https://www.youtube.com/watch?v=DCdGlxBbKU4
  - https://disruptor.googlecode.com/files/Disruptor-1.0.pdf