Chapter 8
Applying Thread Pools

Magnus Andersson

Execution policies

- Not all task are suitable for all execution policies
  - Dependent task
  - Task exploiting thread confinement
  - Response time sensitive tasks
  - ThreadLocal tasks

Starvation deadlock

- Simplest example of deadlock:
  - Single-threaded executor
  - Task **A** submits a new task **B**, which **A** depends on
  - Deadlock!
  - Easy to extrapolate to a concurrent executor
    - Make sure that your pool size is large enough

Sizing the thread pool

- Don’t hard code!
- The black art of thread pool sizing
  - Is your program **compute intensive**?
    - $N_{cpu} + 1$
  - Does your program use a fair amount of **blocking, I/O**, etc?
    - $N_{cpu}\ U_{cpu}\ (1 + \frac{W}{C})$

The ThreadPoolExecutor constuctor

```
public ThreadPoolExecutor(
        int corePoolSize,
        int maximumPoolSize,
        long keepAliveTime,
        TimeUnit unit,
        BlockingQueue<Runnable>
                workQueue,
        ThreadFactory
                threadFactory,
        RejectedExecutionHandler
                handler
) { … }
```

The ThreadPoolExecutor constuctor

```
public ThreadPoolExecutor(
        int corePoolSize,
        int maximumPoolSize,
        long keepAliveTime,
        TimeUnit unit,
        BlockingQueue<Runnable>
                workQueue,
        ThreadFactory
                threadFactory,
        RejectedExecutionHandler
                handler
) { … }
```

# of desired threads

Upper bound on # of threads

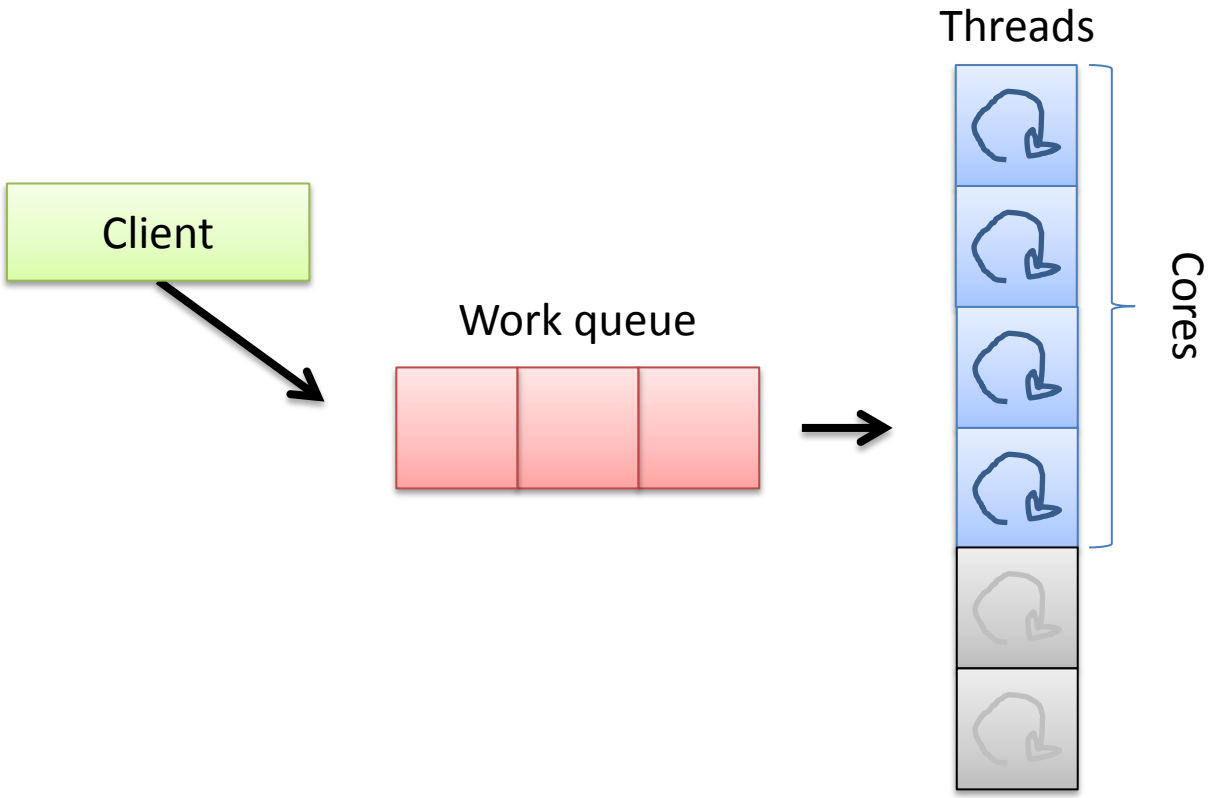
How long idle threads are kept around

`keepAliveTime` time unit (?)

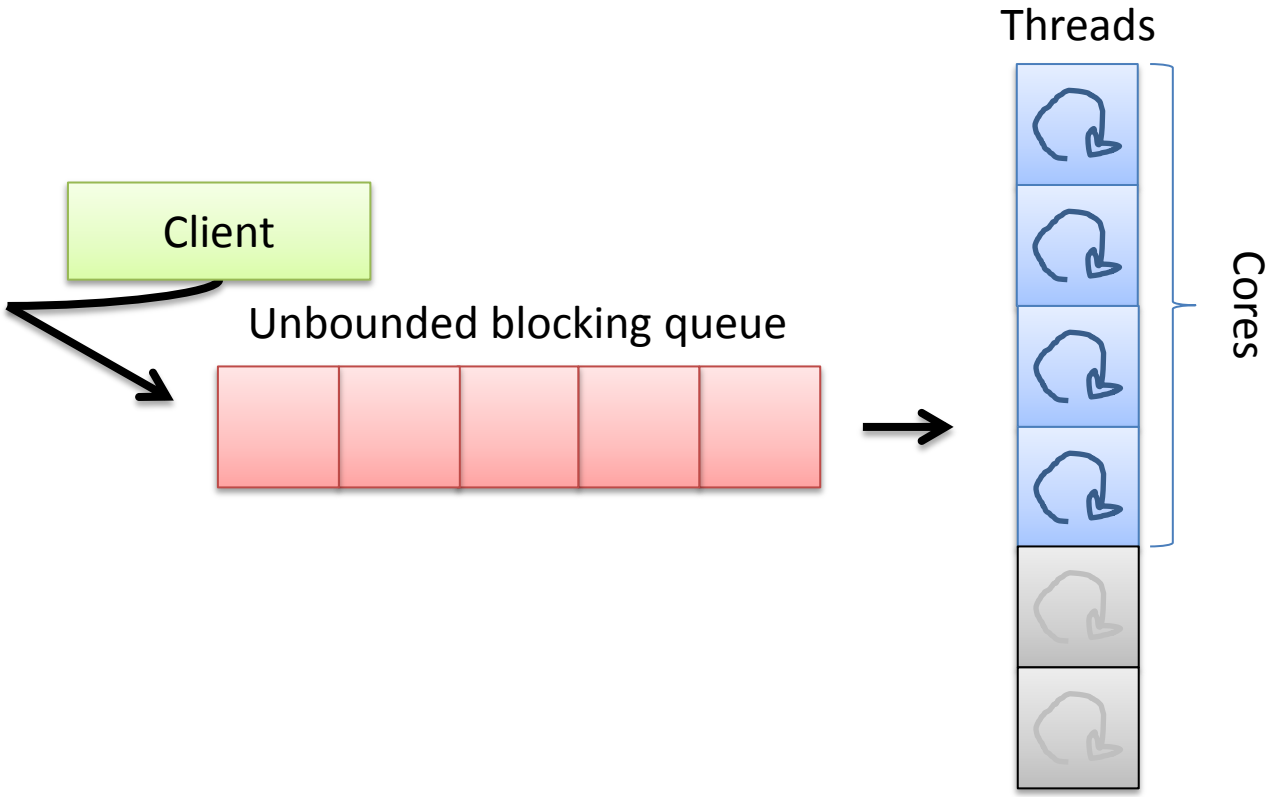
The ThreadPoolExecutor constuctor

```
public ThreadPoolExecutor(
        int corePoolSize,
        int maximumPoolSize,
        long keepAliveTime,
        TimeUnit unit,
        BlockingQueue<Runnable>
                workQueue,
        ThreadFactory
                threadFactory,
        RejectedExecutionHandler
                handler
) { … }
```

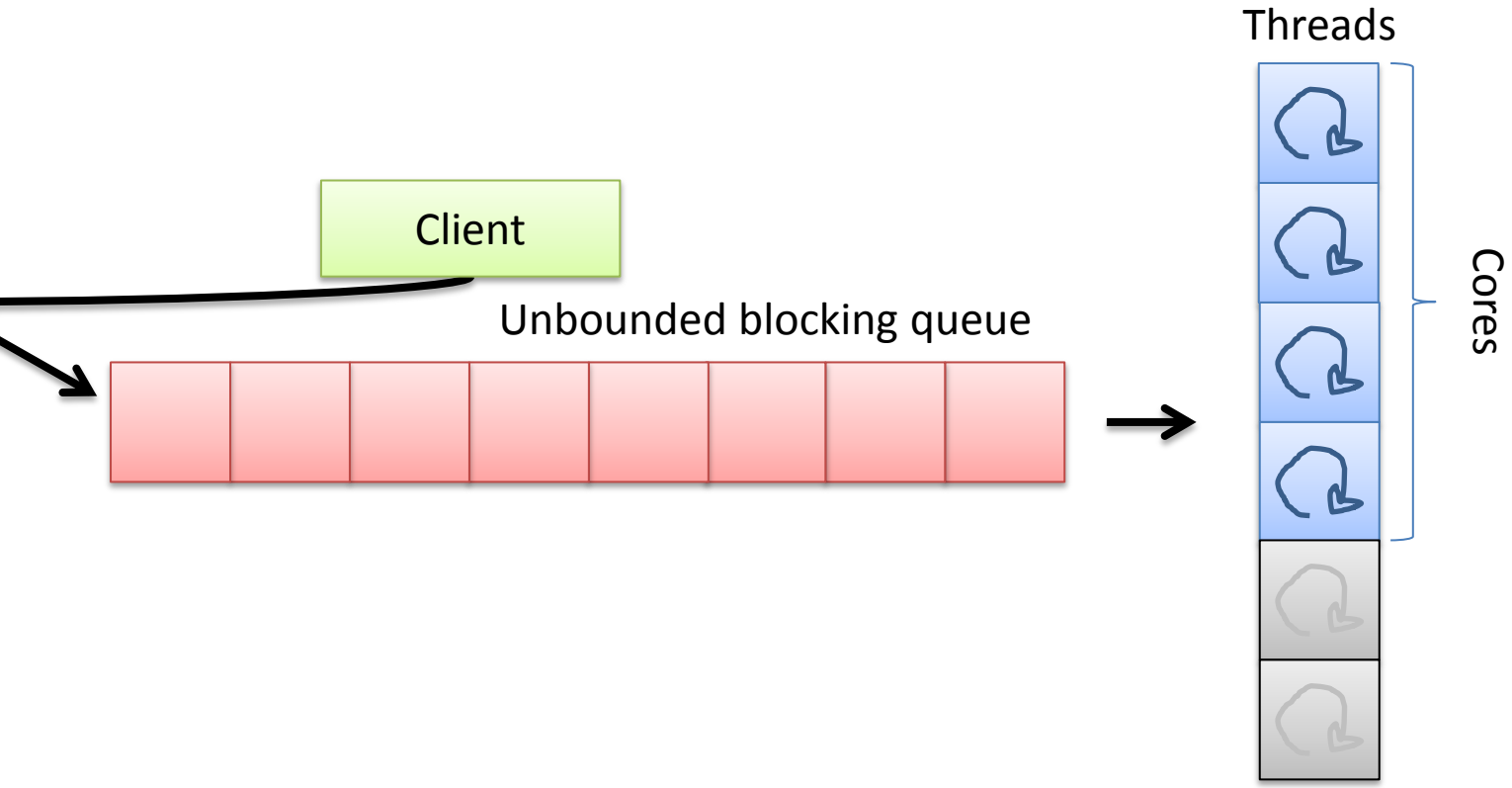
The work queue



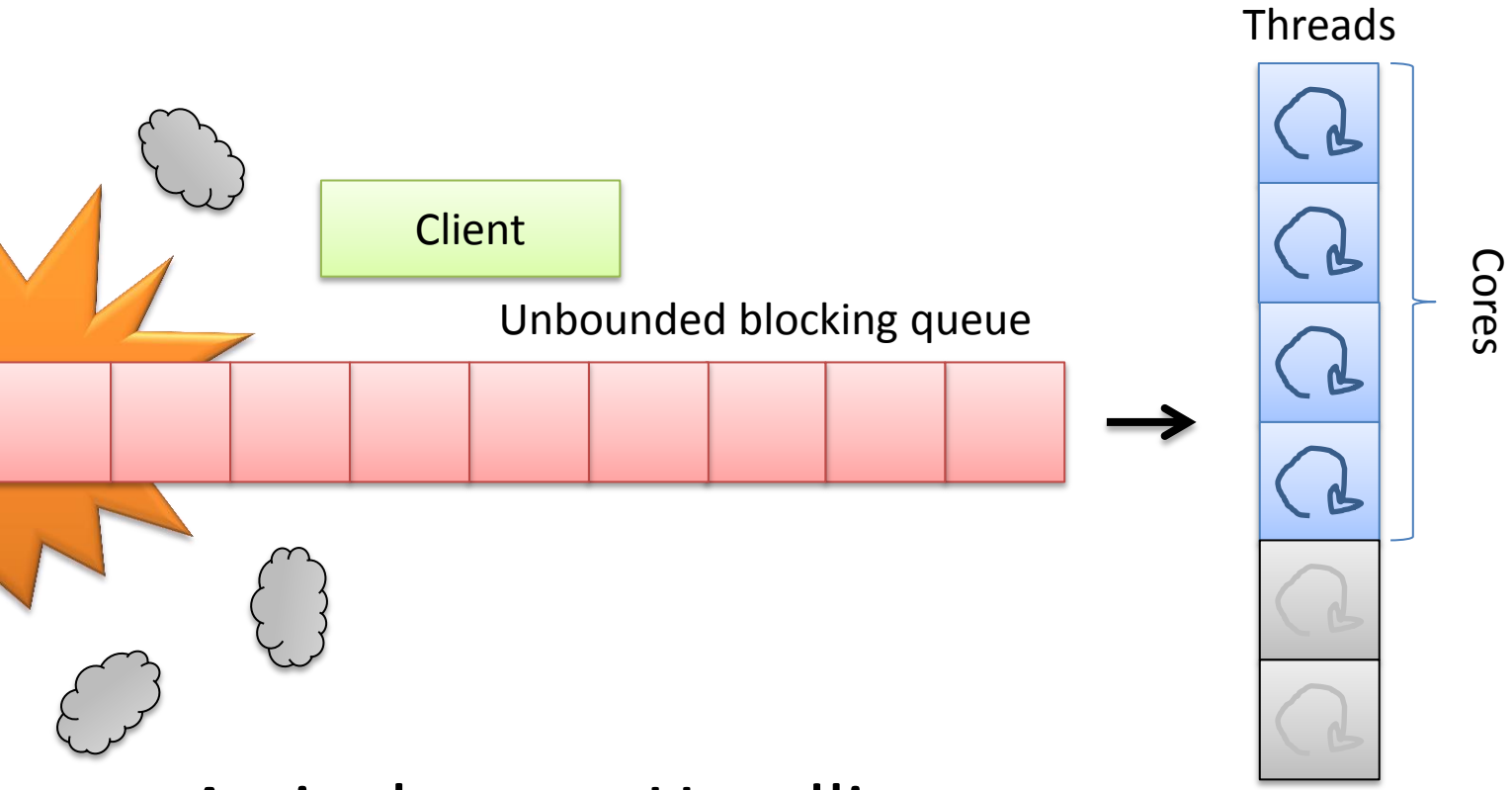
The work queue
Unbounded blocking queue



The work queue
Unbounded blocking queue

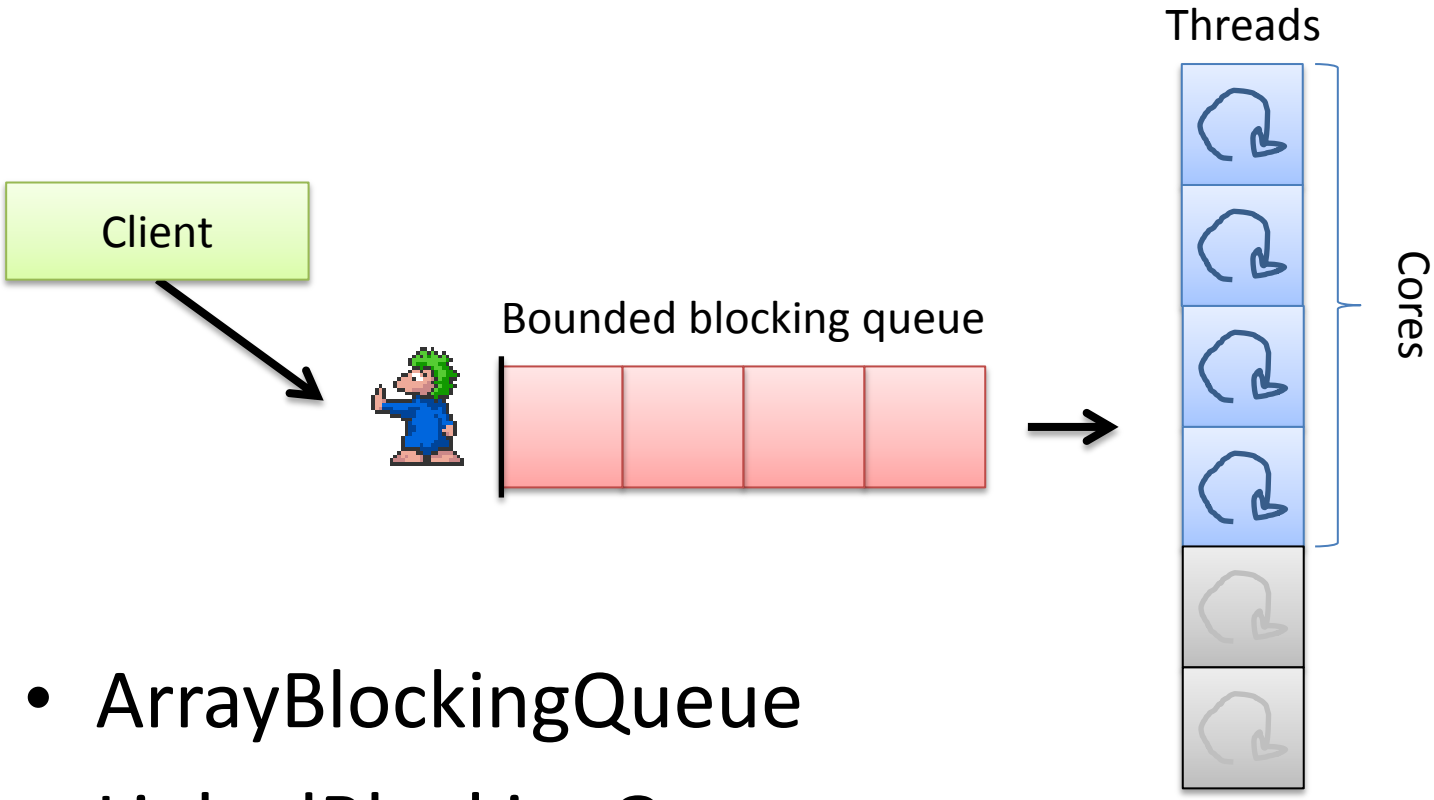


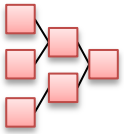
The work queue
Unbounded blocking queue



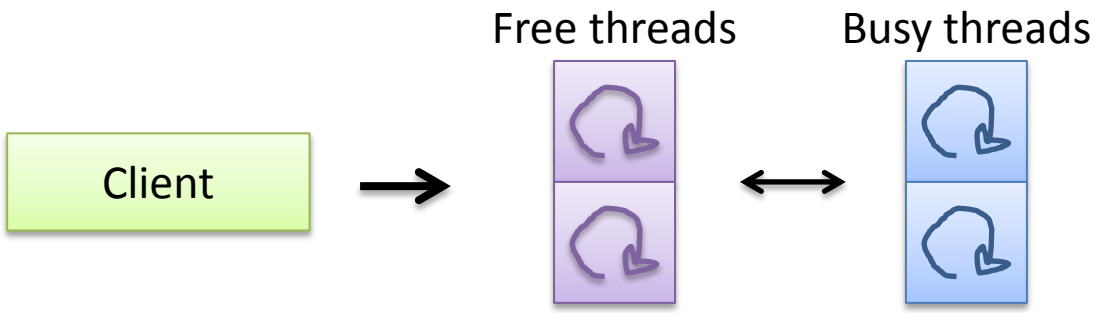
- Arrival rate > Handling rate

The work queue
Bounded blocking queue



- ArrayBlockingQueue
- LinkedBlockingQueue
- PriorityBlockingQueue 

The work queue
Synchronous “queue”

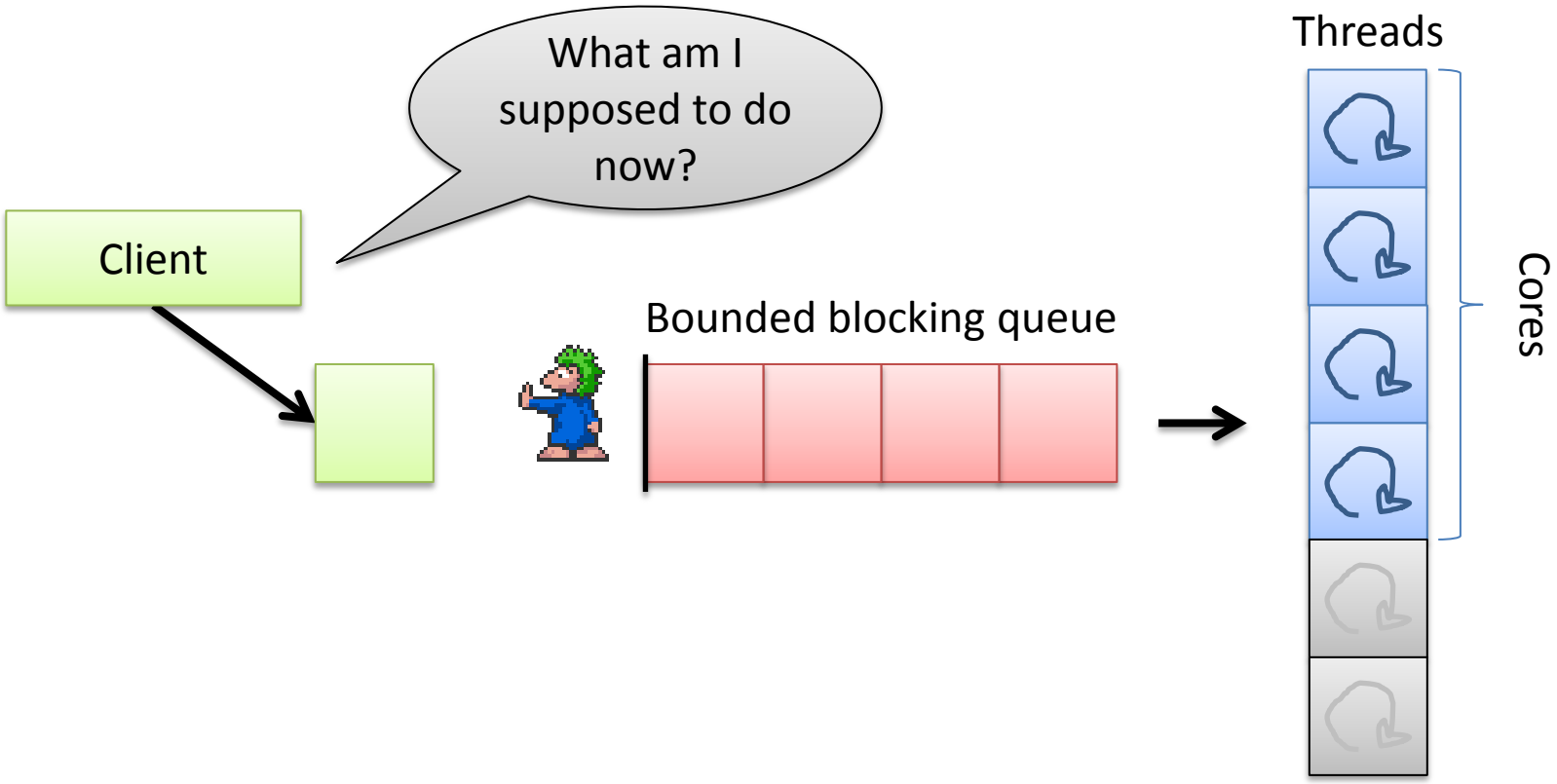


- Direct hand-off to the thread that will handle the task

The ThreadPoolExecutor constuctor

```
public ThreadPoolExecutor(
        int corePoolSize,
        int maximumPoolSize,
        long keepAliveTime,
        TimeUnit unit,
        BlockingQueue<Runnable>
                workQueue,
        ThreadFactory
                threadFactory,
        RejectedExecutionHandler
                handler
) { … }
```

Saturation policy



Saturation policy

- Abort
  - Throws a `RejectedExecutionException`
- Discard
  - Silently kill submitted task
- Discard-oldest
  - Silently kill oldest task
- Caller-runs
  - Push work task back to client
    - Effectively slows down submission rate since client will be busy for a while and can't (shouldn't) submit new tasks

Blocking execute

The fifth Beatle

- Doesn’t exist
- Blocks caller if the queue is full
- Easily implemented using semaphores

```
public void submitTask(final Runnable command) throws InterruptedException {
        semaphore.aquire();
        try {
                exec.execute(new Runnable() {
                        public void run() {
                                try {
                                        command.run();
                                } finally {
                                        semaphore.release();
                                }
                        }
                });
        } catch (RejectedExecutionException e) {
                semaphore.release();
        }
}
```

The ThreadPoolExecutor constuctor

```
public ThreadPoolExecutor(
        int corePoolSize,
        int maximumPoolSize,
        long keepAliveTime,
        TimeUnit unit,
        BlockingQueue<Runnable>
                workQueue,
        ThreadFactory
                threadFactory,
        RejectedExecutionHandler
                handler
) { … }
```

Thread factory

- One method: **`newThread`**
- Configure thread pool threads
- Naming, logging, exception handling

```
public class MyThreadFactory implements ThreadFactory {
        private final String poolName;

        public MyThreadFactory(String poolName) {
                this.poolName = poolName;
        }
        public Thread newThread(Runnable runnable) {
                return new MyAppThread(runnable, poolName);
        }
}
// See MyAppThread listing in book
```

Post-construction modification

- Possible, but could be dangerous
  - Single-threaded executor with increased pool size?
  - Pool size not enough to handle dependent tasks?
- If this is a problem: encapsulate the Executor

Additional ThreadPoolExecutor hooks

- Discriptive names:
  - beforeExecute
  - afterExecute
  - terminated
    - Runs at the very end
- Logging, statistics, etc…
- It’s OK to use ThreadLocal between beforeExecute and afterExecute

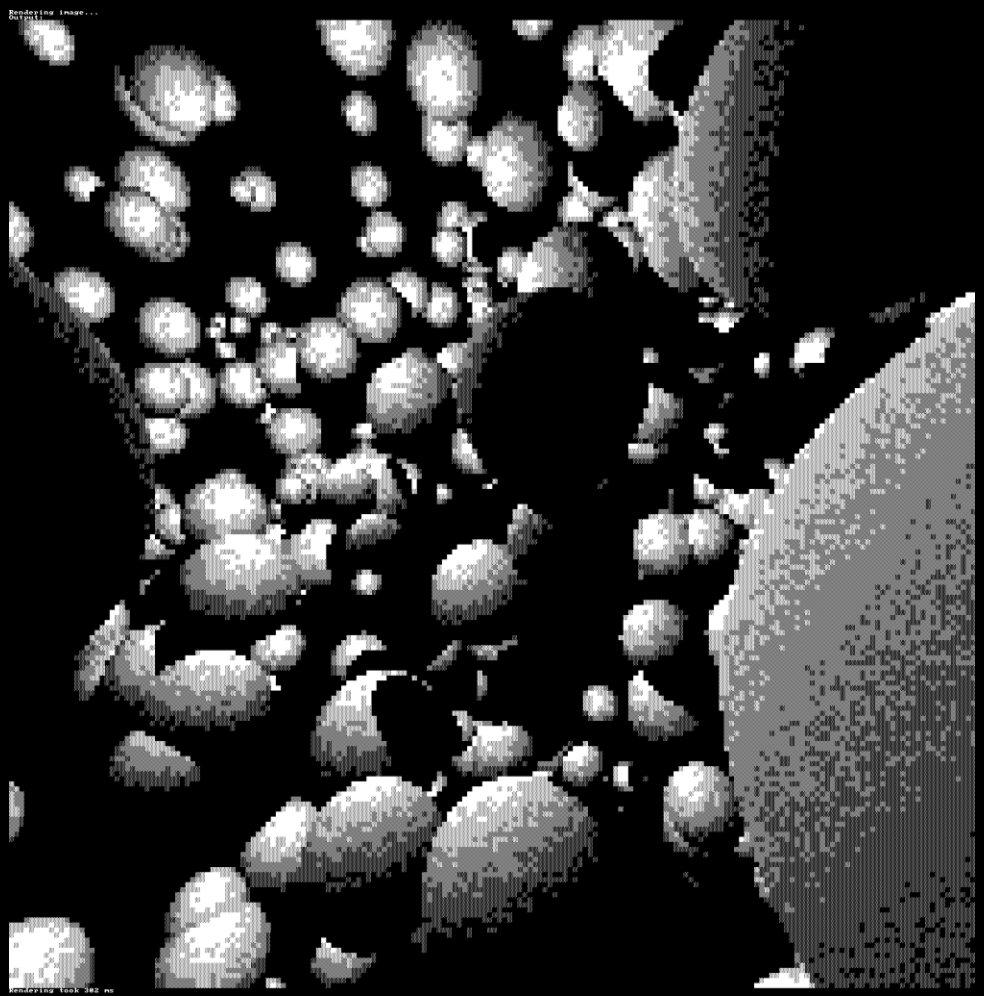
Parallelizing algorithms

```
void processSequentially(List<Element> elements) {
        for (Element e : elements)
                process(e)
}

void processInParallel(Executor exec, List<Elements> elements) {
        for (final Element e : elements) {
                exec.execute(new Runnable() {
                        public void run() { process(e); }
                });
        }
}
```

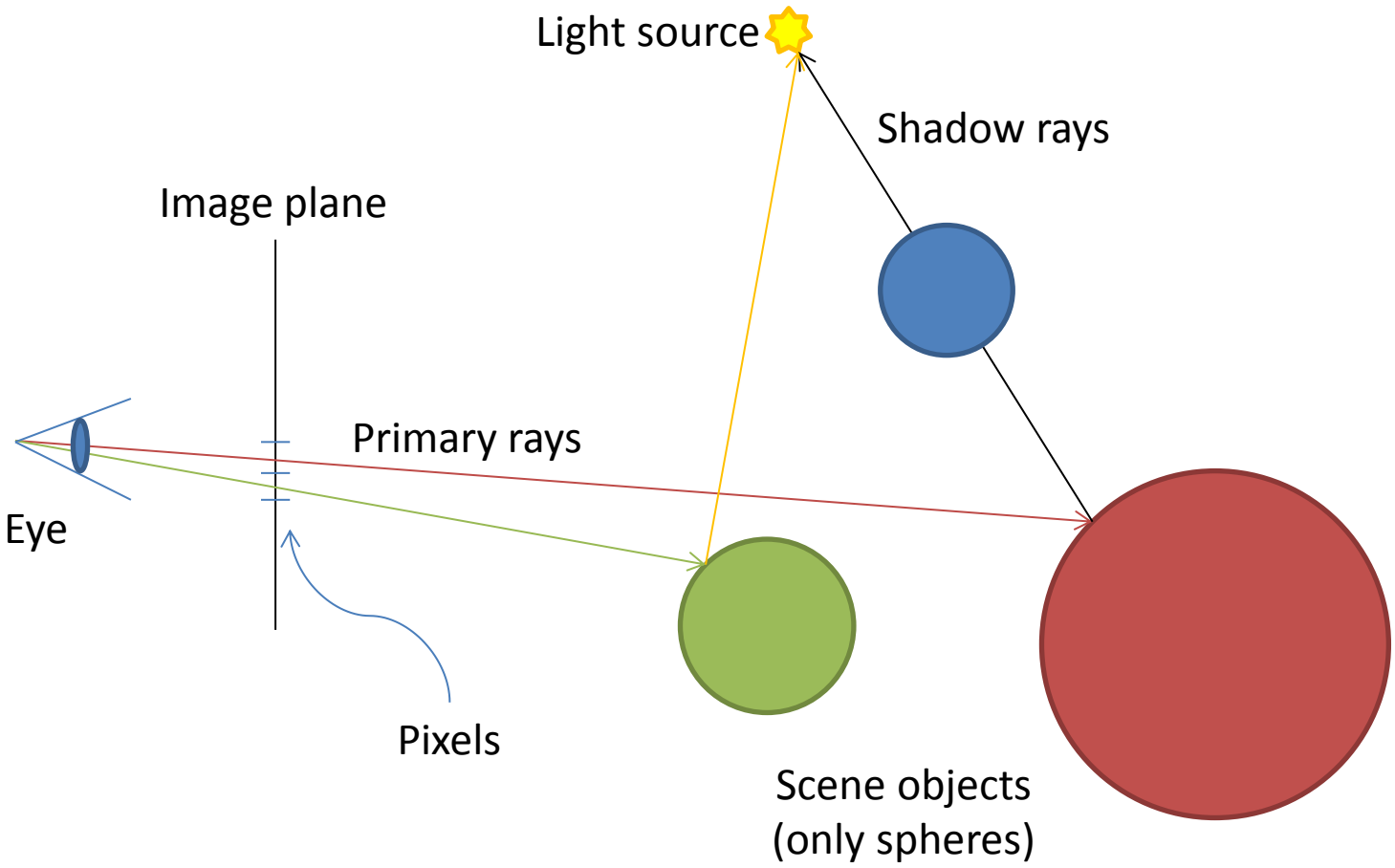
Parallelizing recursive algorithms

```
public<T> void sequentialRecursive(List<Node<T>> nodes,
                Collection<T> results)
{
        for (Node<T> n : nodes) {
                results.add(n.compute());
                sequentialRecursive(n.getChildren(), results);
        }
}

public<T> void parallelRecursive(final Executor exec,
                List<Node<T>> nodes, final Collection<T> results)
{
        for (Node<T> n : nodes) {
                exec.execute(new Runnable() {
                        public void run() { results.add(n.compute()); }
                });
                parallelRecursive(n.getChildren(), results);
        }
}
```

Exercise
Ray tracing



Exercise


Light source
Shadow rays
Image plane
Primary rays
Eye
Pixels
Scene objects
(only spheres)

Exercise

- Ray tracing – “embarrassingly parallel”
- Each pixel is its own task
  - Setup primary ray
  - Test intersection with all scene objects
  - Use closest hit as pixel color
  - If hit:
    - Setup shadow ray
    - Test intersection with all scene objects
    - Make pixel dark if anything is hit
- All of this is already done

Exercise

- Your task:

  1. **`traceSerial()`** serially loops over all screen pixels and performs ray tracing. Create a parallel version in **`traceParallel()`** which results in the same output image. You must configure your own **`ThreadPoolExecutor`** using the constructor (and not the factory), and use it to dispatch your tasks. Tweak it to gain maximum performance.

  You can switch algorithms with the **`private static final boolean serial`** flag at the top of RayTracer.java. No changes to GFX.java should be necessary.

  2. Gather statistics using **`beforeExecute()`** and **`afterExecute()`** on what the average and the maximum task run times were. Print out the result in **`terminate()`**.

- Optional:

  Separate the shadow rays to separate tasks to be re-inserted in to the work queue. The maximum thread count and queue size may only be **40** or less each.

  Although you're not likely to see a speedup from this, it is an interesting concurrency problem to make sure that the output image is still correct (while not starving any threads to death).

Exercise

- Alternatively:

  If you have your own idea of a problem that could be parallelized using thread pools, you are welcome to do that instead. The same criteria apply – you may not use the factory-method to create a ThreadPoolExecutor, but must use the constructor. You must also experiment and tweak the settings to get good overall performance.

  You'll also need to gather some statistics during the execution. Average and maximum task running times, for example.

Fin

[BACKUP]
Sizing the thread pool

$$N_{cpu}\, U_{cpu}\, (1 + \tfrac{W}{C})$$

N_{cpu} = `Runtime.getRuntime().availableProcessors()`

U_{cpu} = target utilization [0, 1]

$\frac{W}{C}$ = Wait to compute time ratio

- Example:
  - 8 processors targeting 0.5 utilization, with a profiled wait time of 2 and compute time of 5:
  - Threads = 8 * 0.5 * (1 + 2/5) = 5.6

[BACKUP] Standard ThreadPoolExecutors

- Executor factory:
  - **`newCachedThreadPool`**
    - SynchronousQueue
    - corePoolSize = 0. maximumPoolSize = INF. Timeout = 1 min
  - **`newFixedThreadPool`**
    - Unbounded LinkedBlockingQueue
    - corePoolSize = maximumPoolSize
  - **`newSingleThreadExecutor`**
    - Unbounded LinkedBlockingQueue
    - corePoolSize = maximumPoolSize = 1
  - **`newScheduledThreadExecutor`**
    - DelayQueue
    - corePoolSize = maximumPoolSize = 1

[BACKUP]
The ThreadPoolExecutor constuctor

```
public ThreadPoolExecutor(
        int corePoolSize,
        int maximumPoolSize,
        long keepAliveTime,
        TimeUnit unit,
        BlockingQueue<Runnable> workQueue,
        ThreadFactory threadFactory,
        RejectedExecutionHandler handler)
```