Concurrent Programming (Part II)
Lecture 14: Multi-Threaded Socket-Based Servers and Thread Pools

Dr Kevin Bryson

K.Bryson@cs.ucl.ac.uk

Room 8.04

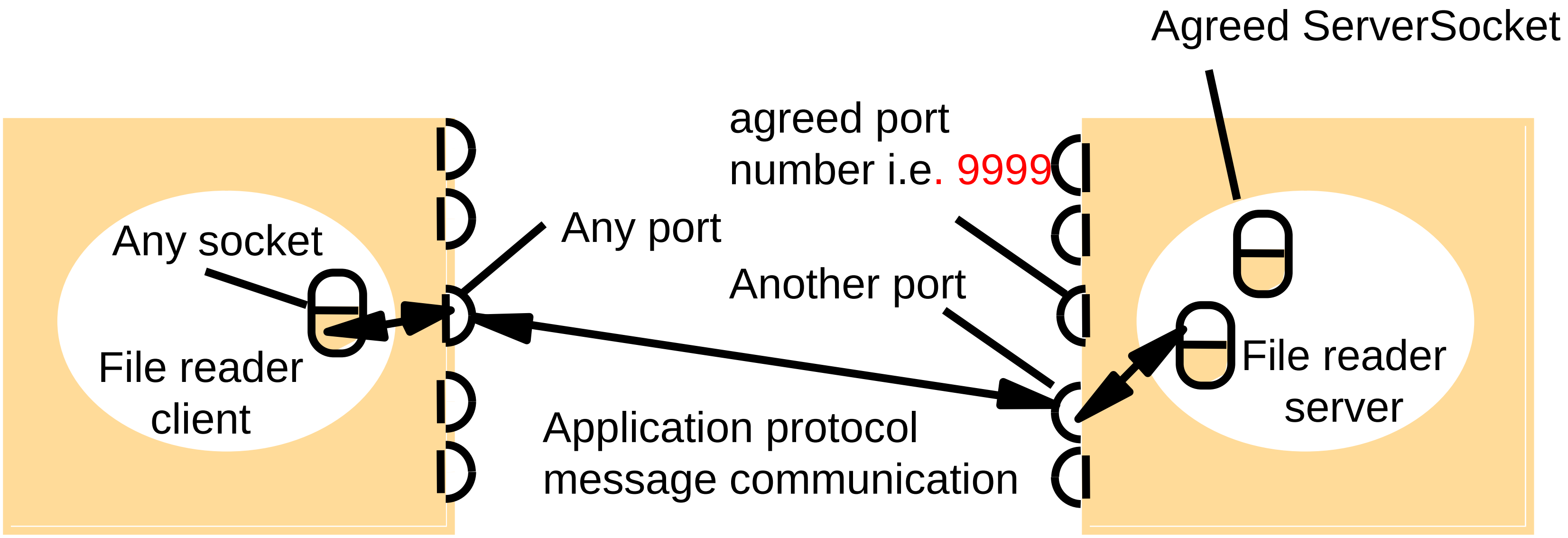
Course Web Site on Moodle
http://moodle.ucl.ac.uk/course/view.php?id=753
Enrolment Key: ATOMIC

Overview

- In the last lecture we looked at how to construct a client/server architecture using sockets.
- This was a single-threaded server, it could only service one client after another client (although buffering of the socket streams gave the illusion it may be multi-threaded …)
- In this lecture we will examine how to make the server multi-threaded and how server response time is affected.

- Key Question: What factors affect server performance ?
- This question will lead us into a discussion of Thread Pools.

Recall the File Reader Server Demo



- Recall that the FileReaderServer is single threaded.
- The single ‘Main’ thread blocks waiting for a request at the ServerSocket accept() method.
- It accepts a client connection and continues to process it.
- Only when it is finished will it accept another connection.

Why would we want to make the server multi-threaded? Discuss …

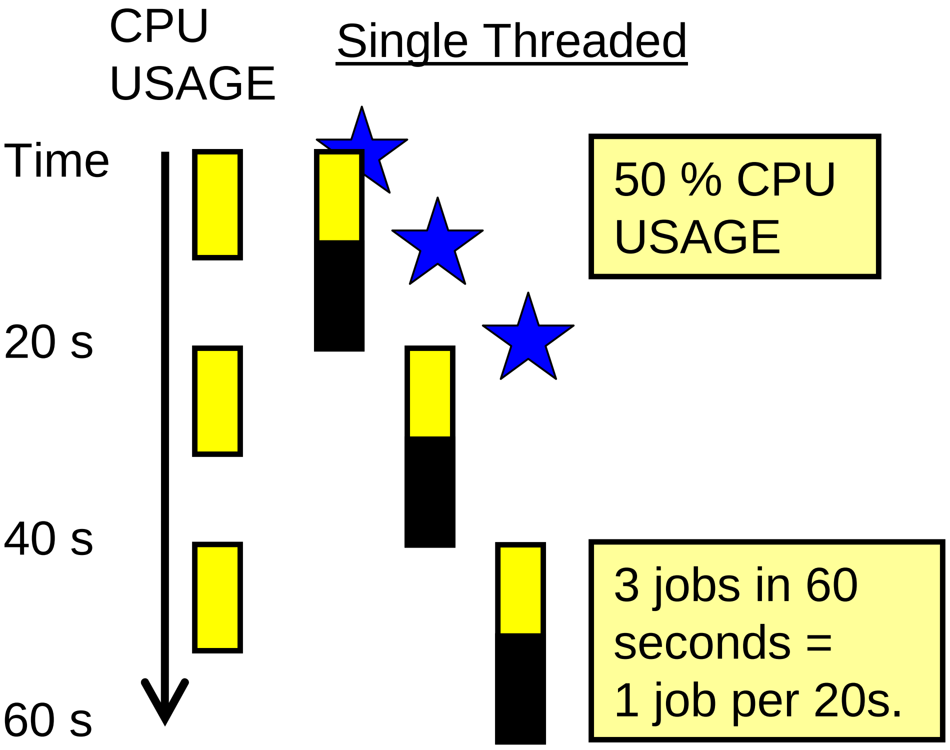
- If tasks **block** (for instance on I/O), even a single processor will not be optimally used (since it will be idle when the Thread blocks).
- If the server has multiple processors – these may not be fully utilized executing a single task at a time
- In these cases the **throughput** will be lower than ideal (i.e. the average number of jobs processed per unit time)

- ***Making the server multi-threaded can increase processor utilization and hence throughput (although not always!)***

Tasks blocking on IO - increasing throughput

Consider this scenario:

- Single CPU with requests taking 20 seconds to run sequentially – this consists of 10 seconds of CPU (🟨) and 10 seconds (■) waiting for disk (i.e. blocking)
- Three requests appear 10 seconds apart (indicated by ★)



CPU USAGE

Single Threaded

Time

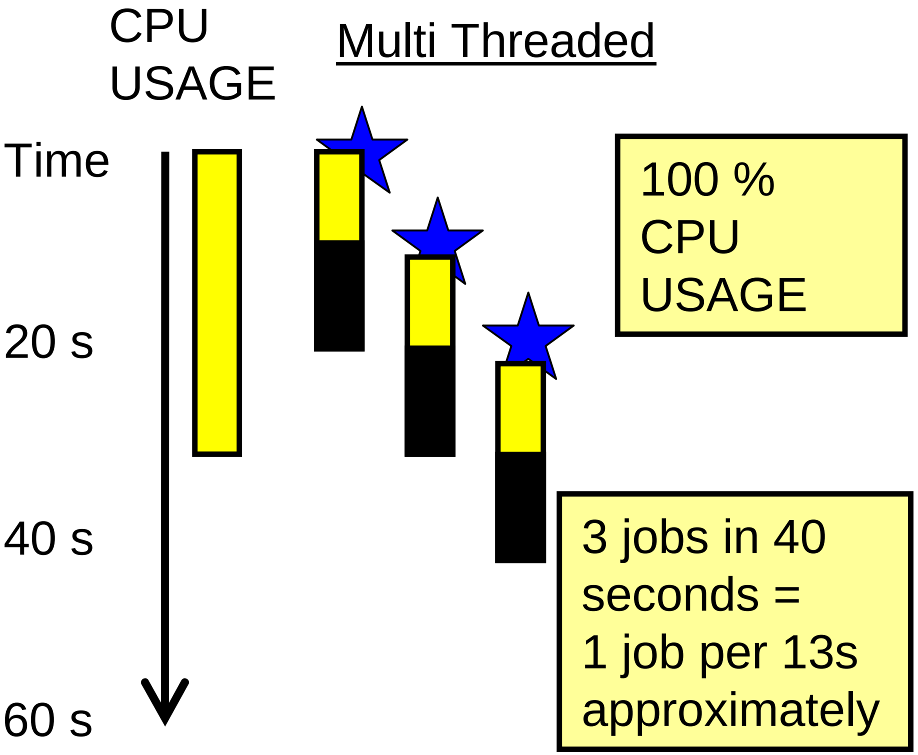
20 s

40 s

60 s

50 % CPU USAGE

3 jobs in 60 seconds = 1 job per 20s.



CPU USAGE

Multi Threaded

Time

20 s

40 s

60 s

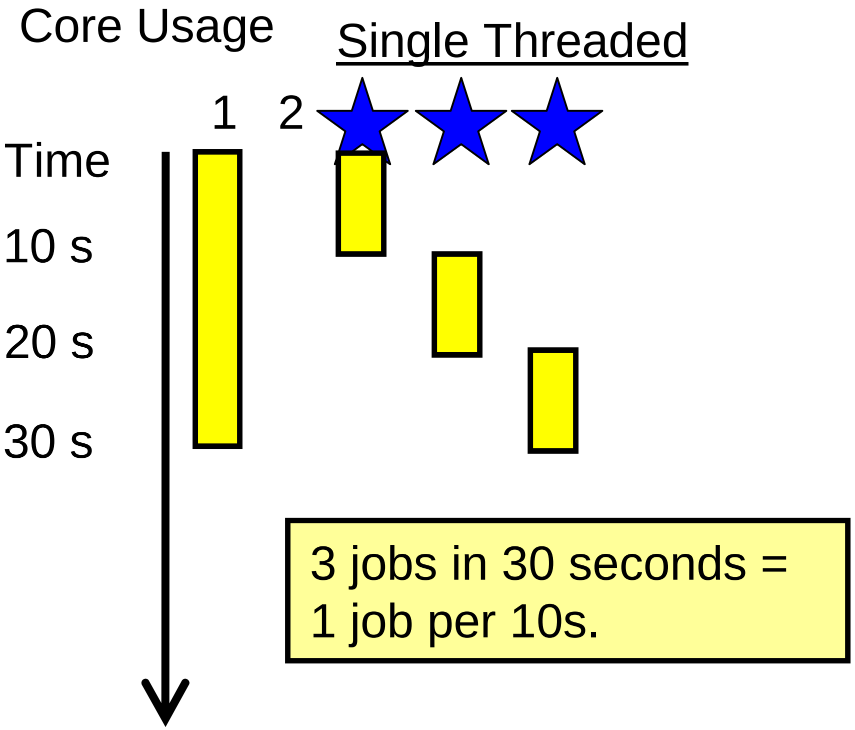
100 % CPU USAGE

3 jobs in 40 seconds = 1 job per 13s approximately

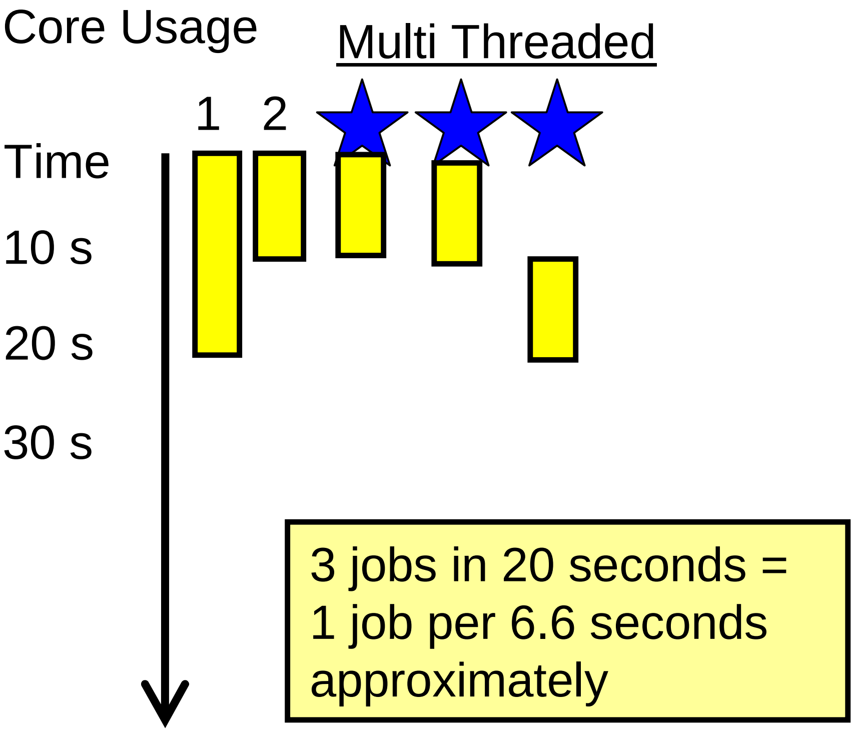
Multiple CPUs/cores - increasing throughput

Consider this scenario:

- Dual-core CPU with requests taking 10 seconds to run sequentially – consisting entirely of CPU time (🟨)
- Three requests appear at the same time (indicated by ★)
- Ideal case - usually there are additional overheads.



3 jobs in 30 seconds = 1 job per 10s.



3 jobs in 20 seconds = 1 job per 6.6 seconds approximately

Implementing a Multi-Threaded Server

- The multi-threaded File Reader Server **functionally** does exactly the same as the single-threaded File Reader Server – it sends back the contents of a specified file given a filename using an identical protocol.
- But its **non-functional** behaviour in terms of response times under load is very different.

- A key implementation difference is that it now starts a new Thread for every client request. This ‘ServiceThread’ actually carries out the required task.
- I have also handed out a ‘TimedClient’ listing which is identical to the previous listing except it also returns the time it takes for the server to respond.
- We will read through the code and discuss the changes.

Comparison between the single-threaded and multi-threaded servers

- We will run the single-threaded server with 3 simultaneous requests.

- Write down the response times taken here …
- Based on the previous slides – what would you expect the 3 response times to be in the case of the multi-threaded server?
- Write down you predictions here …

Are the response times as you predicted?

- We will now run the 3 simultaneous requests using the multi-threaded server …

- Even though the tasks are not CPU bound we do not get the expected throughput if we assume the CPU is the main resource.

- Can you think of any reasons for this?
- How would you test these hypotheses?

Thread per Request Problems …

So what’s wrong with a Thread per request ?
Not ideal (in some cases) because:

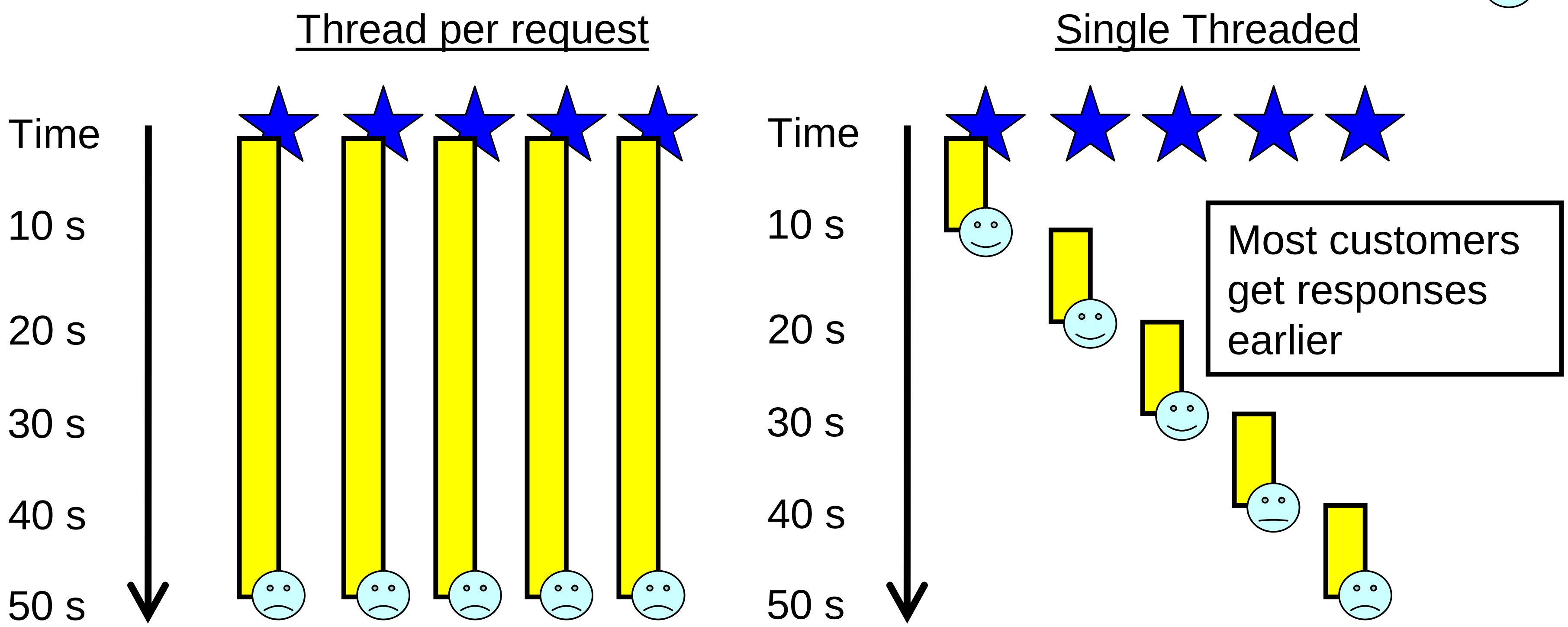
- As load on server increases the number of Threads created is unbounded … resource allocation problems.
- Limited resources include CPU, memory, disk space, disk IO, network IO, OS limits on number of threads, etc.

- **Perceived throughput** is very poor for a heavily loaded system with a Thread per request.
- CPU-bound processes will get smaller and smaller time slices on the CPU as the number of Threads increase – leading to excessively long delays for ***most threads***.
- Similar situation for other machine resources …

Perceived Throughput Scenario

Consider this scenario:

- Requests take 10 seconds of CPU and they are completely CPU bound (i.e. no I/O required). ☺
- Five requests appear at about the same time – happy user? ☹



Most customers get responses earlier

A Thread Pool Helps Server Optimization

- Creating a Thread is not a trivial operation and uses CPU cycles. Each thread requires resources and having an unbounded number of Threads may result in resource exceptions such as OutOfMemory and inefficient use of resources.

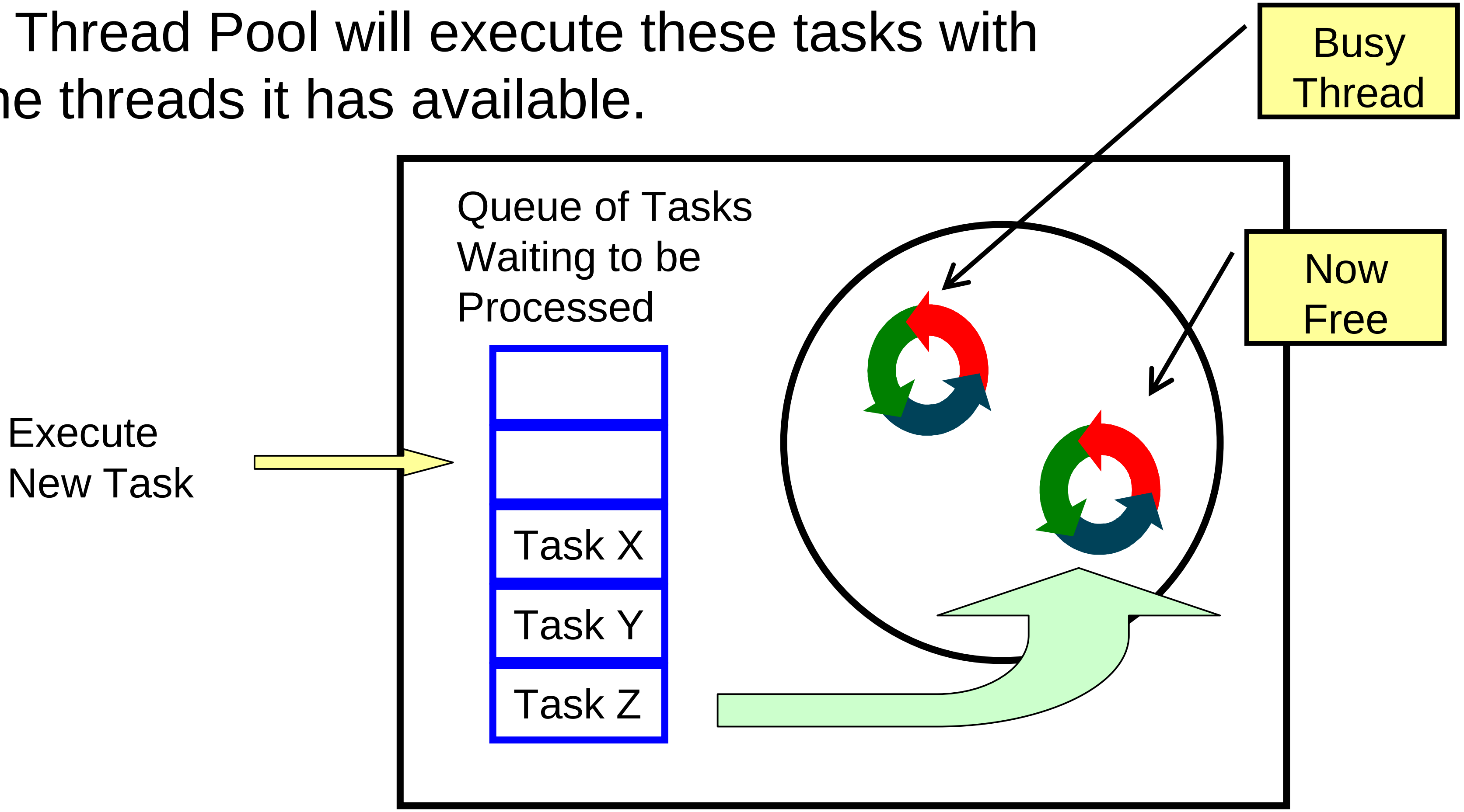
- A Thead Pool consist of a pool of Threads that can be reused.
- The number of Threads within the pool can dynamically adjust **within set limits** to accommodate the current load.
- Thread Pool parameter optimization is complex and generally involves experimentation.

A Thread Pool Helps Server Optimization

Schematic of a Thread Pool with two Threads available

A set of tasks is assigned to the Thread Pool

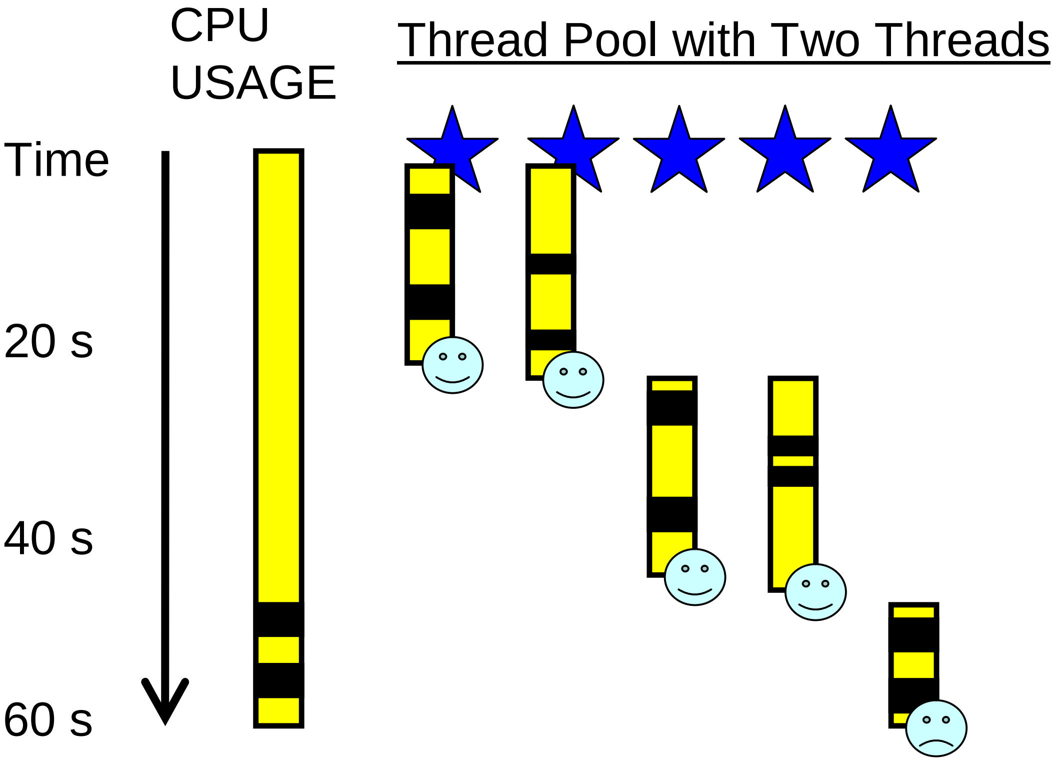
The Thread Pool will execute these tasks with the threads it has available.



A Thread Pool allows optimization between different concerns such as CPU utilization, IO resources, throughput, etc.

Consists of a bounded number of available Threads (sometimes a fixed number) which are reused.

If we have a Pool of 2 Threads in the previous scenario.



Java 5 Executor Framework

- Fortunately Java 5 now has a standard framework for uncoupling the concerns about how an overall system creates ‘tasks’ (for instance Runnables) and how it goes about executing these tasks.
- Classes that can execute ‘tasks’ implement the Executor interface.
- This allows a new type of Executor to be substituted into a system by only changing a single line of code (where it creates the concrete instance of the executor) … rather than all the places that submit different types of ‘tasks’.
- This ‘task’ decomposition of a concurrent system is a particularly powerful architecture.

```
package java.util.concurrent;

public interface Executor {
      void execute(Runnable command);
}
```

Java 5 Thread Pool Implementation

- Within Java 5, a Thread Pool is a type of executor and can execute Runnable tasks.

```
Executor pool = new ThreadPoolExecutor(
                    10,         // Core Pool Size = 10 Threads.
                    20,         // Max Pool Size = 20 Threads.
                    50000L,     // Thread idle ‘timeout’ = 50 seconds.
                    TimeUnit.MILLISECONDS,
                    new LinkedBlockingQueue<Runnable>());
```

- A task (implementing the interface `Runnable`) can be assigned to the pool using:

```
Runnable task = … some Runnable object …
pool.execute(task);
```

Summary

- Optimization of multithreaded applications
  - Why a single threaded solution may not be ideal in some cases.
  - Why a thread-per-task solution may not be ideal in some cases.
  - The Thread Pool solution.

- A very brief introduction to the Java 5 Executor interface and Thread Pool class.