Network Applications: High-performance Server Design

Qiao Xiang

https://qiaoxiang.me/courses/cnns-xmuf21/index.shtml

10/21/2021

This deck of slides are heavily based on CPSC 433/533 at Yale University, by courtesy of Dr. Y. Richard Yang.

Outline

- Admin and recap
- High-performance network server design

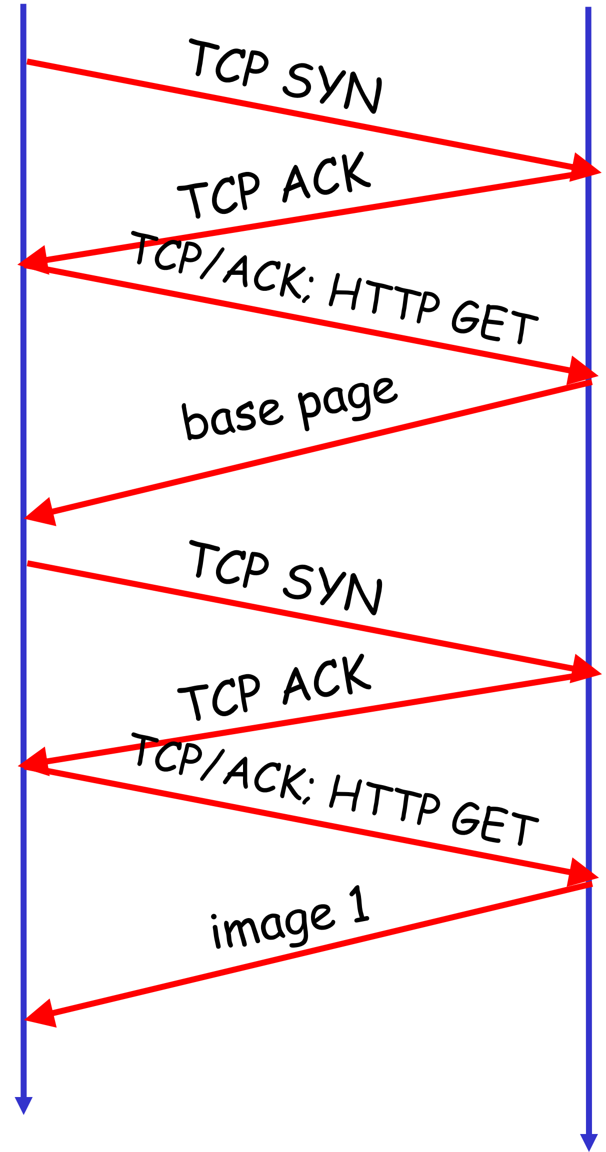
Admin

- Lab Assignment Three
  - Part 1: Due Nov. 11
  - Part 2: To be posted

- Exam 1 date?

Recap: Latency of Basic HTTP/1.0

- >= 2 RTTs per object:
  - TCP handshake --- 1 RTT
  - client request and server responds --- at least 1 RTT (if object can be contained in one packet)



Recap: Substantial Efforts to Speedup HTTP/1.0

- Reduce the number of objects fetched [Browser cache]

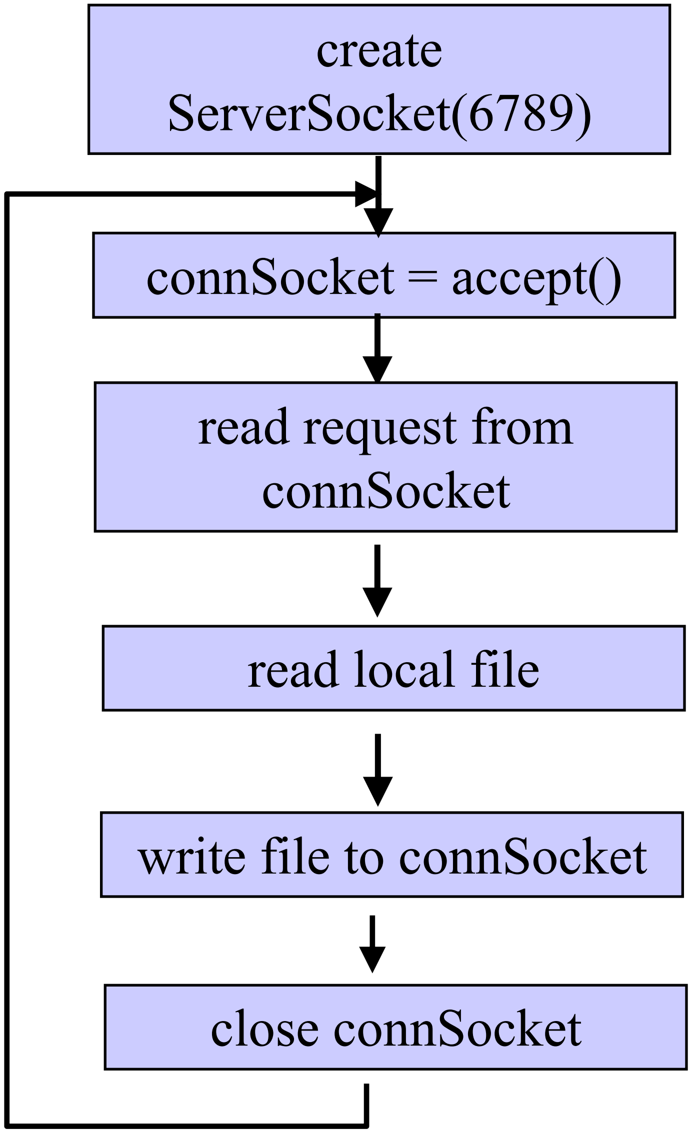
- Reduce data volume [Compression of data]
- Header compression [HTTP/2]

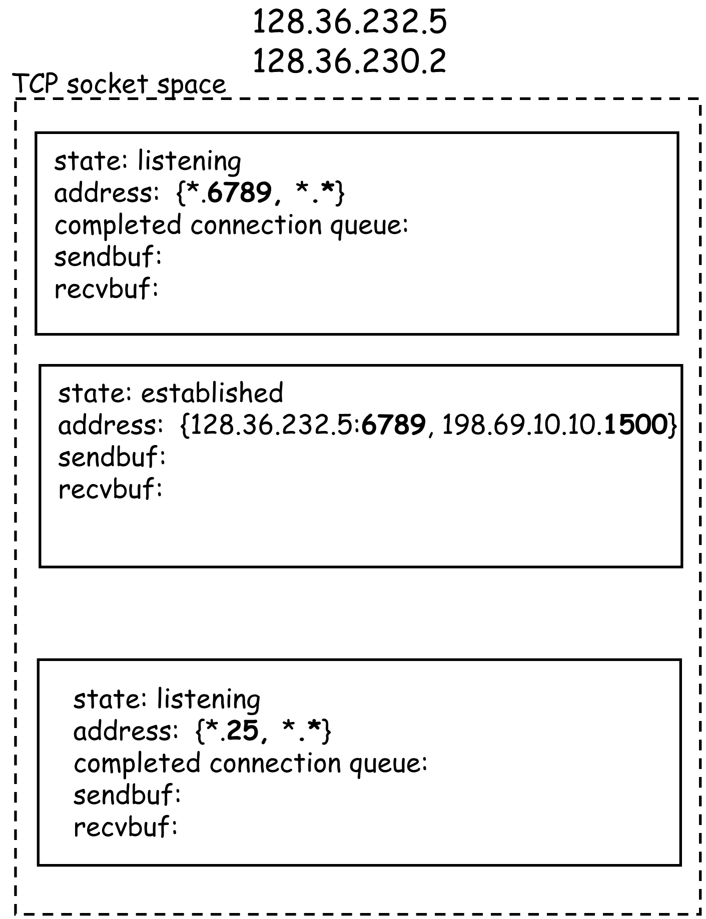
- Reduce the latency to the server to fetch the content [Proxy cache]
- Remove the extra RTTs to fetch an object [Persistent HTTP, aka HTTP/1.1]

- Increase concurrency [Multiple TCP connections]
- Asynchronous fetch (multiple streams) using a single TCP [HTTP/2]

- Server push [HTTP/2]

WebServer Implementation



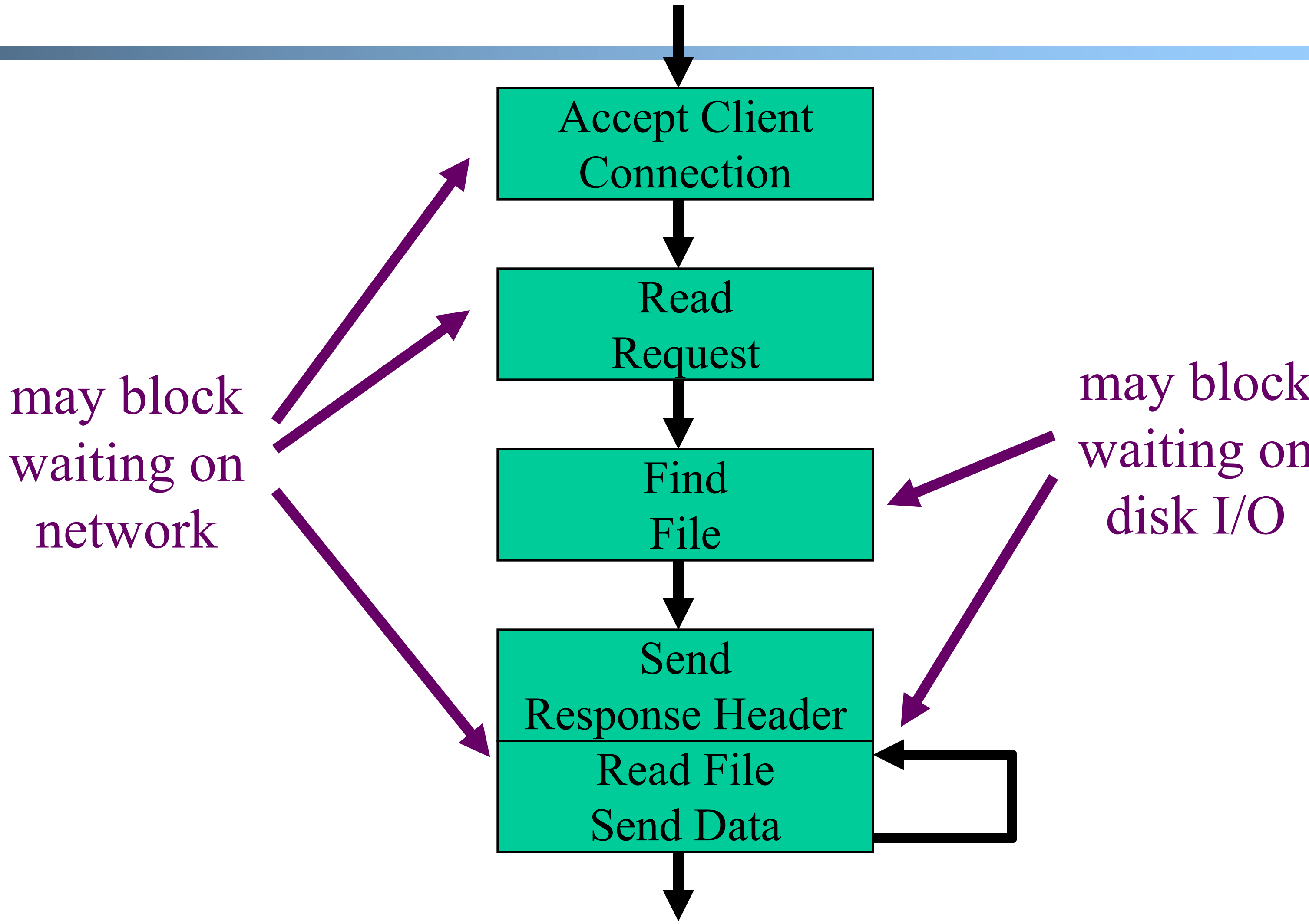


Discussion: what does each step do and how long does it take?

Demo

- Try TCPServer
- Start two TCPClient
  - Client 1 starts early but stops
  - Client 2 starts later but inputs first

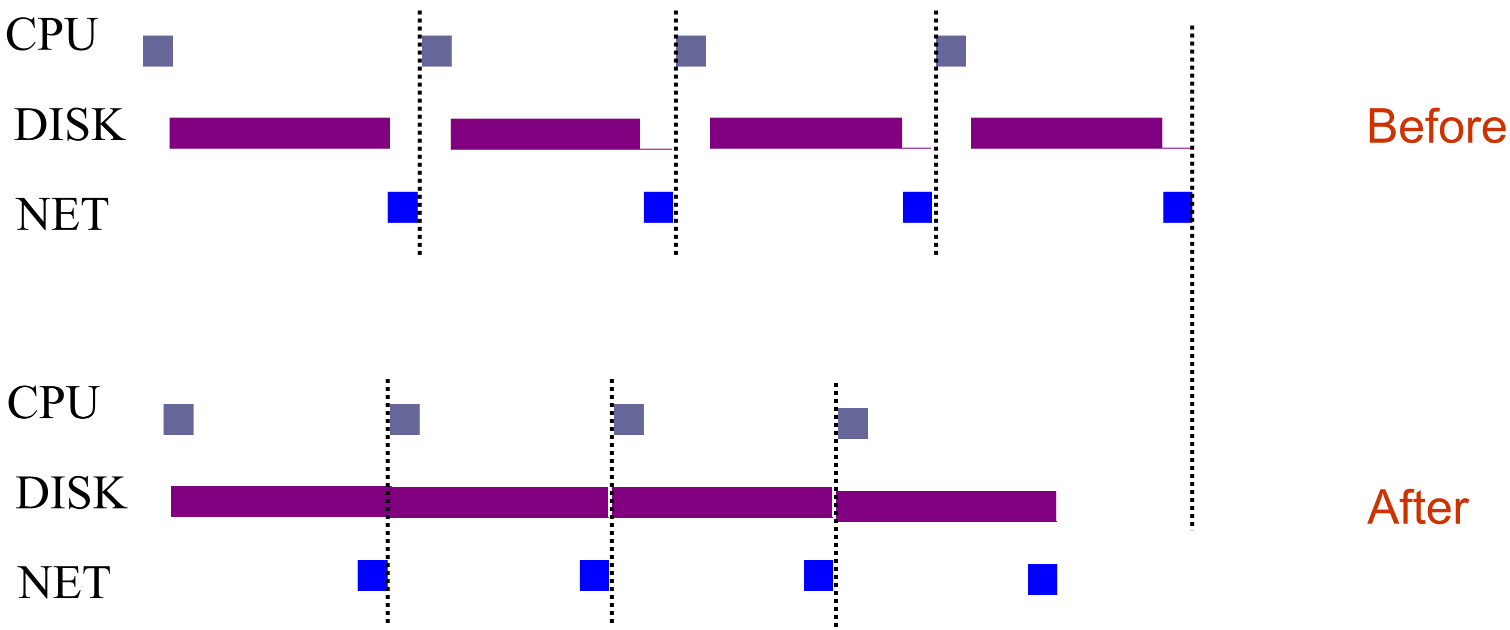
Server Processing Steps

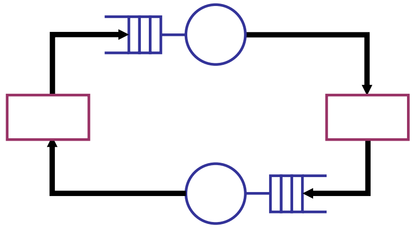


Writing High Performance Servers: Major Issues

- Many socket and IO operations can cause a process to block, e.g.,
  - `accept`: waiting for new connection;
  - `read` a socket waiting for data or close;
  - `write` a socket waiting for buffer space;
  - I/O `read/write` for disk to finish

Goal: Limited Only by Resource Bottleneck



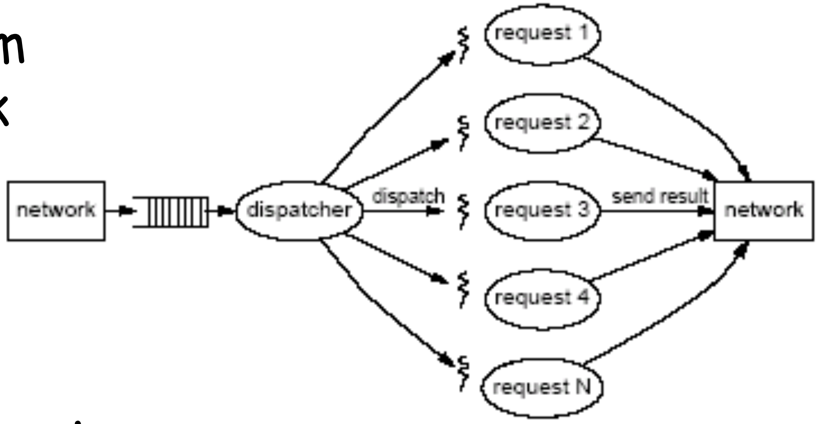


Outline

- Admin and recap
- Network server design
  - Overview
  - *Multi-thread network servers*

Multi-Threaded Servers

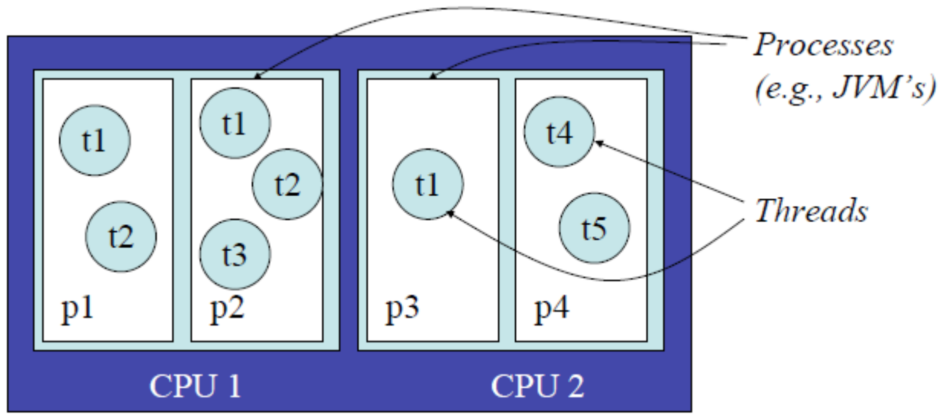
- Motivation:
  - Avoid blocking the whole program (so that we can reach bottleneck throughput)
- Idea: introduce threads
  - A thread is a sequence of instructions which may execute in parallel with other threads
  - When a blocking operation happens, only the flow (thread) performing the operation is blocked



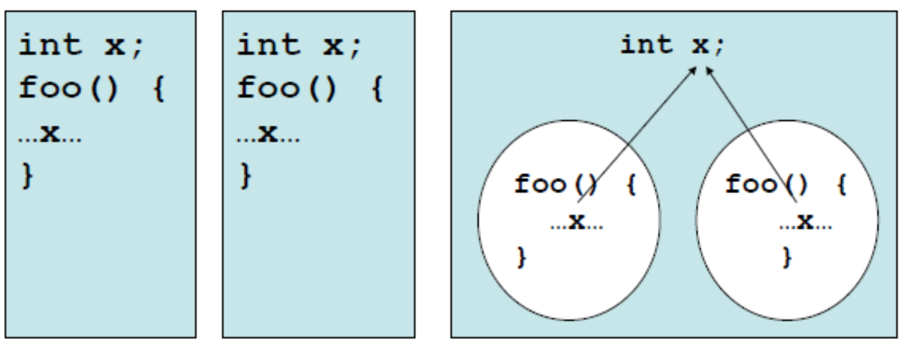
Background: Java Thread Model

- Every Java application has at least one thread
  - The "main" thread, started by the JVM to run the application's main() method
  - Most JVMs use POSIX threads to implement Java threads
- main() can create other threads
  - Explicitly, using the Thread class
  - Implicitly, by calling libraries that create threads as a consequence (RMI, AWT/Swing, Applets, etc.)

Thread vs Process



A computer



Processes do not share data

Threads share data within a process

Creating Java Thread

- Two ways to implement Java thread
  1. Extend the `Thread` class
     - Overwrite the `run()` method of the `Thread` class
  2. Create a class C implementing the `Runnable` interface, and create an object of type C, then use a `Thread` object to wrap up C
- A thread starts execution after its `start()` method is called, which will start executing the thread's (or the `Runnable` object's) `run()` method
- A thread terminates when the `run()` method returns

http://java.sun.com/javase/6/docs/api/java/lang/Thread.html

Option 1: Extending Java Thread

```
class PrimeThread extends Thread {
    long minPrime;

    PrimeThread(long minPrime) {
        this.minPrime = minPrime;
    }

    public void run() {
       // compute primes larger than minPrime  . . .
    }
}

PrimeThread p = new PrimeThread(143);
p.start();
```

Option 1: Extending Java Thread

```
class RequestHandler extends Thread {
    RequestHandler(Socket connSocket) {
      // …
    }
    public void run() {
      // process request
    }
    …
}

Thread t =  new RequestHandler(connSocket);
t.start();
```

Option 2: Implement the Runnable Interface

```
class PrimeRun implements Runnable {
    long minPrime;
    PrimeRun(long minPrime) {
        this.minPrime = minPrime;
    }

    public void run() {
        // compute primes larger than minPrime  . . .
    }
}

PrimeRun p = new PrimeRun(143);

new Thread(p).start();
```

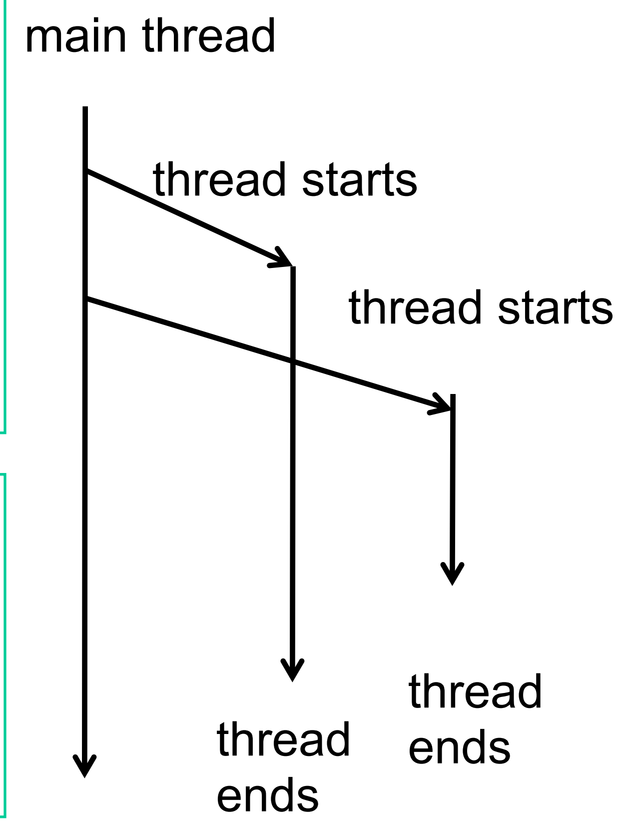
Example: a Multi-threaded TCPServer

- Turn TCPServer into a multithreaded server by creating a thread for each accepted request

Per-Request Thread Server

```
main() {
    ServerSocket s = new ServerSocket(port);
    while (true) {
        Socket conSocket = s.accept();
        RequestHandler rh
           =  new RequestHandler(conSocket);
        Thread t = new Thread (rh);
        t.start();
    }
```

```
class RequestHandler implements Runnable {
    RequestHandler(Socket connSocket) { … }
    public void run() {
        //
    } }
```



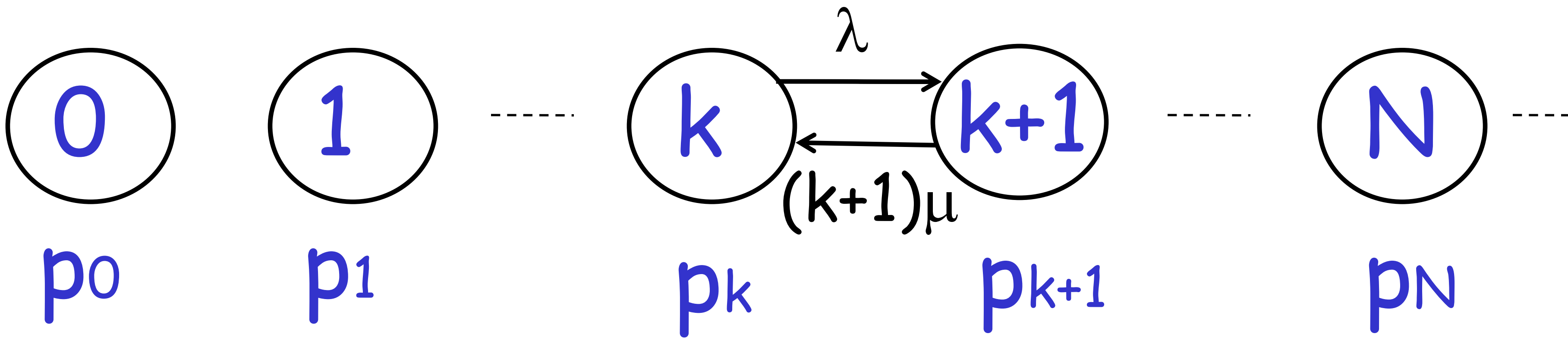
Try the per-request-thread TCP server: TCPServerMT.java

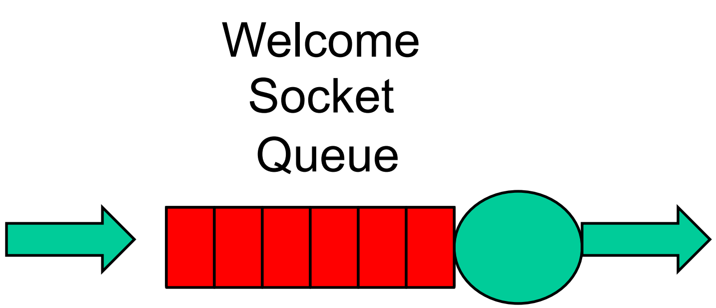
Summary: Implementing Threads

```
class RequestHandler
        extends Thread {
   RequestHandler(Socket connSocket)
   {
     …
   }
   public void run() {
     // process request
   }
   …
}


Thread t =  new RequestHandler(connSocket);
t.start();
```

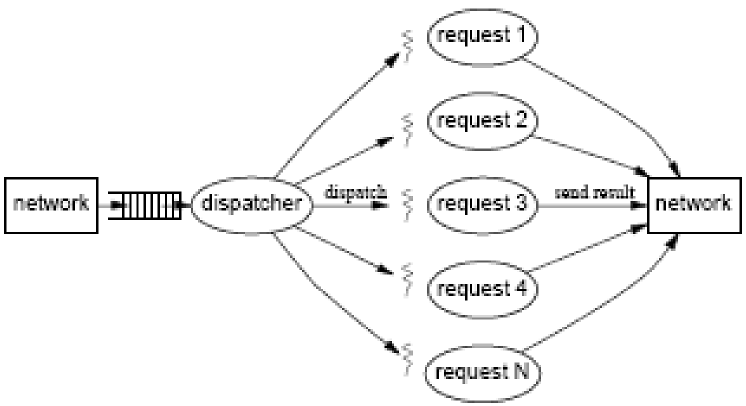
```
class RequestHandler
          implements Runnable {
   RequestHandler(Socket connSocket)
   {
     …
   }
   public void run() {
     // process request
   }
   …
}

RequestHandler rh =  new
          RequestHandler(connSocket);
Thread t = new Thread(rh);
t.start();
```

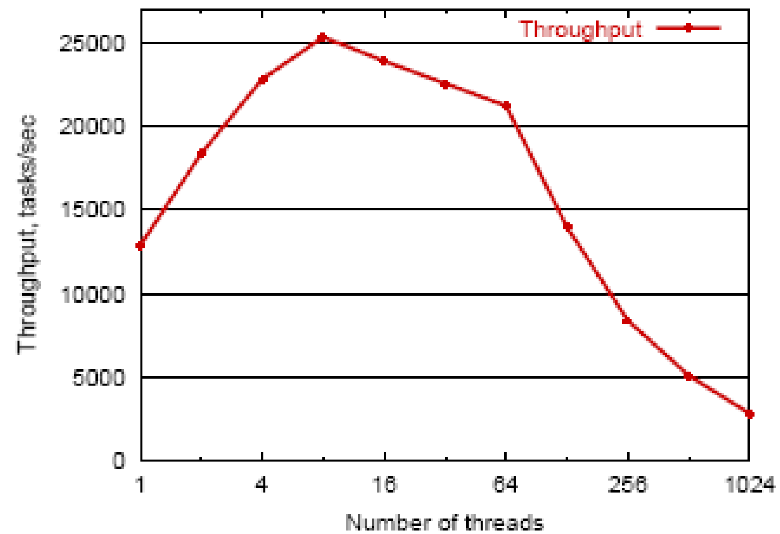
Modeling Per-Request Thread Server: Theory





Problem of Per-Request Thread: Reality





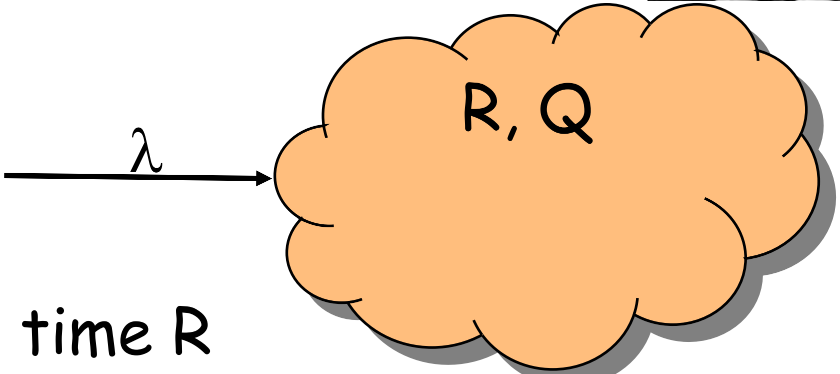
(937 MHz x86, Linux 2.2.14, each thread reading 8KB file)

- High thread creation/deletion overhead

- Too many threads → resource overuse → throughput meltdown → response time explosion
  - Q: given avg response time and connection arrival rate, how many threads active on avg?

Background: Little's Law (1961)

- For any system with no or (low) loss.
- Assume
  - mean arrival rate λ, mean time R at system, and mean number Q of requests at system
- Then relationship between Q, λ, and R:

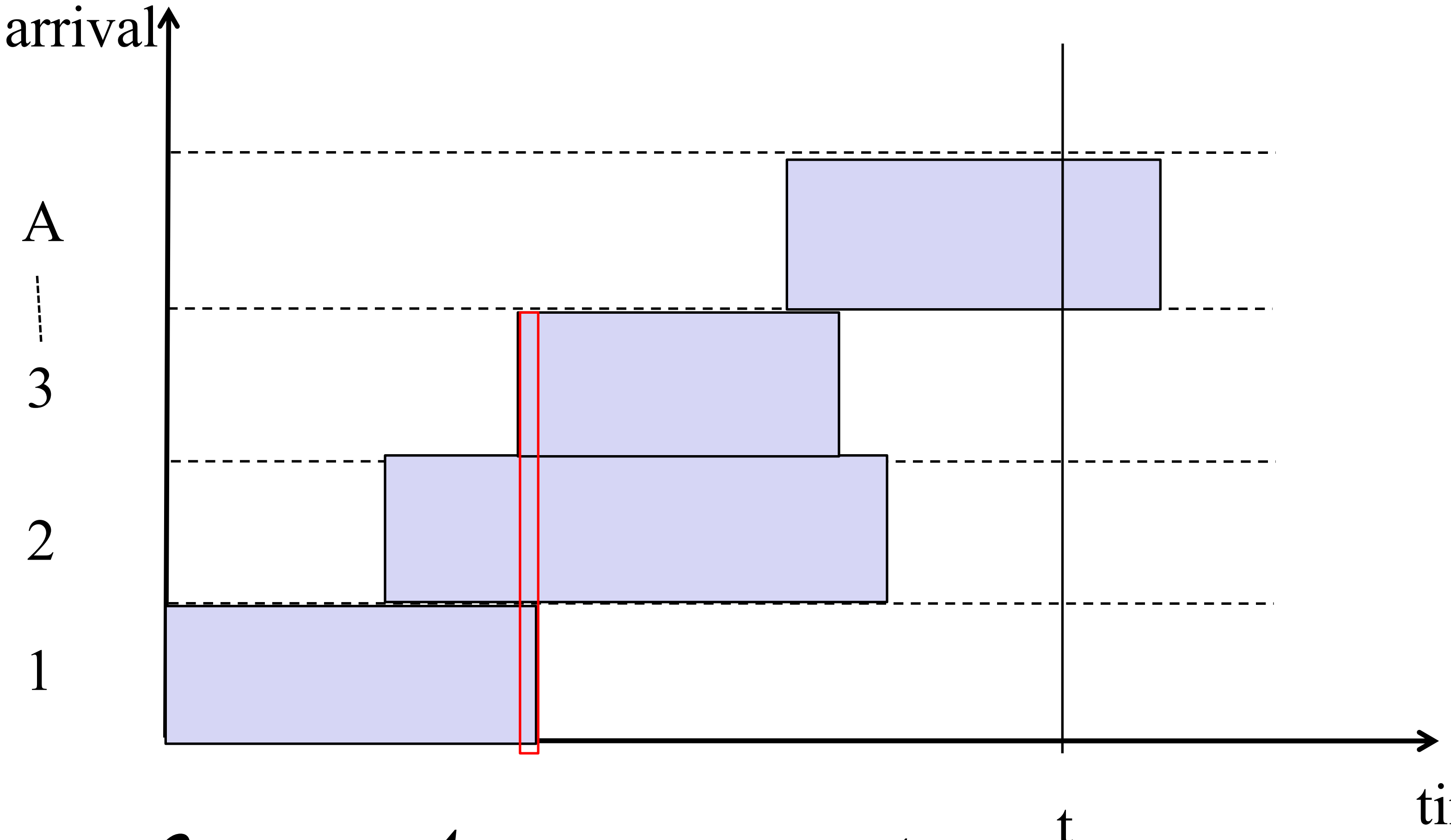


$$Q = \lambda R$$

Example: XMU admits 3000 students each year, and mean time a student stays is 4 years, how many students are enrolled?

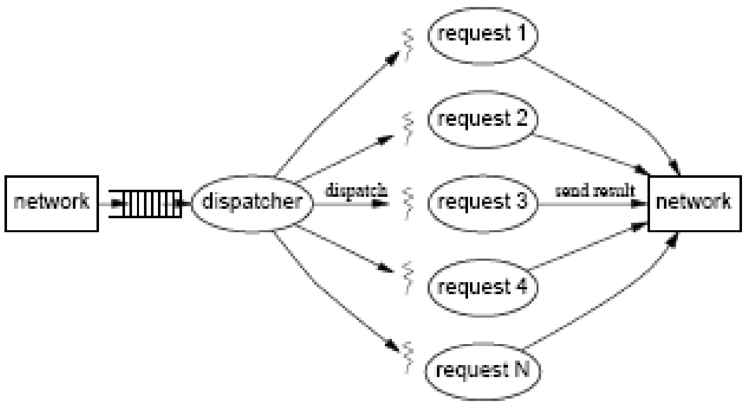
Little's Law: Proof

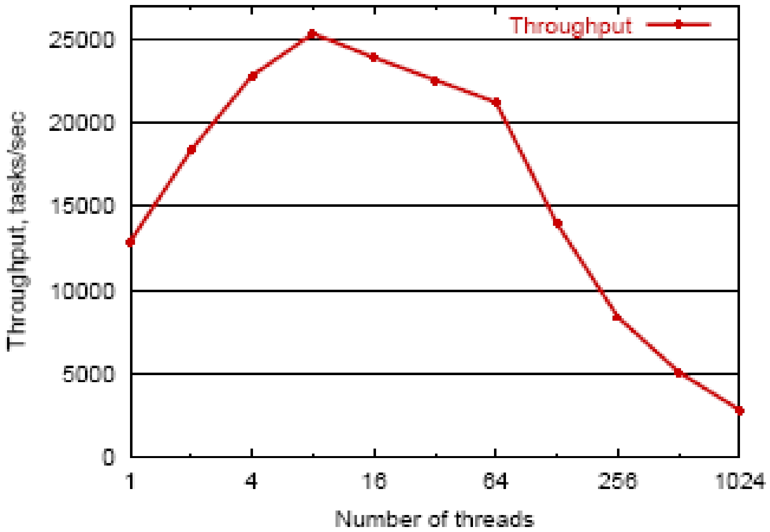
$$Q = \lambda R$$



$$\lambda = \frac{A}{t} \qquad R = \frac{Area}{A} \qquad Q = \frac{Area}{t}$$

Discussion: How to Address the Issue





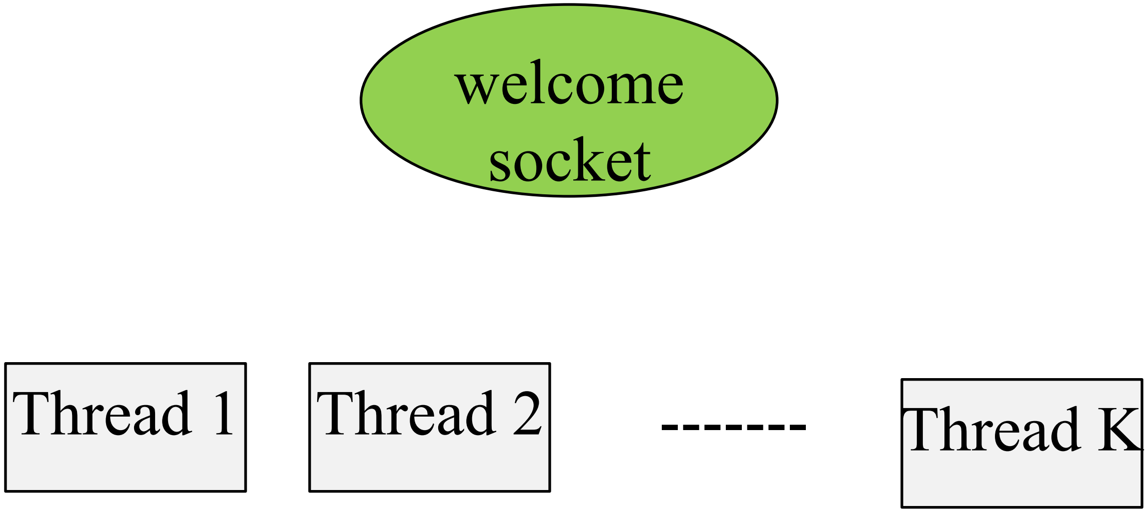
(937 MHz x86, Linux 2.2.14, each thread reading 8KB file)

Outline

- Admin and recap
- High-performance network server design
  - Overview
  - Threaded servers
    - Per-request thread
      - problem: large # of threads and their creations/deletions may let overhead grow out of control
    - *Thread pool*

Using a Fixed Set of Threads (Thread Pool)

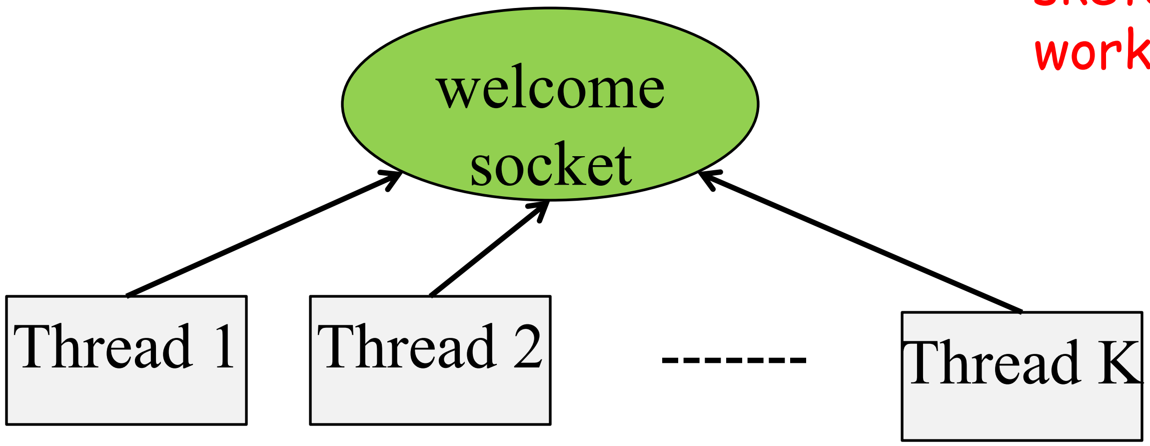
- Design issue: how to distribute the requests from the welcome socket to the thread workers



Design 1: Threads Share Access to the welcomeSocket

```
WorkerThread {
  void run {
    while (true) {
       Socket myConnSock = welcomeSocket.accept();
       // process myConnSock
       myConnSock.close();
    } // end of while
}
```

sketch; not working code

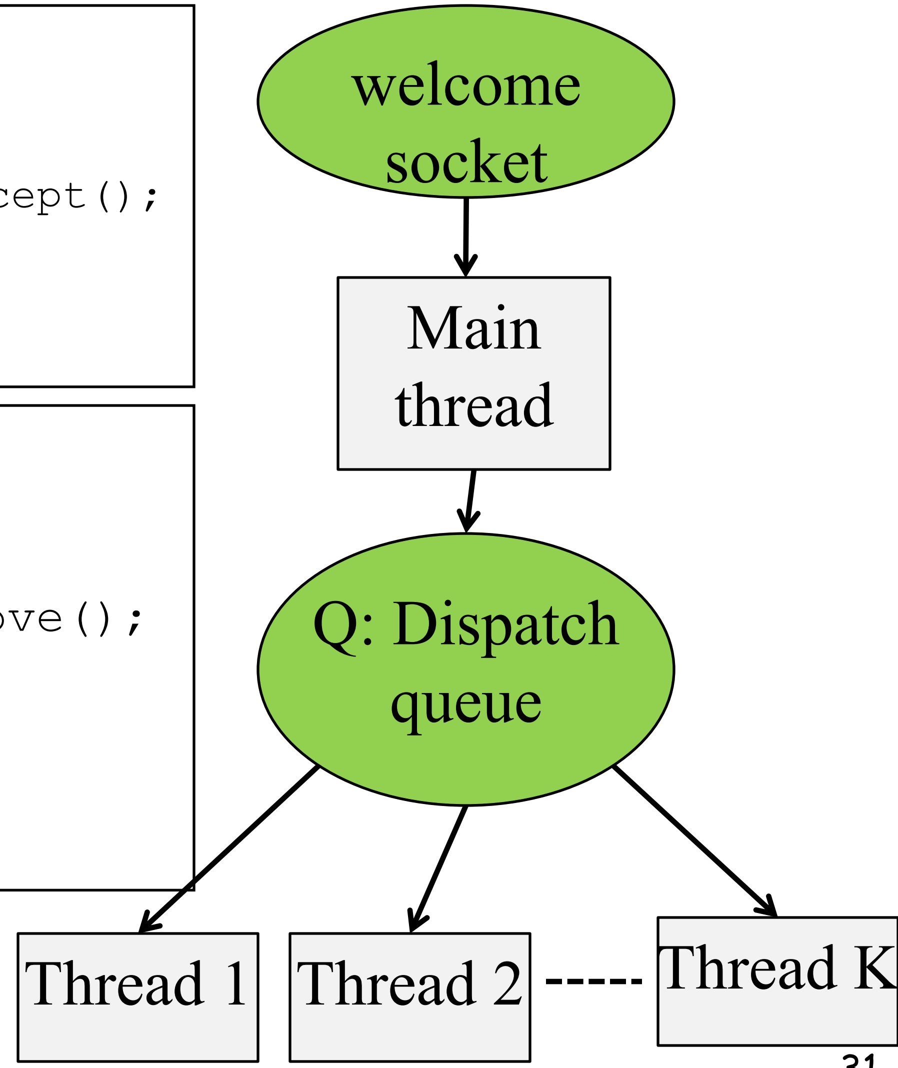


Design 2: Producer/Consumer

```
main {
  void run {
    while (true) {
       Socket con = welcomeSocket.accept();
       Q.add(con);
    } // end of while
}
```

```
WorkerThread {
  void run {
    while (true) {
       Socket myConnSock = Q.remove();
       // process myConnSock
       myConnSock.close();
    } // end of while
}
```

sketch; not
working code



Common Issues Facing Designs 1 and 2

- Both designs involve multiple threads modifying the same data concurrently
  - Design 1: welcomeSocket
  - Design 2: Q
- In our original TCPServerMT, do we have multiple threads modifying the same data concurrently?

Concurrency and Shared Data

- Concurrency is easy if threads don't interact
  - Each thread does its own thing, ignoring other threads
  - Typically, however, threads need to communicate/coordinate with each other
  - Communication/coordination among threads is often done by *shared data*

Simple Example

```
public class ShareExample extends Thread {
    private static int cnt = 0; // shared state, count
                               // total increases
    public void run() {
        int y = cnt;
        cnt = y + 1;
    }

    public static void main(String args[]) {
        Thread t1 = new ShareExample();
        Thread t2 = new ShareExample();
        t1.start();
        t2.start();
        Thread.sleep(1000);
        System.out.println("cnt = " + cnt);
    }
```

Q: What is the result of the program?

Simple Example

What if we add a println:

```
int y = cnt;
System.out.println("Calculating…");
cnt = y + 1;
```

What Happened?

- A thread was preempted in the middle of an operation
- The operations from reading to writing `cnt` should be *atomic* with no interference access to `cnt` from other threads
- But the scheduler interleaves threads and caused a race condition

- Such bugs can be extremely hard to reproduce, and also hard to debug

Synchronization

- Refers to mechanisms allowing a programmer to control the execution order of some operations across different threads in a concurrent program.
- We use Java as an example to see synchronization mechanisms
- We'll look at locks first.

Java Lock (1.5)

```
interface Lock {
    void lock();
    void unlock();
     ... /* Some more stuff, also */
}
class ReentrantLock implements Lock { ... }
```

- Only one thread can hold a lock at once
- Other threads that try to acquire it *block (or become suspended)* until the lock becomes available
- *Reentrant lock can be reacquired by same thread*
  - As many times as desired
  - No other thread may acquire a lock until it has been released the same number of times that it has been acquired
  - Do not worry about the reentrant perspective, consider it a lock

Java Lock

- Fixing the ShareExample.java problem

```
import java.util.concurrent.locks.*;
public class ShareExample extends Thread {
    private static int cnt = 0;
    static Lock lock = new ReentrantLock();

    public void run() {
        lock.lock();
        int y = cnt;
        cnt = y + 1;
        lock.unlock();
    }
    …
}
```

Java Lock

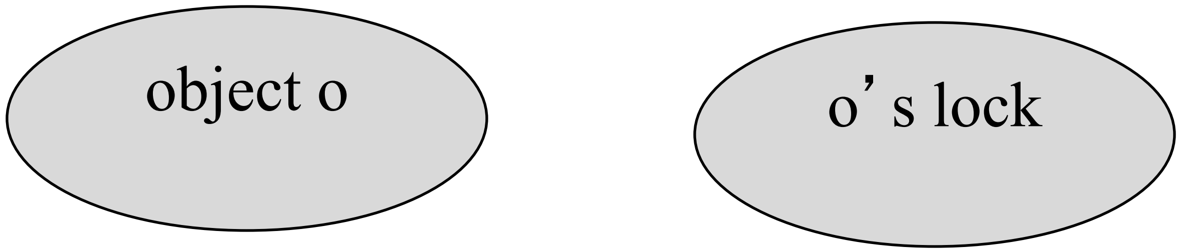
- It is recommended to use the following pattern

```
…
lock.lock();
try {
    // processing body
} finally {
    lock.unlock();

}
```

Java `synchronized`

- This pattern is really common
  - Acquire lock, do something, release lock after we are done, under any circumstances, even if exception was raised, the method returned in the middle, etc.
- Java has a language construct for this
  - `synchronized (obj) {` *`body`* `}`
    - Utilize the design that every Java object has its own implicitly lock object, also called the intrinsic lock, monitor lock or simply monitor
      - Obtains the lock associated with **obj**
      - Executes ***body***
      - Release lock when scope is exited
      - Even in cases of exception or method return

Discussion



- An object and its associated lock are different !
- Holding the lock on an object does not affect what you can do with that object in any way
- Examples:
  - `synchronized(o) { ... } // acquires lock named o`
  - `o.f (); // someone else can call o’s methods`
  - `o.x = 3; // someone else can read and write o’s fields`

Synchronization on `this`

```
class C {
    int cnt;
    void inc() {
        synchronized (this) {
            cnt++;
        } // end of sync
    } // end of inc
}
```

```
C c = new C();
```

```
Thread 1
c.inc();
```

```
Thread 2
c.inc();
```

- A program can often use `this` as the object to lock
- Does the program above have a data race?
  - No, both threads acquire locks on the same object before they access shared data

Synchronization on `this`

```
class C {
   static int cnt;
   void inc() {
      synchronized (this) {
         cnt++;
      } // end of sync
   } // end of inc

   void dec() {
      synchronized (this) {
         cnt--;
      } // end of sync
   } // end of dec
}
```

```
C c = new C();
```

```
Thread 1
c.inc();
```

```
Thread 2
c.dec();
```

- Does the program above have a data race?
  - No, both threads acquire locks on the same object before they access shared data

Example

- See
  - ShareWelcome/Server.java
  - ShareWelcome/ServiceThread.java

Discussion

- You would not need the lock for `accept` if Java were to label the call as thread safe (synchronized)
- One reason Java does not specify `accept` as thread safe is that one could register your own socket implementation with ServerSocket.setSocketFactory
- Always consider thread safety in your design
  - If a resource is shared through concurrent read/write, write/write), consider thread-safe issues.

Why not Synchronization

- Synchronized method invocations generally are going to be slower than non-synchronized method invocations
- Synchronization gives rise to the possibility of deadlock, a severe performance problem in which your program appears to hang

Synchronization Overhead

- Try SyncOverhead.java

Synchronization Overhead

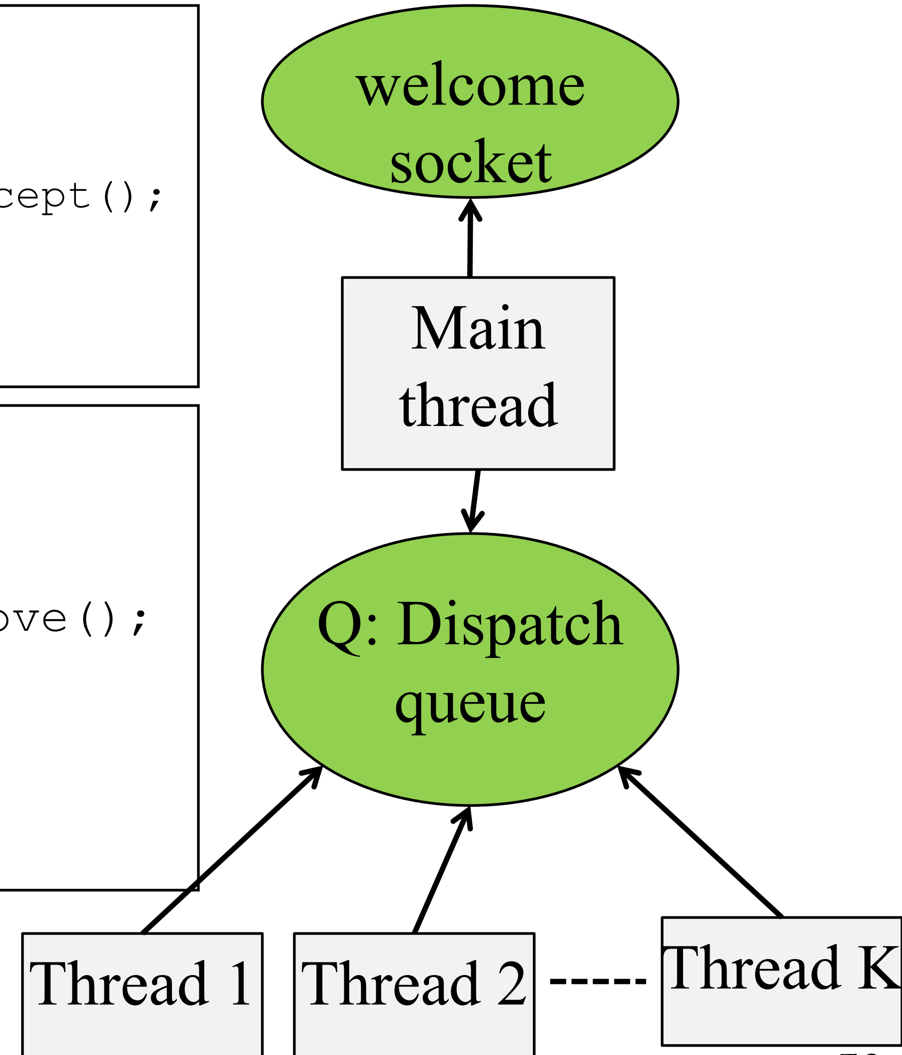
- Try SyncOverhead.java

Method	Time (ms; 5,000,000 exec)
no sync	8 ms
synchronized method	18 ms
synchronized on this	18 ms
lock	89 ms
lock and finally	88 ms

Design 2: Producer/Consumer

```
main {
  void run {
    while (true) {
       Socket con = welcomeSocket.accept();
       Q.add(con);
    } // end of while
}
```

```
WorkerThread {
  void run {
    while (true) {
       Socket myConnSock = Q.remove();
       // process myConnSock
       myConnSock.close();
    } // end of while
}
```



How to turn it into working code?

Main

```
main {
  void run {
    while (true) {
       Socket con = welcomeSocket.accept();
       Q.add(con);
    } // end of while
}
```

```
main {
  void run {
    while (true) {
       Socket con = welcomeSocket.accept();
       synchronized(Q) {
         Q.add(con);
       }
    } // end of while
}
```

Worker

```
WorkerThread {
  void run {
    while (true) {
        Socket myConnSock = Q.remove();
        // process myConnSock
        myConnSock.close();
    } // end of while
}
```

```
while (true) {
    // get next request
    Socket myConn = null;
    while (myConn==null) {
      synchronize(Q) {
        if (!Q.isEmpty())
            myConn = (Socket) Q.remove();
      }
    } // end of while
    // process myConn
}
```

Example

- try
  - ShareQ/Server.java
  - ShareQ/ServiceThread.java

Problem of ShareQ Design

- Worker thread continually spins (busy wait) until a condition holds

```
while (true) { // spin
   lock;
   if (Q.condition) // {
      // do something
   } else {
      // do nothing
   }
   unlock
} //end while
```

- Can lead to high utilization and slow response time

- Q: Does the shared welcomeSock have busy-wait?