Network Applications: High-performance Server Design

Qiao Xiang, Congming Gao

https://sngroup.org.cn/courses/cnns-xmuf23/index.shtml

10/26/2023

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

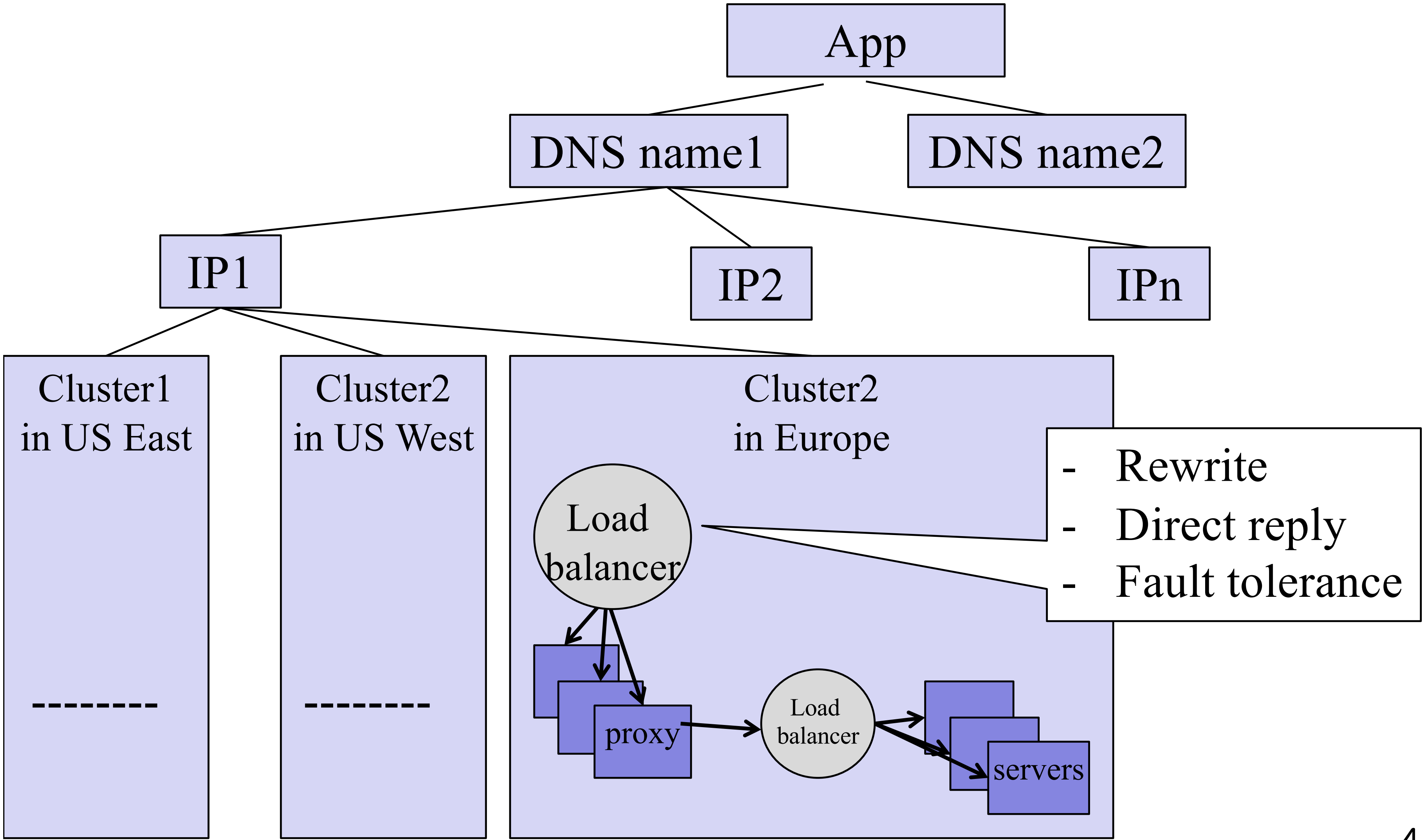
Outline

- Admin and recap
- High-performance network server design
  - Overview
  - Threaded servers
    - Per-request thread
      - problem: large # of threads and their creations/deletions may let overhead grow out of control
    - Thread pool
      - Design 1: Service threads compete on the welcome socket
      - Design 2: Service threads and the main thread coordinate on the shared queue
        - polling (busy wait)
        - suspension: wait/notify

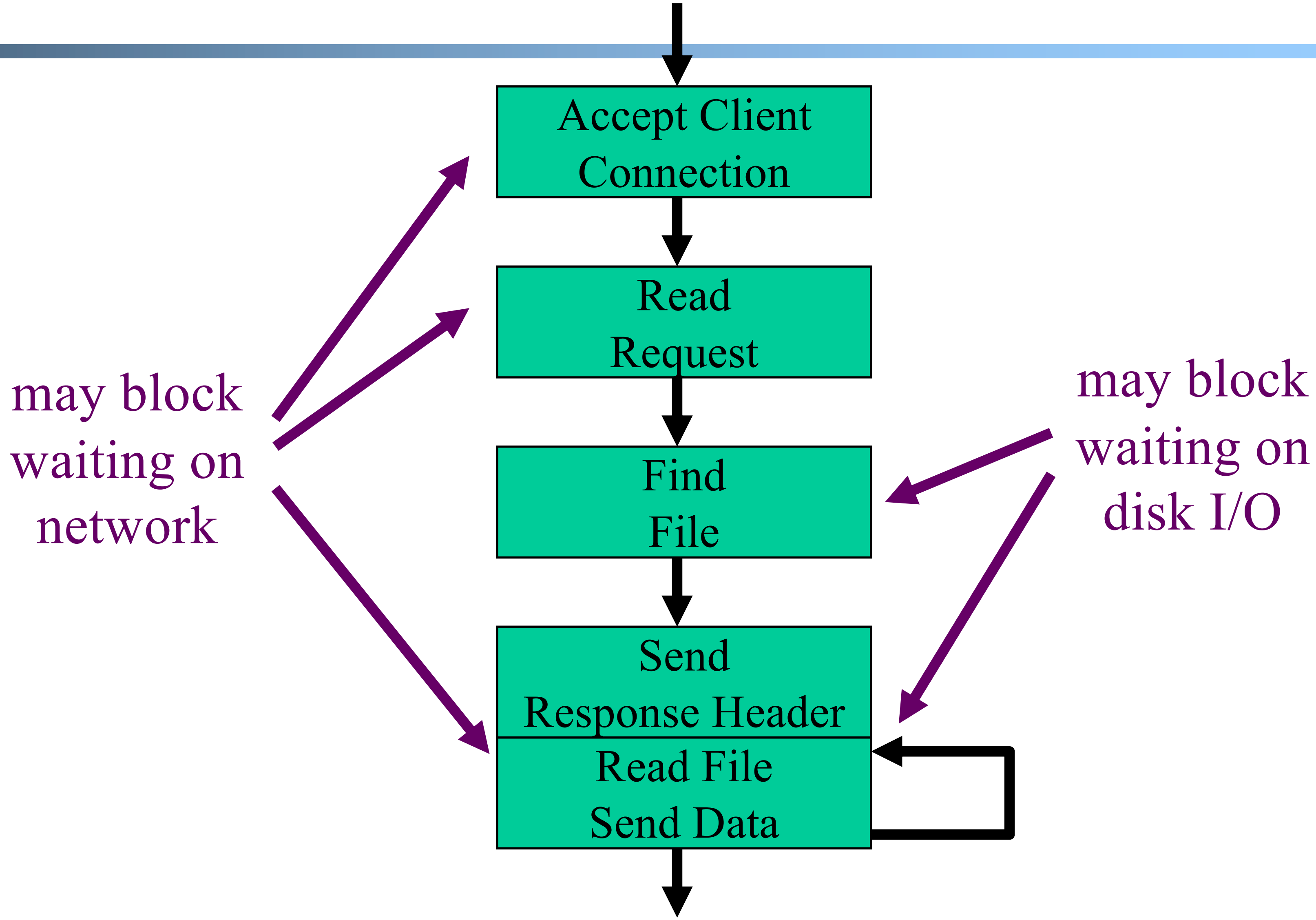
Admin

- Exam 1 date: 2:30-4:10pm, Nov. 9
- Assignment 3: to be posted today

Recap: Direction Mechanisms

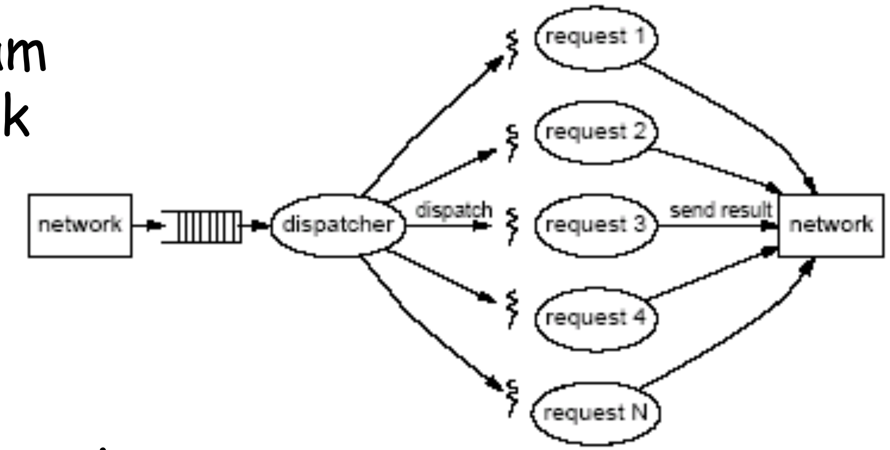


Recap: Server Processing Steps



Recap: 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



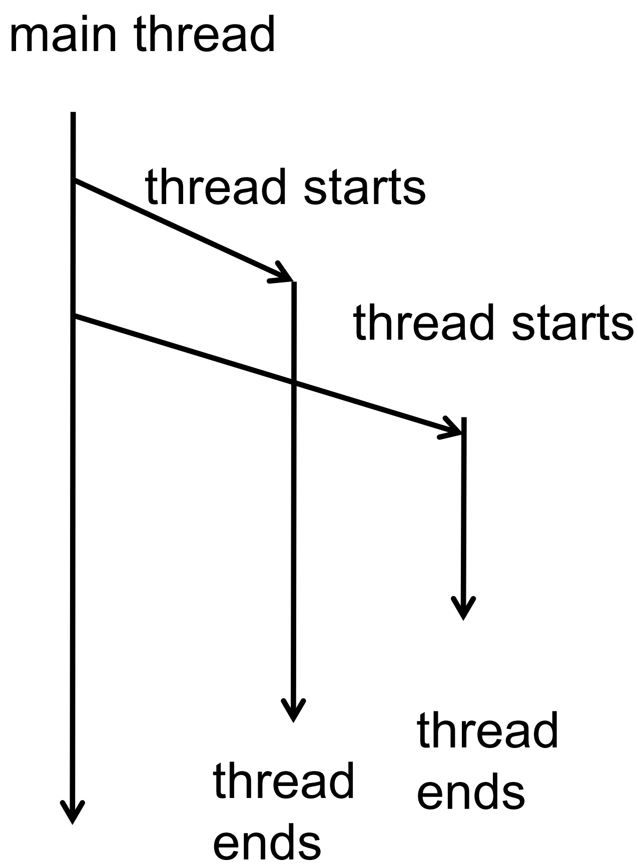
Background: Java Thread Model

- 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.)

Recap: 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() {
      //
   } }
```



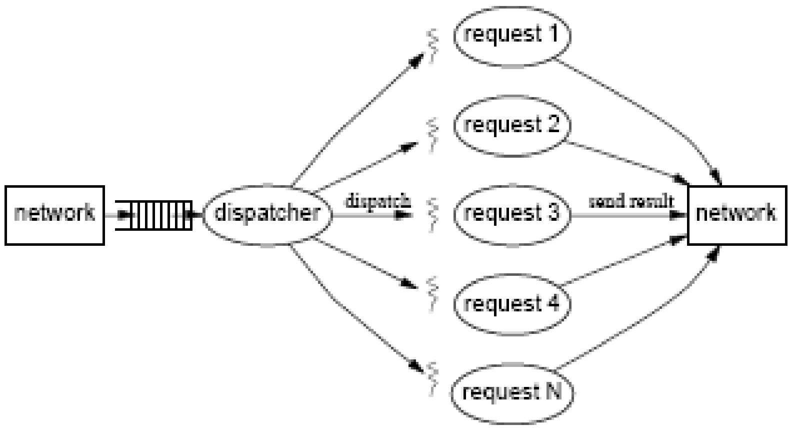
Try the per-request-thread TCP server: TCPServerMT.java

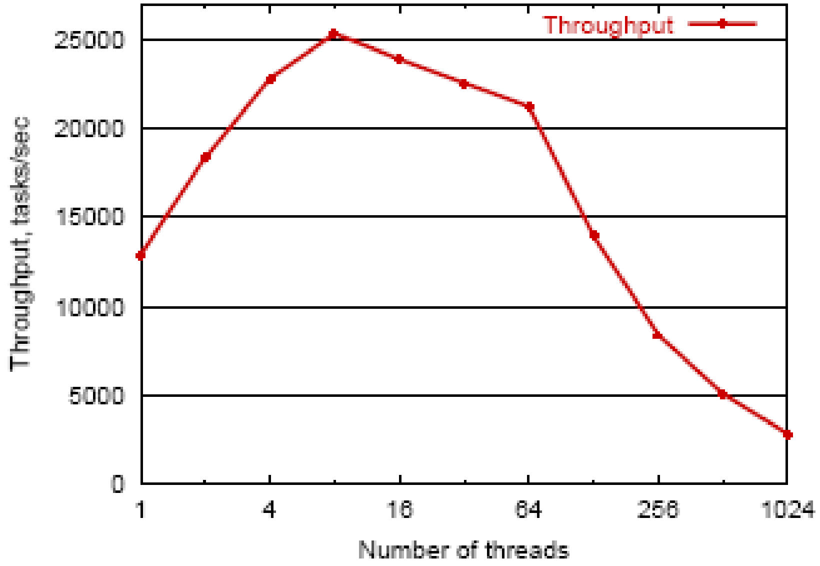
Recap: Implementing Threads

```
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();
```

Problem of Per-Request Thread: Reality





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

- High thread creation/deletion overhead

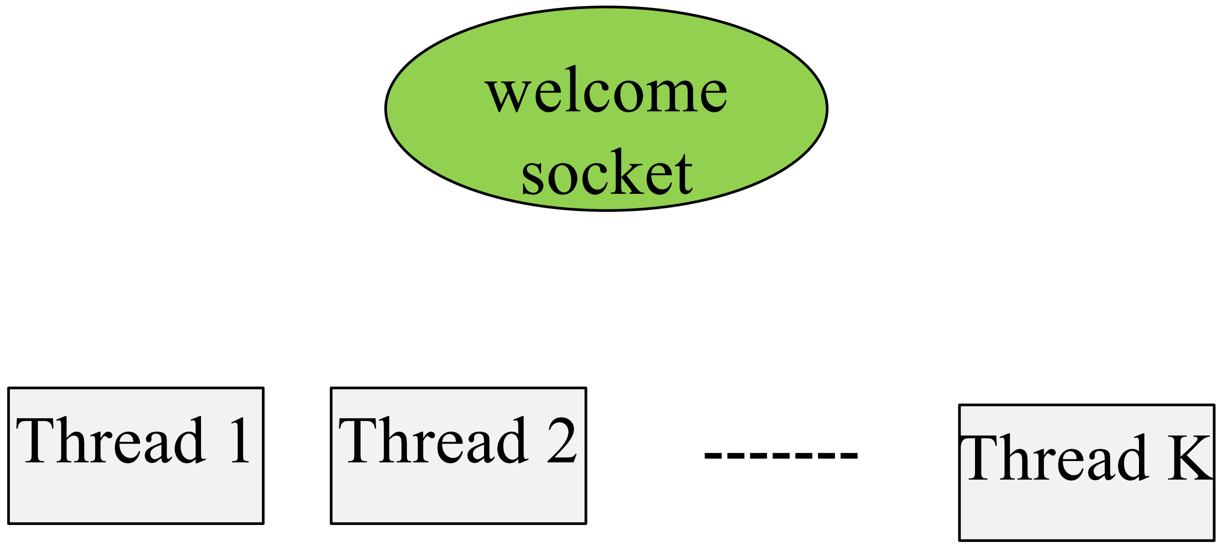
- Too many threads → resource overuse → throughput meltdown → response time explosion
  - Q: given avg response time and connection arrival rate, how many threads active on avg?

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)

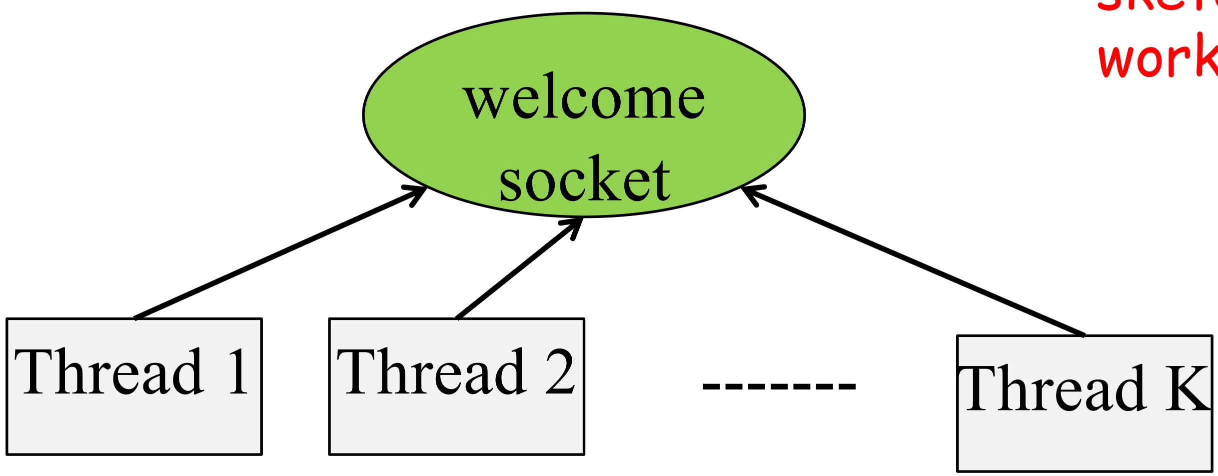
- Design issue: how to distribute the requests from the welcome socket to the thread workers



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

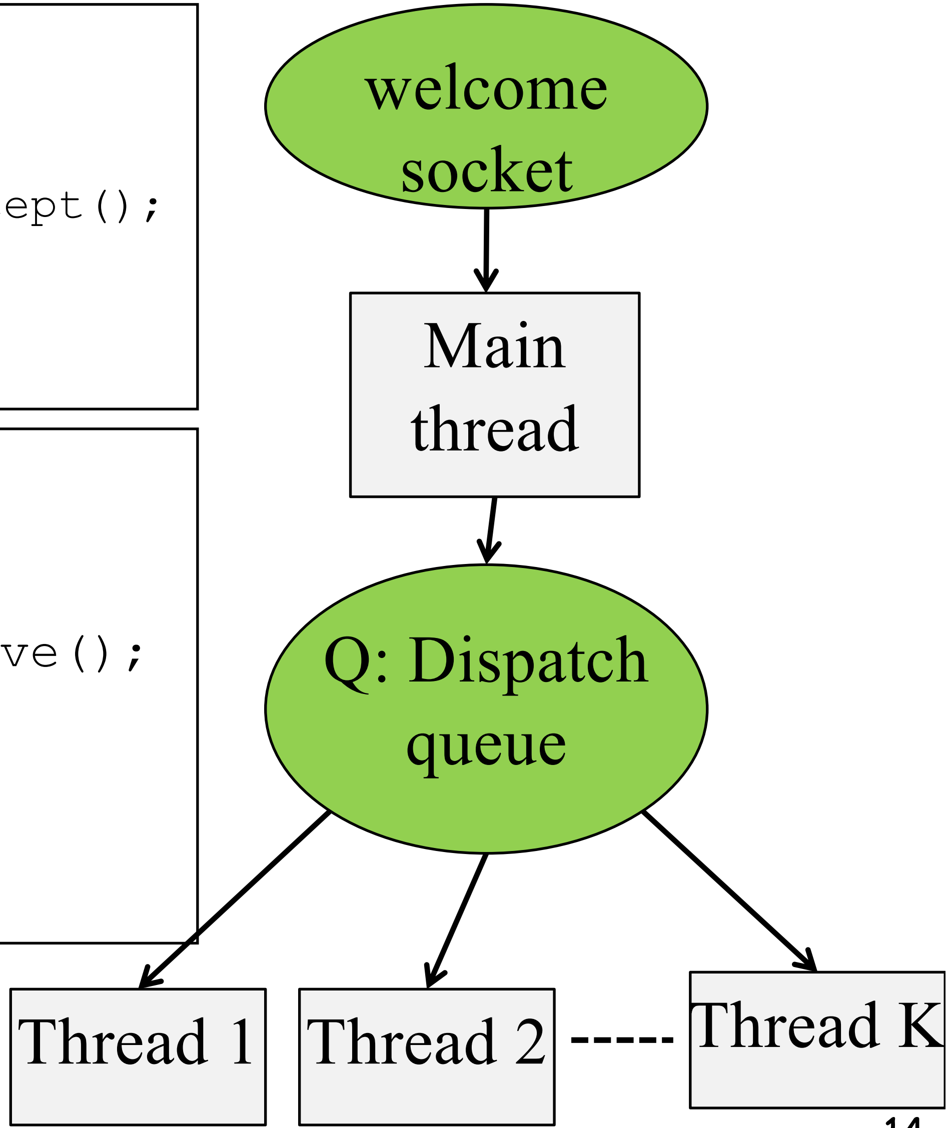


Design 2: Producer/Consumer

```
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

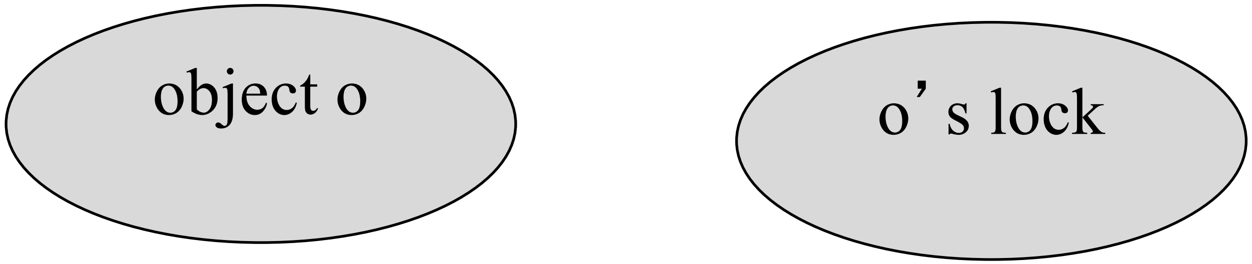
- It is recommended to use the following pattern

```
…
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

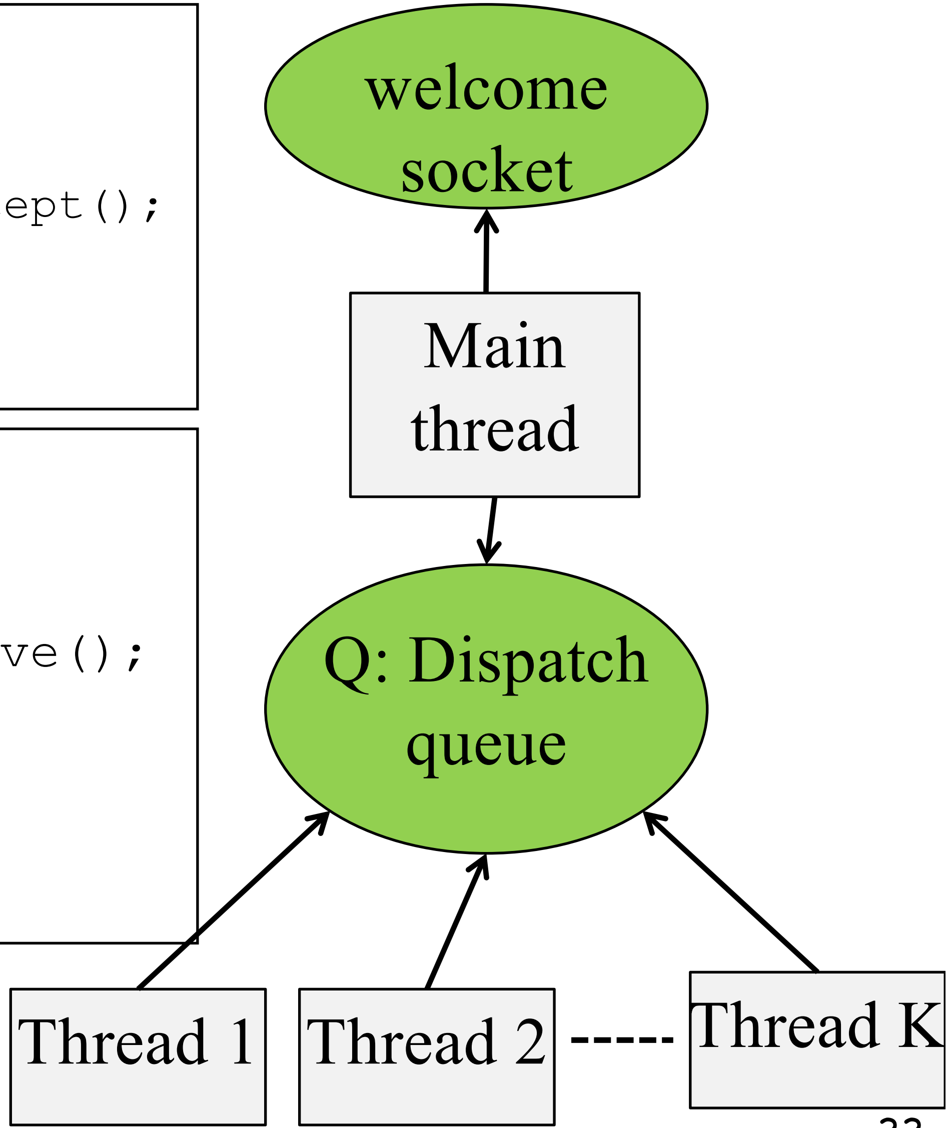
- Try SyncOverhead.java

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?

Solution: Suspension

- Put thread to sleep to avoid busy spin
- Thread life cycle: while a thread executes, it goes through a number of different phases
  - New: created but not yet started
  - Runnable: is running, or can run on a free CPU
  - Blocked: waiting for socket/I/O, a lock, or suspend (wait)
  - Sleeping: paused for a user-specified interval
  - Terminated: completed

Solution: Suspension

```
while (true) {
    // get next request
    Socket myConn = null;
    while (myConn==null) {
      lock Q;
      if (Q.isEmpty()) // {
        // stop and wait
      } else {
        // get myConn from Q
      }
      unlock Q;
    }
    // get the next request; process
}
```

Hold lock? (points to "// stop and wait")

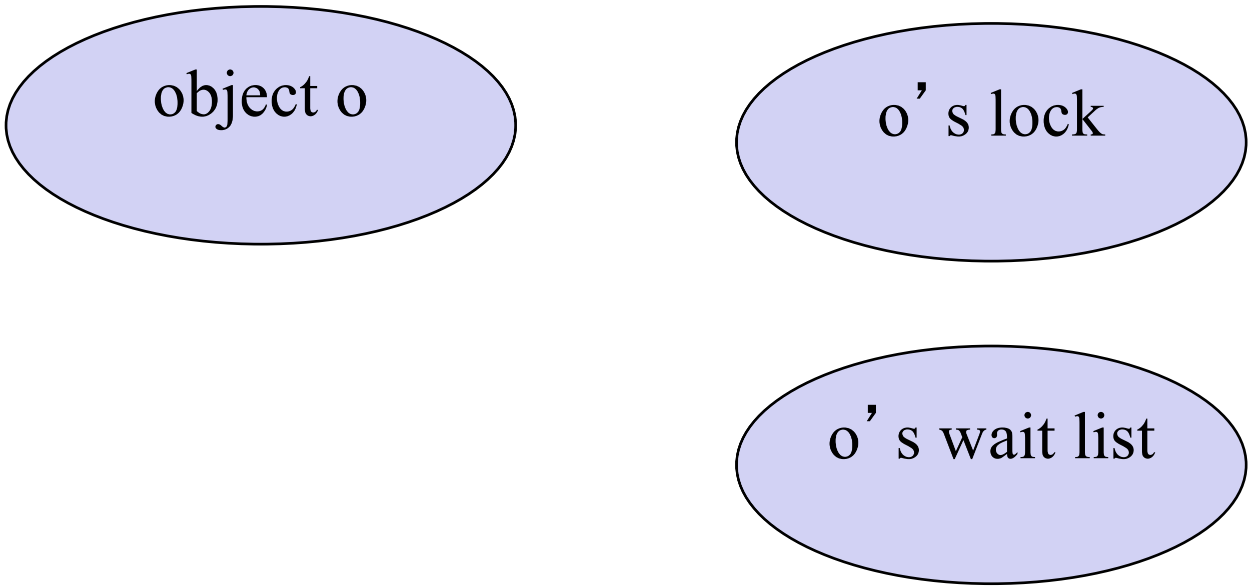
Solution: Suspension

```
while (true) {
   // get next request
   Socket myConn = null;
   while (myConn==null) {
     lock Q;
     if (Q.isEmpty()) // {
       // stop and wait
     } else {
       // get myConn from Q
     }
     unlock Q;
   }
   // get the next request; process
}
```

Design pattern:
- Need to release lock to avoid deadlock (to allow main thread write into Q)
- Typically need to reacquire lock after waking up

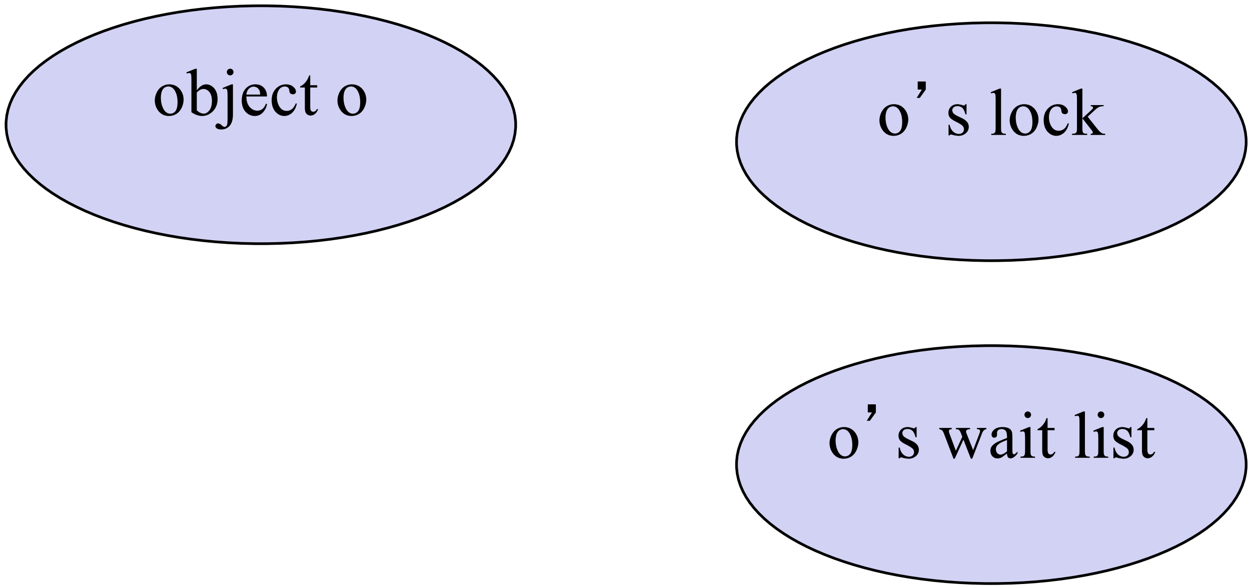
Wait-sets and Notification

- Every Java Object has an associated wait-set (called wait list) in addition to a lock object



Wait-sets and Notification

- Wait list object can be manipulated only while the object lock is held
  - Otherwise, `IllegalMonitorStateException` is thrown



Wait-sets

- Thread enters the wait-set by invoking `wait()`
  - `wait()` releases the lock
    - No other held locks are released
  - then the thread is suspended

- Can add optional time `wait(long millis)`
  - `wait()` is equivalent to `wait(0)` - wait forever
  - for robust programs, it is typically a good idea to add a timer

Worker

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

```
while (true) {
   // get next request
   Socket myConn = null;
   synchronized(Q) {
       while (Q.isEmpty()) {
         Q.wait();
       }
       myConn = Q.remove();
    } // end of sync
    // process request in myConn
} // end of while
```

Note the while loop; no guarantee that Q is not empty when wake up

Wait-set and Notification (cont)

- Threads are released from the wait-set when:
  - `notifyAll()` is invoked on the object
    - All threads released (typically recommended)
  - `notify()` is invoked on the object
    - One thread selected at 'random' for release
  - The specified time-out elapses
  - The thread has its `interrupt()` method invoked
    - `InterruptedException` thrown
  - A spurious wakeup occurs
    - Not (yet!) spec'ed but an inherited property of underlying synchronization mechanisms e.g., POSIX condition variables

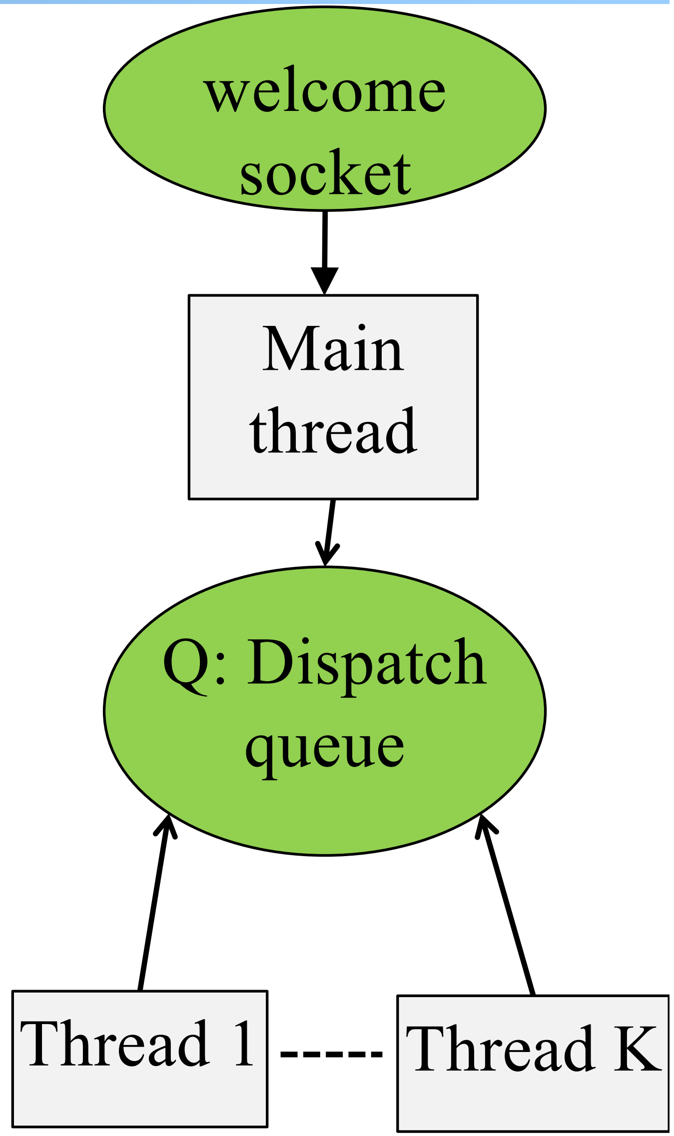
Notification

- Caller of `notify()` must hold lock associated with the object
- Those threads awoken must reacquire lock before continuing
  - (This is part of the function; you don't need to do it explicitly)
  - Can't be acquired until notifying thread releases it
  - A released thread contends with all other threads for the lock

Main Thread

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

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



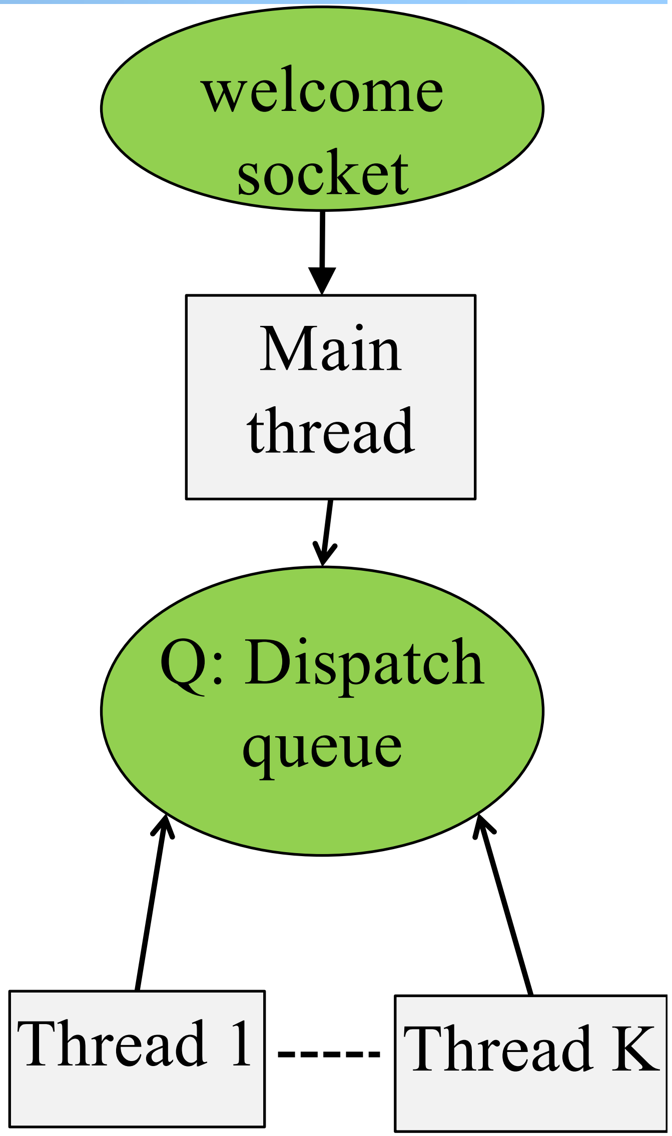
Worker

Busy wait

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

Suspend

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



Worker: Another Format

```
while (true) {
    // get next request
    Socket myConn = null;
    synchronized(Q) {
        while (Q.isEmpty()) {
            Q.wait();
        }
        myConn = Q.remove();
    } // end of sync
    // process request in myConn
} // end of while
```

Note the while loop; no guarantee that Q is not empty when wake up

Example

- See
  - WaitNotify/Server.java
  - WaitNotify/ServiceThread.java

Summary: Guardian via Suspension: Waiting

```
synchronized (obj) {
    while (!condition) {
      try { obj.wait(); }
      catch (InterruptedException ex)
      { ... }
    } // end while
    // make use of condition
  } // end of sync
```

- Golden rule: Always test a condition in a loop
  - Change of state may not be what you need
  - Condition may have changed again
- Break the rule only after you are sure that it is safe to do so

Summary: Guarding via Suspension: Changing a Condition

```
synchronized (obj) {
   condition = true;
   obj.notifyAll(); // or obj.notify()
}
```

- Typically use `notifyAll()`
- There are subtle issues using notify(), in particular when there is interrupt

Note

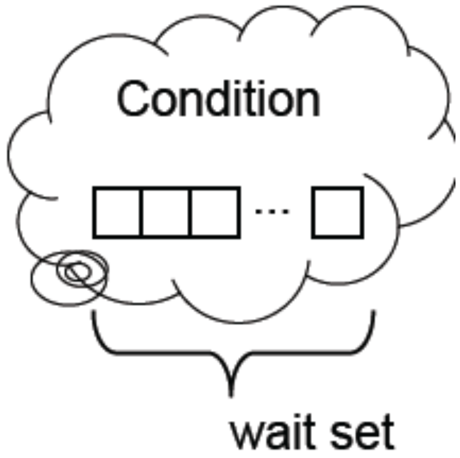
- Use of wait(), notifyAll() and notify() similar to
  - Condition queues of classic Monitors
  - Condition variables of POSIX PThreads API
  - In C# it is called Monitor (http://msdn.microsoft.com/en-us/library/ms173179.aspx)

- Python Thread module in its Standard Library is based on Java Thread model (https://docs.python.org/3/library/threading.html)
  - "The design of this module is loosely based on Java's threading model. However, where Java makes locks and condition variables basic behavior of every object, they are separate objects in Python."

Java (1.5)

```
interface Lock { Condition newCondition(); ... }
interface Condition {
   void await();
   void signalAll(); ...
}
```

- Condition created from a Lock
- `await` called with lock held
  - Releases the lock
    - But not any other locks held by this thread
  - Adds this thread to wait set for lock
  - Blocks the thread



- `signallAll` called with lock held
  - Resumes all threads on lock's wait set
  - Those threads must reacquire lock before continuing
    - (This is part of the function; you don't need to do it explicitly)

Producer/Consumer Example

```
Lock lock = new ReentrantLock();
Condition ready = lock.newCondition();
boolean valueReady = false;
Object value;
```

```
void produce(Object o) {
   lock.lock();
   while (valueReady)
     ready.await();
   value = o;
   valueReady = true;
   ready.signalAll();
   lock.unlock();
}
```

```
Object consume() {
   lock.lock();
   while (!valueReady)
      ready.await();
   Object o = value;
   valueReady = false;
   ready.signalAll();
   lock.unlock();
}
```

Blocking Queues in Java

- Design Pattern for producer/consumer pattern with blocking, e.g.,
  - put/take
- Two handy implementations
  - LinkedBlockingQueue (FIFO, may be bounded)
  - ArrayBlockingQueue (FIFO, bounded)
  - (plus a couple more)

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html

Beyond Class: Complete Java Concurrency Framework

Executors

- **Executor**
- **ExecutorService**
- **ScheduledExecutorService**
- Callable
- **Future**
- ScheduledFuture
- Delayed
- CompletionService
- **ThreadPoolExecutor**
- **ScheduledThreadPoolExecutor**
- AbstractExecutorService
- Executors
- FutureTask
- ExecutorCompletionService

Queues

- BlockingQueue
- ConcurrentLinkedQueue
- LinkedBlockingQueue
- ArrayBlockingQueue
- SynchronousQueue
- PriorityBlockingQueue
- DelayQueue

Concurrent Collections

- ConcurrentMap
- ConcurrentHashMap
- CopyOnWriteArray{List,Set}

Synchronizers

- CountDownLatch
- Semaphore
- Exchanger
- CyclicBarrier

Locks: java.util.concurrent.locks

- Lock
- Condition
- ReadWriteLock
- AbstractQueuedSynchronizer
- LockSupport
- ReentrantLock
- ReentrantReadWriteLock

Atomics: java.util.concurrent.atomic

- Atomic[Type]
- Atomic[Type]Array
- Atomic[Type]FieldUpdater
- Atomic{Markable,Stampable}Reference

See jcf slides for a tutorial.

Correctness

- Threaded programs are typically more complex.
- What types of properties do you analyze to verify server correctness?

```
// worker
void run() {
  while (true) {
    // get next request
    Socket myConn = null;
    synchronized(Q) {
      while (Q.isEmpty()) {
        Q.wait();
      } // end of while
      myConn = Q.remove();
    } // end of sync
    // process request in myConn
  } // end of while
} // end of run()
```

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

Key Correctness Properties

- Safety
- Liveness (progress)
- Fairness
  - For example, in some settings, a designer may want the threads to share load equally

Safety Properties

- What safety properties?
  - No read/write; write/write conflicts
    - holding lock Q before reading or modifying shared data Q and Q.wait_list
  - Q.remove() is not on an empty queue
- There are formal techniques to model server programs and analyze their properties, but we will use basic analysis
  - This is enough in many cases

Make Program Explicit

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

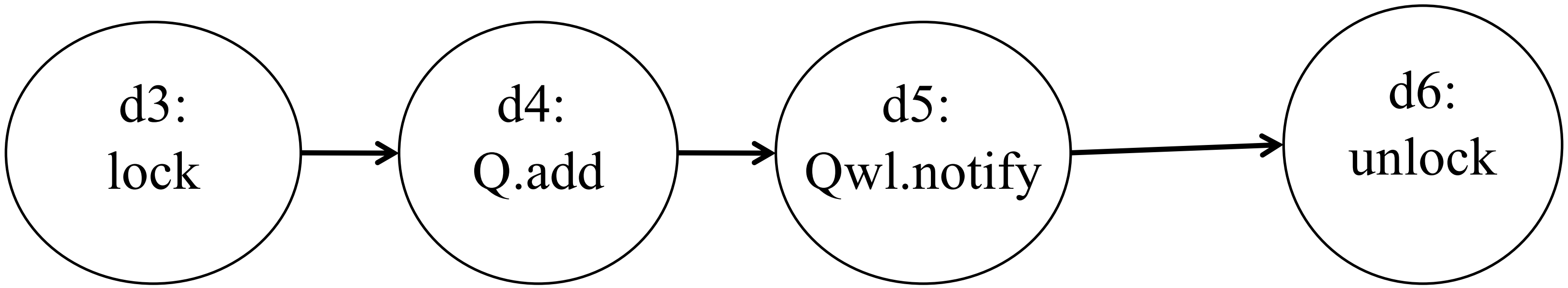
```
// dispatcher
   void run() {
     while (true) {
        Socket con = welcomeSocket.accept();
        lock(Q) {
        Q.add(con);
        notify Q.wait_list; // Q.notifyAll();
        unlock(Q);
     } // end of while
   } // end of run()
```

```
// service thread
void run() {
   while (true) {
      // get next request
      Socket myConn = null;
      synchronized(Q) {
         while (Q.isEmpty()) {
            Q.wait();
         } // end of while
         myConn = Q.remove();
      } // end of sync
       // process request in myConn
   } // end of while
}
```

```
// service thread
void run() {
1. while (true) {
      // get next request
2.    Socket myConn = null;
3.    lock(Q);
4.    while (Q.isEmpty()) {
5.       unlock(Q)
6.       add to Q.wait_list;
7.       yield until marked to wake; //wait
8.       lock(Q);
9.    } // end of while
10.   myConn = Q.remove();
11.   unlock(Q);
      // process request in myConn
    } // end of while
}
```

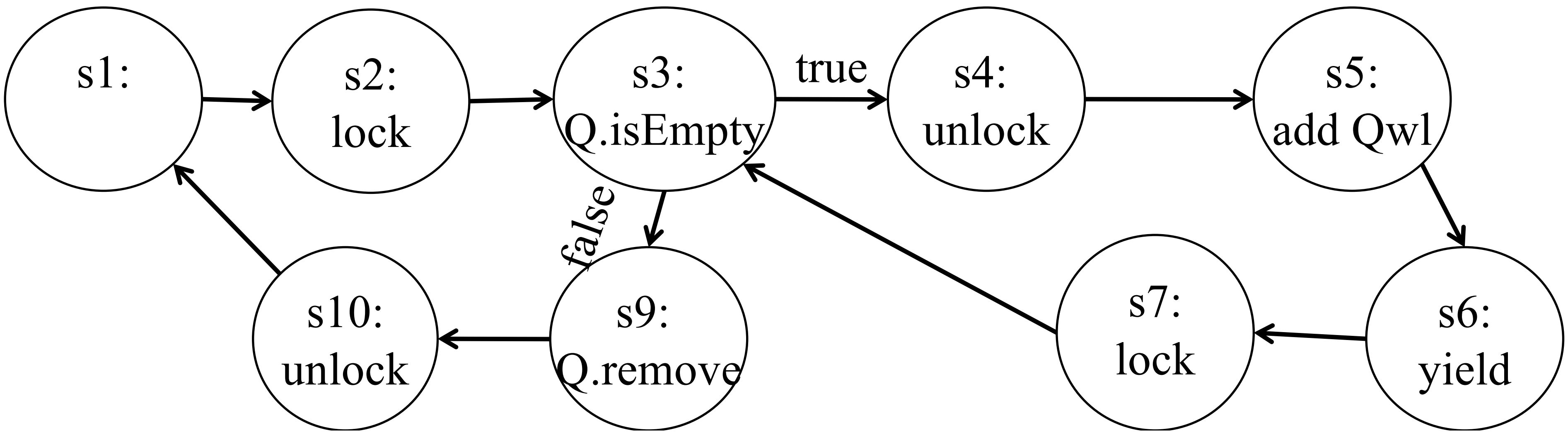
Statements to States (Dispatcher)

```
// dispatcher
void run() {
1.      while (true) {
2.          Socket con = welcomeSocket.accept();
3.          lock(Q) {
4.          Q.add(con);
5.          notify Q.wait_list; // Q.notifyAll();
6.          unlock(Q);
        } // end of while
} // end of run()
```

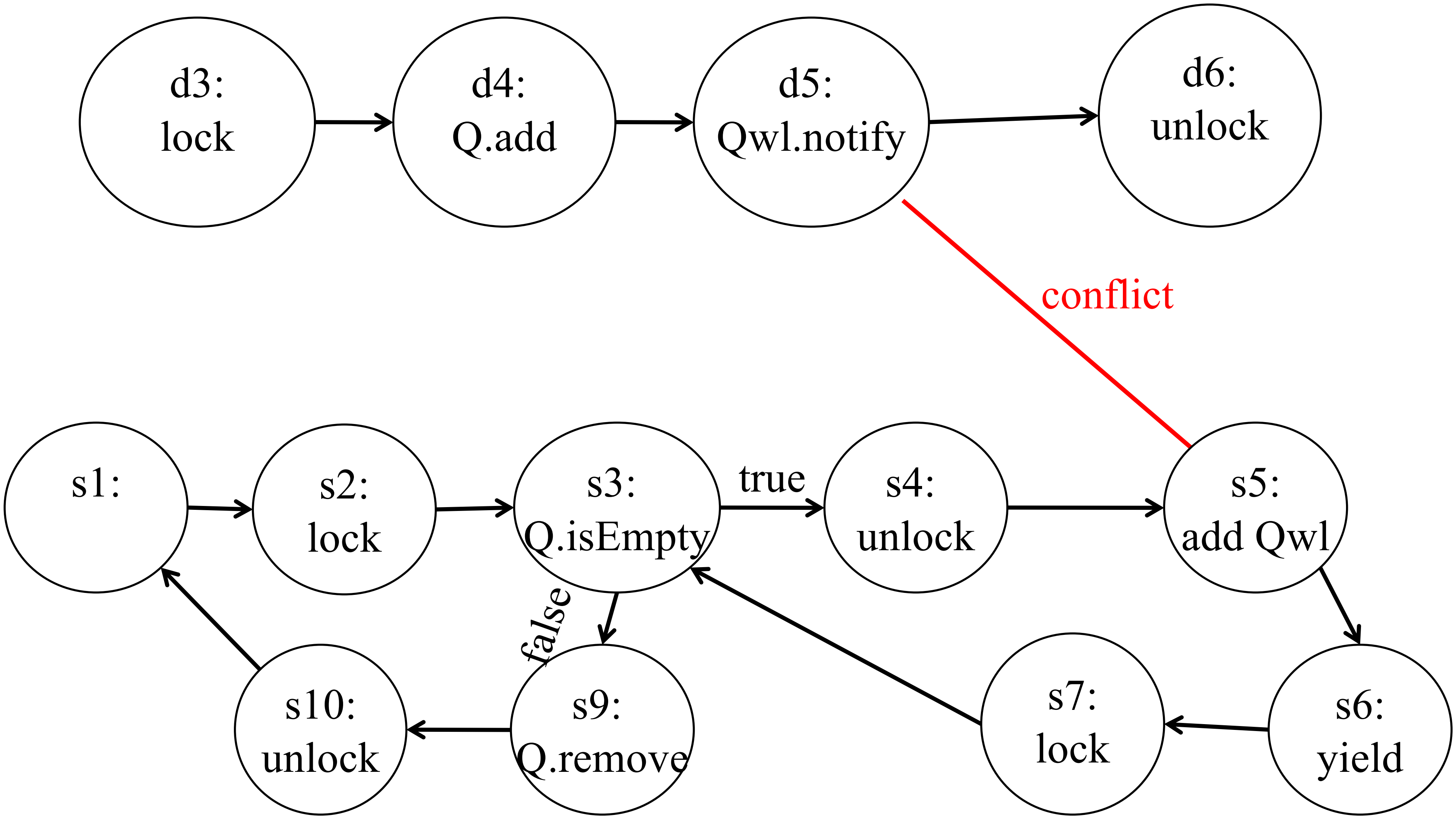


Statements to States (Service)

```
    while (true) {
       // get next request
       Socket myConn = null;
       lock(Q);
       while (Q.isEmpty()) {
          unlock(Q)
          add to Q.wait_list;
          yield; //wait
          lock(Q);
       } // end of while isEmpty
       myConn = Q.remove();
       unlock(Q);
       // process request in myConn
    } // end of while
```



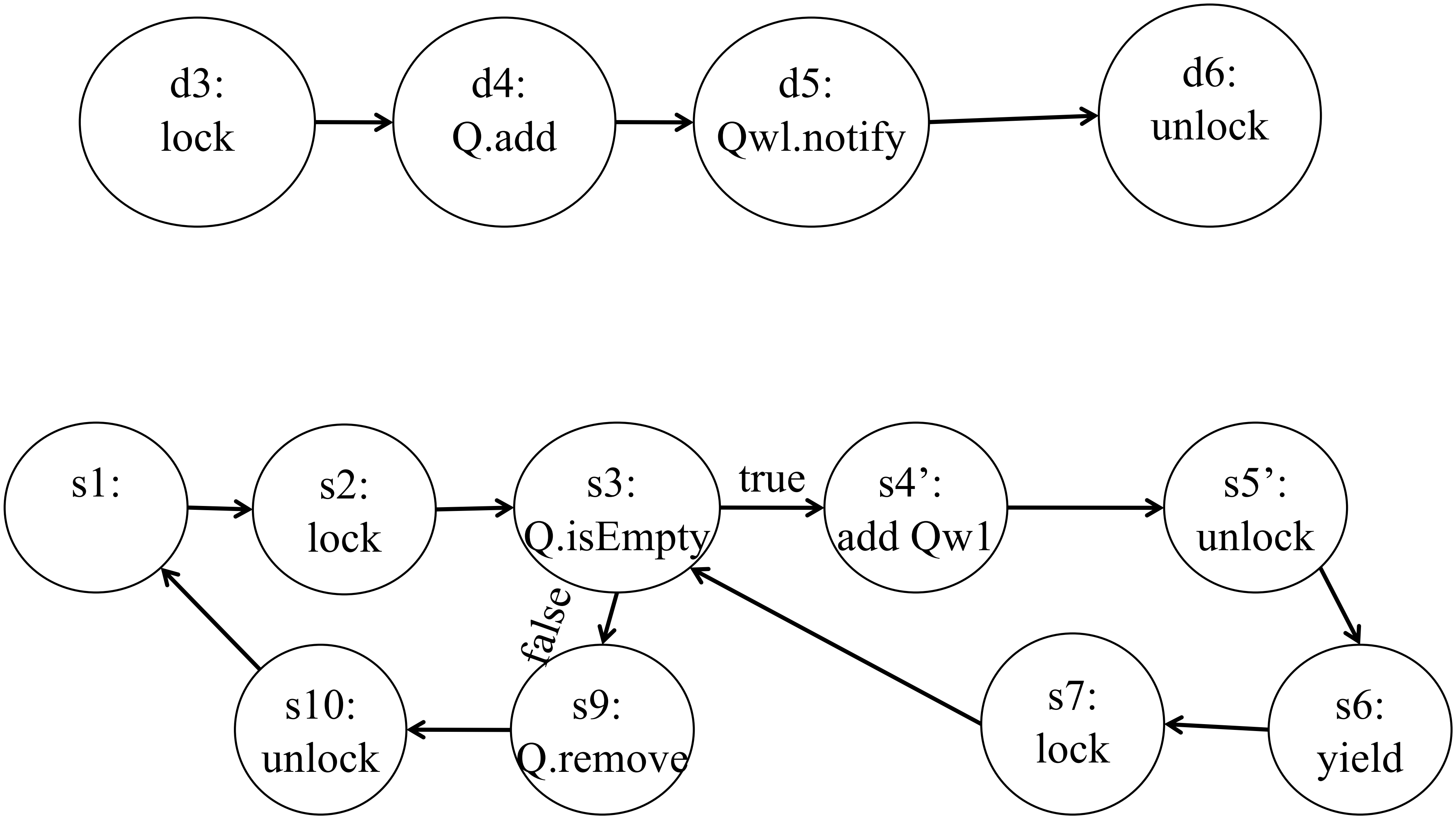
Check Safety



Real Implementation of wait

```
while (true) {
    // get next request
    Socket myConn = null;
    lock(Q);
    while (Q.isEmpty()) {
        add to Q.wait_list;
        unlock(Q); after add to wait list
        yield; //wait
        lock(Q);
    }
    myConn = Q.remove();
    unlock(Q);
    // process request in myConn
} // end of while
```

Check Safety

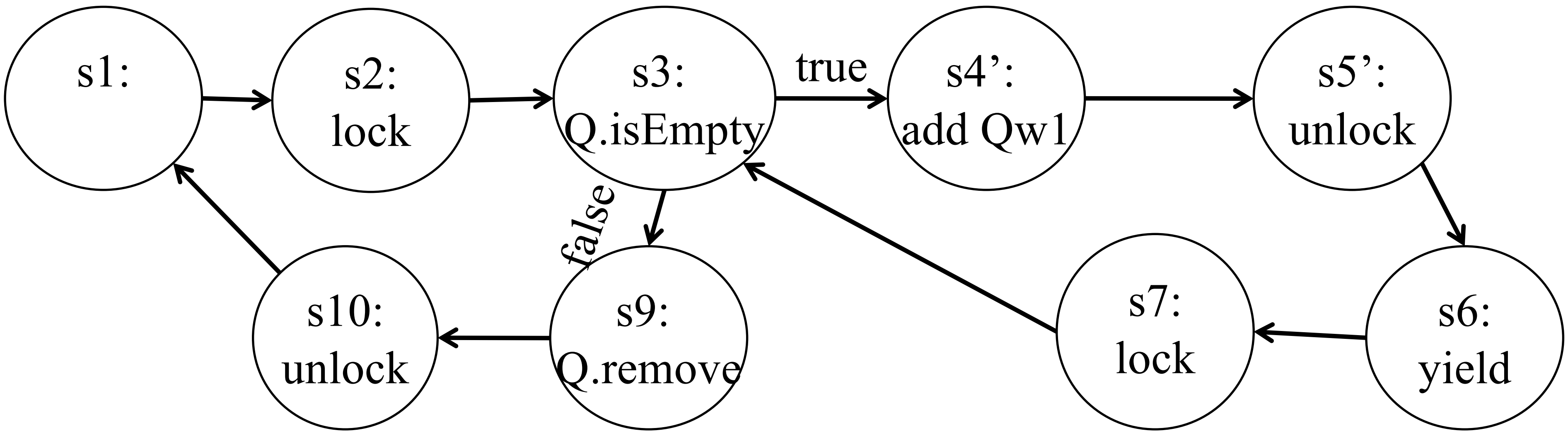


Liveness Properties

- What liveness (progress) properties?
  - dispatcher thread can always add to Q
  - every connection in Q will be processed

Dispatcher Thread Can Always Add to Q

- Assume dispatcher thread is blocked
- Suppose Q is not empty, then each iteration removes one element from Q
- In finite number of iterations, all elements in Q are removed and all service threads unlock and block
  - Need to assume each service takes finite amount of time (bound by a fixed T_0)



Each Connection in Q is Processed

- Cannot be guaranteed unless
  - there is fairness in the thread scheduler, or
  - put a limit on Q size to block the dispatcher thread

Summary: Program Correctness Analysis

- Safety
  - No read/write; write/write conflicts
    - holding lock Q before reading or modifying shared data Q and Q.wait_list
  - Q.remove() is not on an empty queue
- Liveness (progress)
  - dispatcher thread can always add to Q
  - every connection in Q will be processed
- Fairness
  - For example, in some settings, a designer may want the threads to share load equally

Use Java ThreadPoolExecutor

```java
server = new ServerSocket(port);
System.out.println("Time server listens at port: " + port);

// Create Java Executor Pool
TimeServerHandlerExecutePool myExecutor
   = new TimeServerHandlerExecutePool(50, 10000);


Socket socket = null;
while (true) {
   socket = server.accept();
   myExecutor.execute(new TimeServerHandler(socket));
} // end of while
```

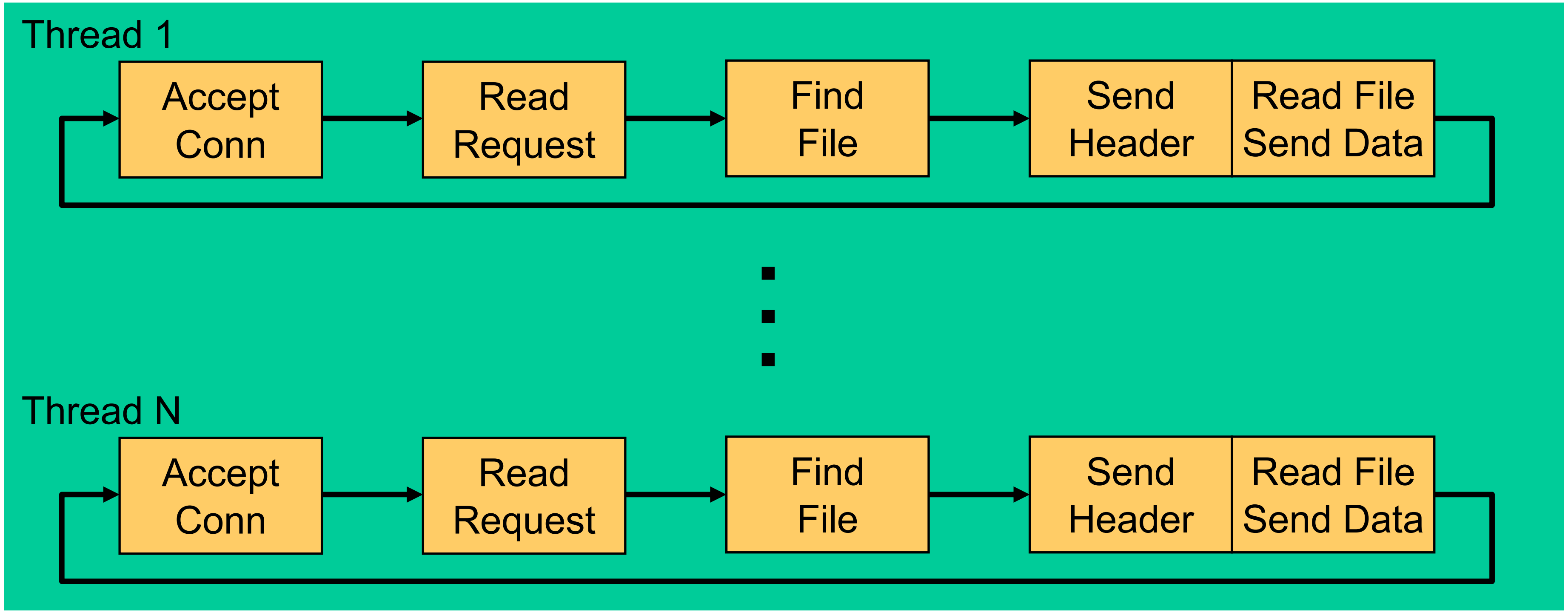
Use Java ThreadPoolExecutor

```java
public class TimeServerHandlerExecutePool {

    private ExecutorService executor;

    public  TimeServerHandlerExecutePool(int maxPoolSize, int queueSize) {
       executor = new ThreadPoolExecutor(
                        Runtime.getRuntime().availableProcessors(),
                        maxPoolSize,
                        120L, TimeUnit.SECONDS,
                        new ArrayBlockingQueue<java.lang.Runnable>(queueSize)
                   );
    }

    public  void execute(java.lang.Runnable task) {
        executor.execute(task);
    }
}
```

For Java ThreadPoolExecutor scheduling algorithm, see:
https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ThreadPoolExecutor.html

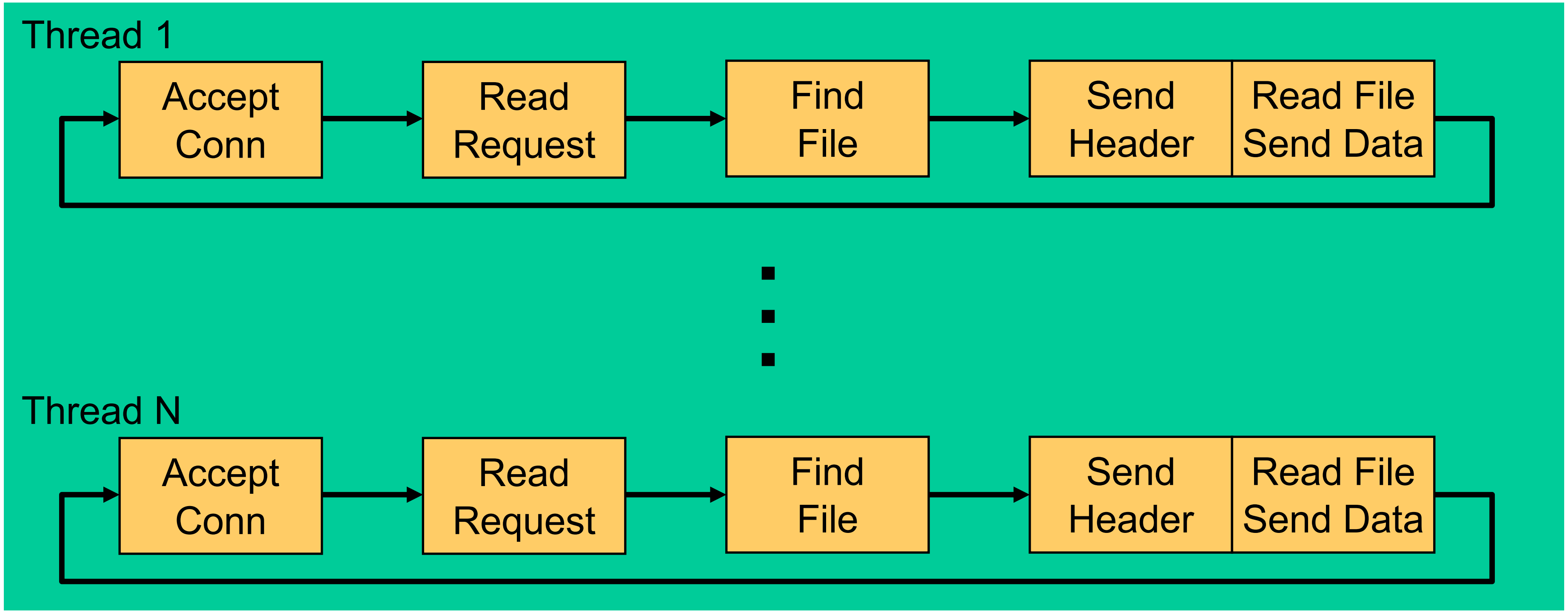
Summary: Thread-Based Network Server

- Multiple threads (execution sequences) offer multiple execution sequences => blocking causes only one thread being blocked
- Intuitive (sequential) programming model
- Shared address space simplifies optimizations



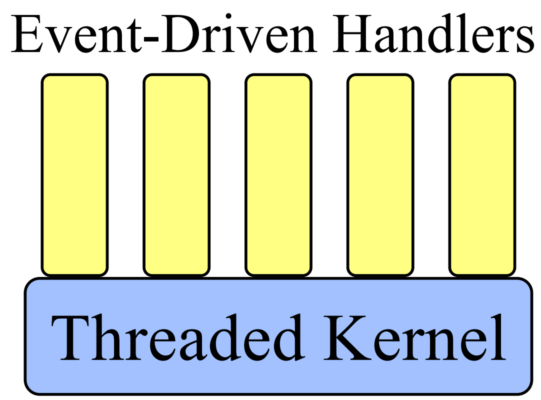
Summary: Thread-Based Network Server

- Thread creation overhead
- Thread synchronization overhead
  - Need to handle synchronization -> otherwise race condition
  - Handle synchronization -> Overhead, complexity (e.g., wait/notify, deadlock)
  - Thread size (how many threads) difficult to tune
- Still cannot handle well the large-number of long, idle connections problem (why?)



Should You Use Threads?

- Typically avoid threads for io
  - Use event-driven, not threads, for GUIs, servers, distributed systems.
- Use threads where true CPU concurrency is needed.
  - Where threads needed, isolate usage in threaded application kernel: keep most of code single-threaded.



[Ousterhout 1995]