Network Applications: High-performance Server Design

Qiao Xiang

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

10/26/2021

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 servers
  - Threaded design
    - Per-request thread
    - Thread pool
      - Busy wait
      - Wait/notify
  - Asynchronous design

Admin

- Lab assignment 2 due on Oct. 28
- Lab assignment 4 (Web server - part 2) to be posted later this week.

- Date for exam 1?
  - Nov. 11 (4:40-6:20pm, lab class), 12 (7:10-8:50pm), 13?

Recap: Thread-Based Network Servers

- Why: blocking operations; threads (execution sequences) so that only one thread is blocked
- How:
  - 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

Recap: Example

- try
  - ShareQ/Server.java
  - ShareQ/ServiceThread.java
  - Use top to observer the CPU utilization

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?

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*

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?

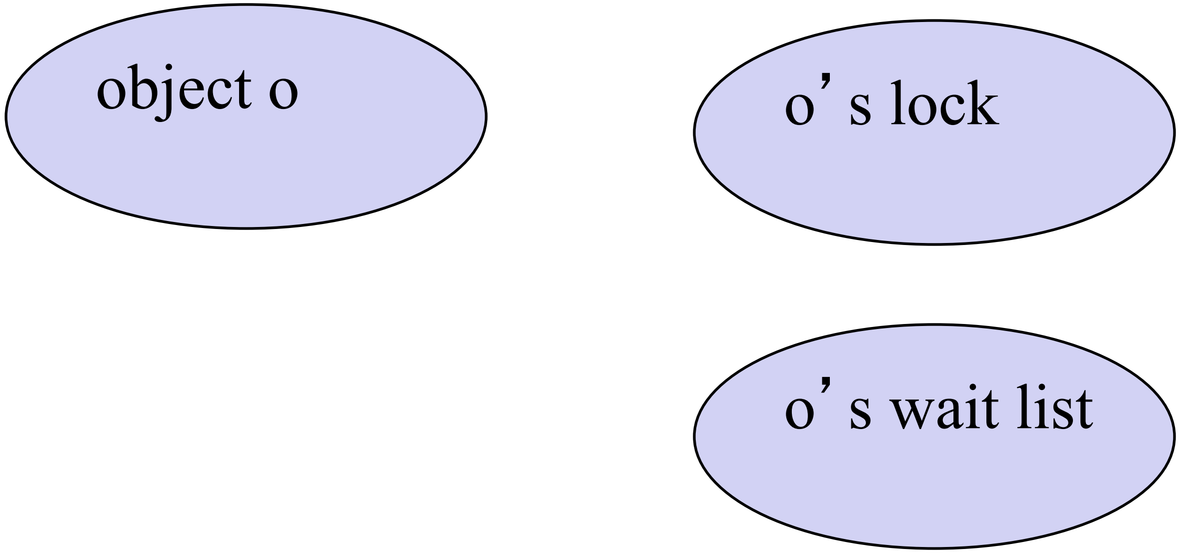
Solution: Suspension

```
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

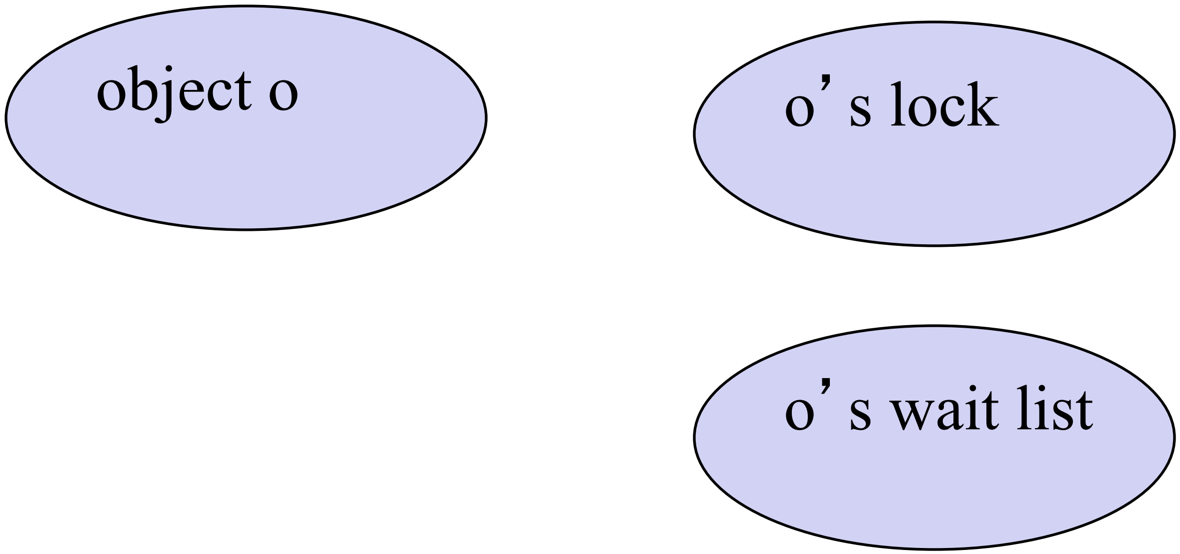
Wait-sets and Notification

- 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

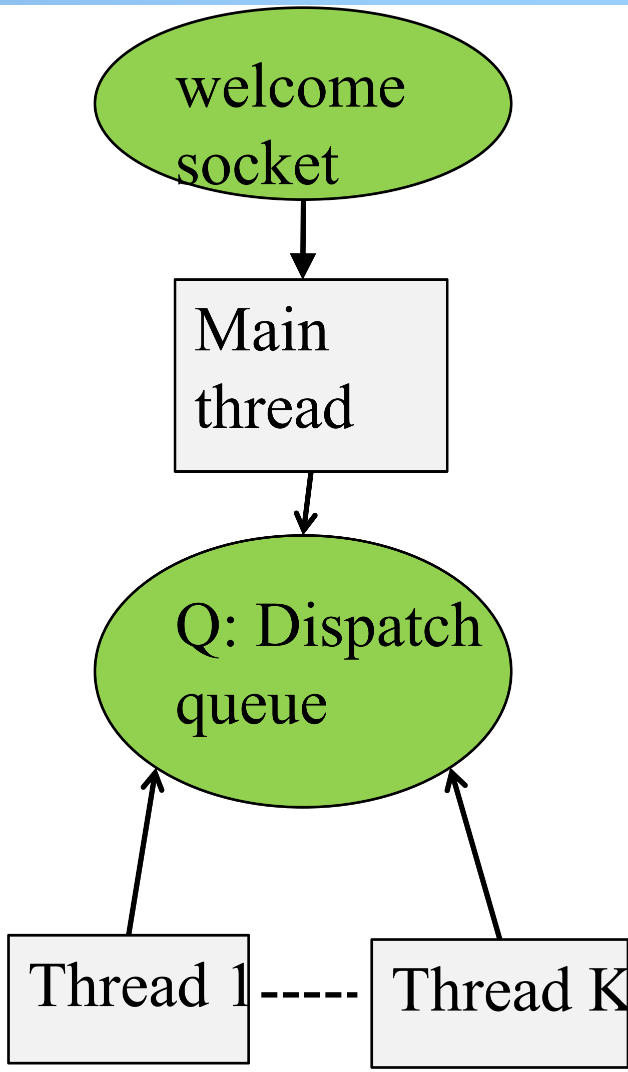
Notification

- 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
}
```



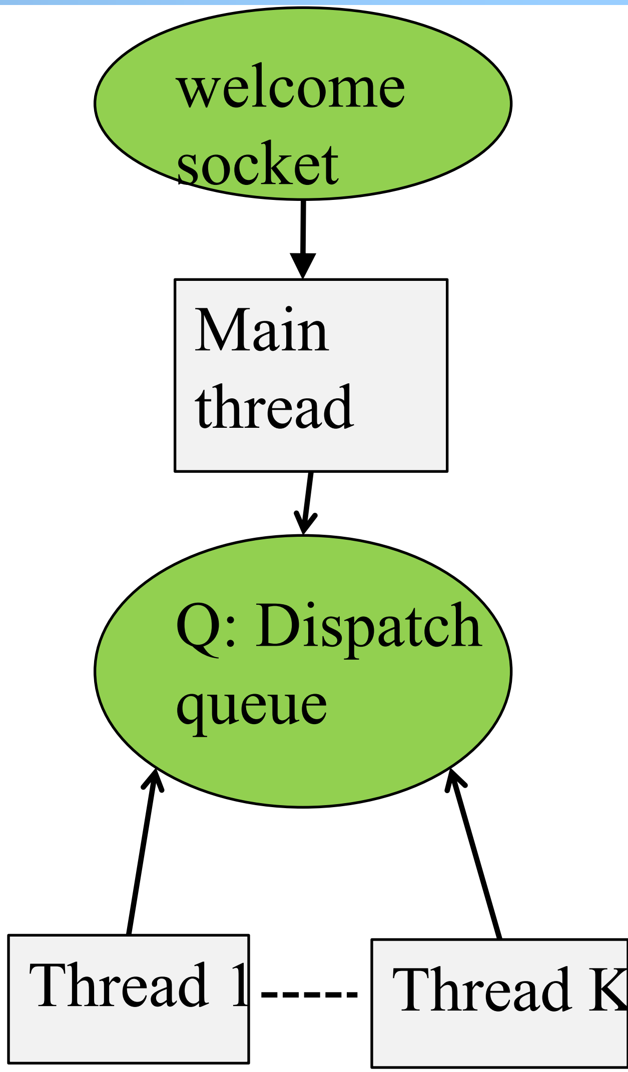
Worker

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

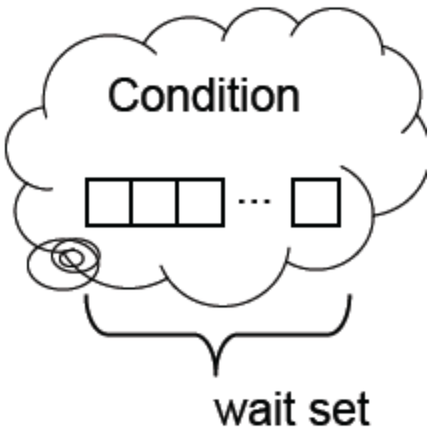
- Use of wait(), notifyAll() and notify() similar to
  - Condition queues of classic Monitors
  - Condition variables of POSIX PThreads API
  - In C# it is called Monitor (http://msdn.microsoft.com/en-us/library/ms173179.aspx)

- 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()
```

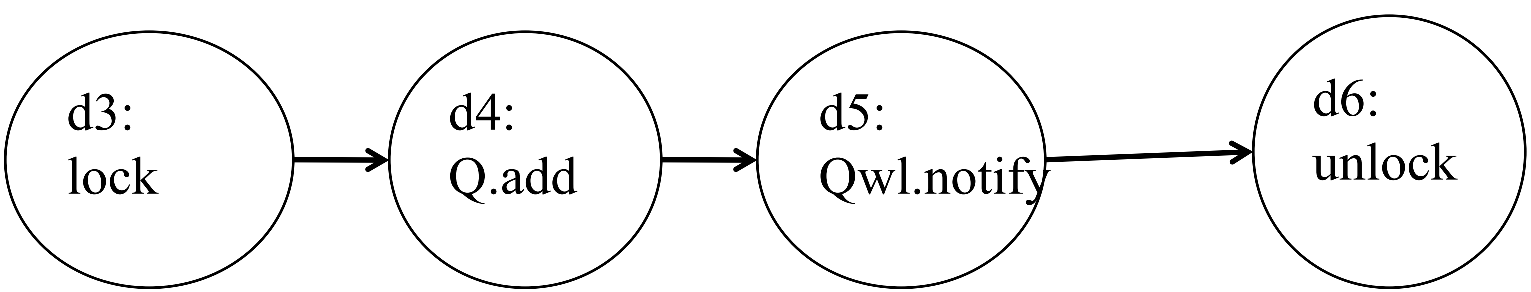
```
// dispatcher
   void run() {
     while (true) {
        Socket con = welcomeSocket.accept();
        lock(Q) {
        Q.add(con);
        notify Q.wait_list; // Q.notifyAll();
        unlock(Q);
     } // end of while
   } // end of run()
```

```
// 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
}
```

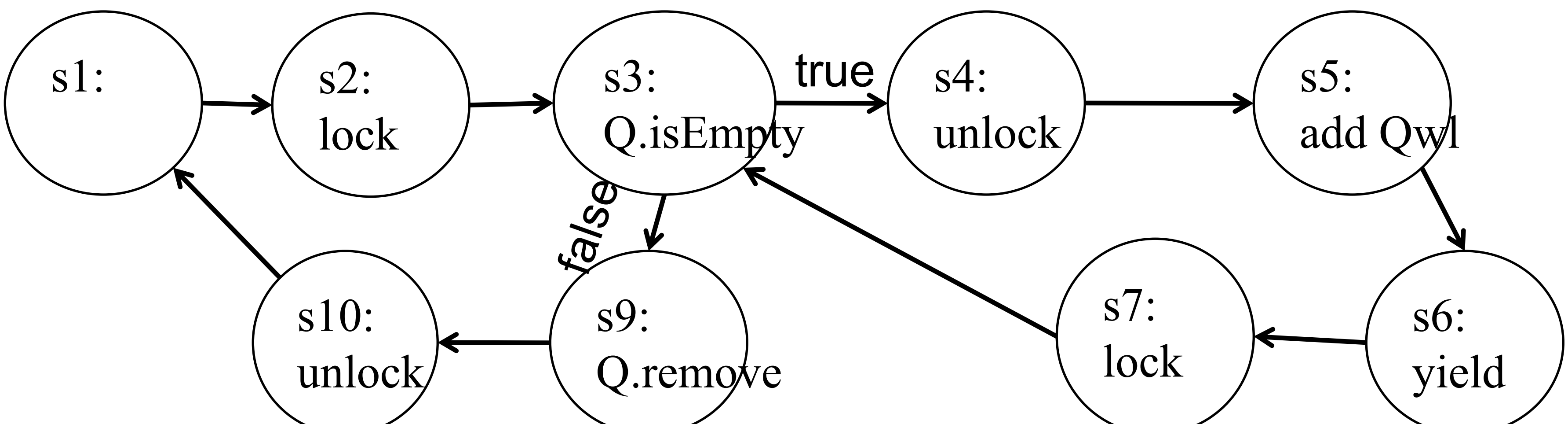
Statements to States (Dispatcher)

```
// 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()
```

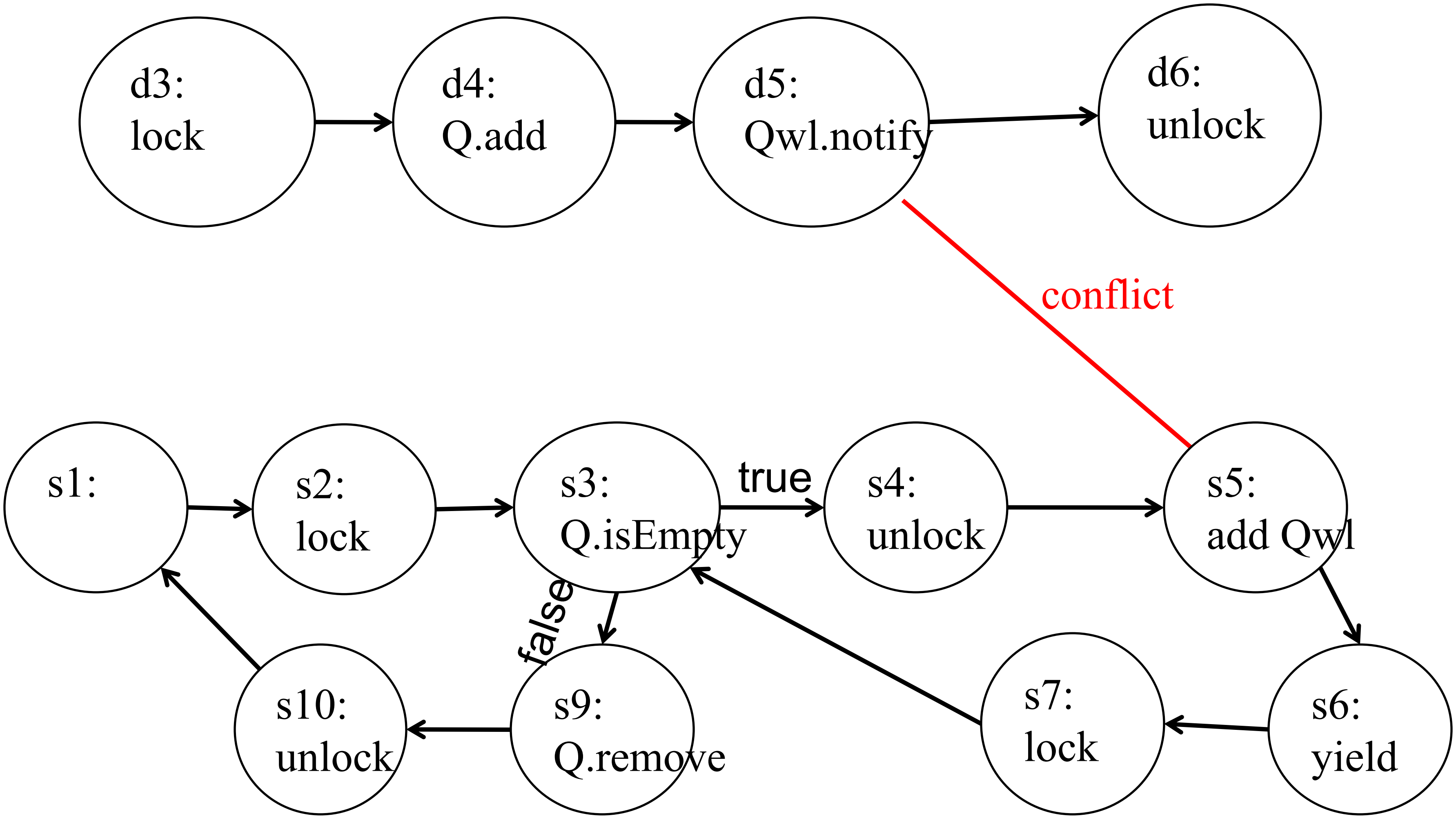


Statements to States (Service)

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



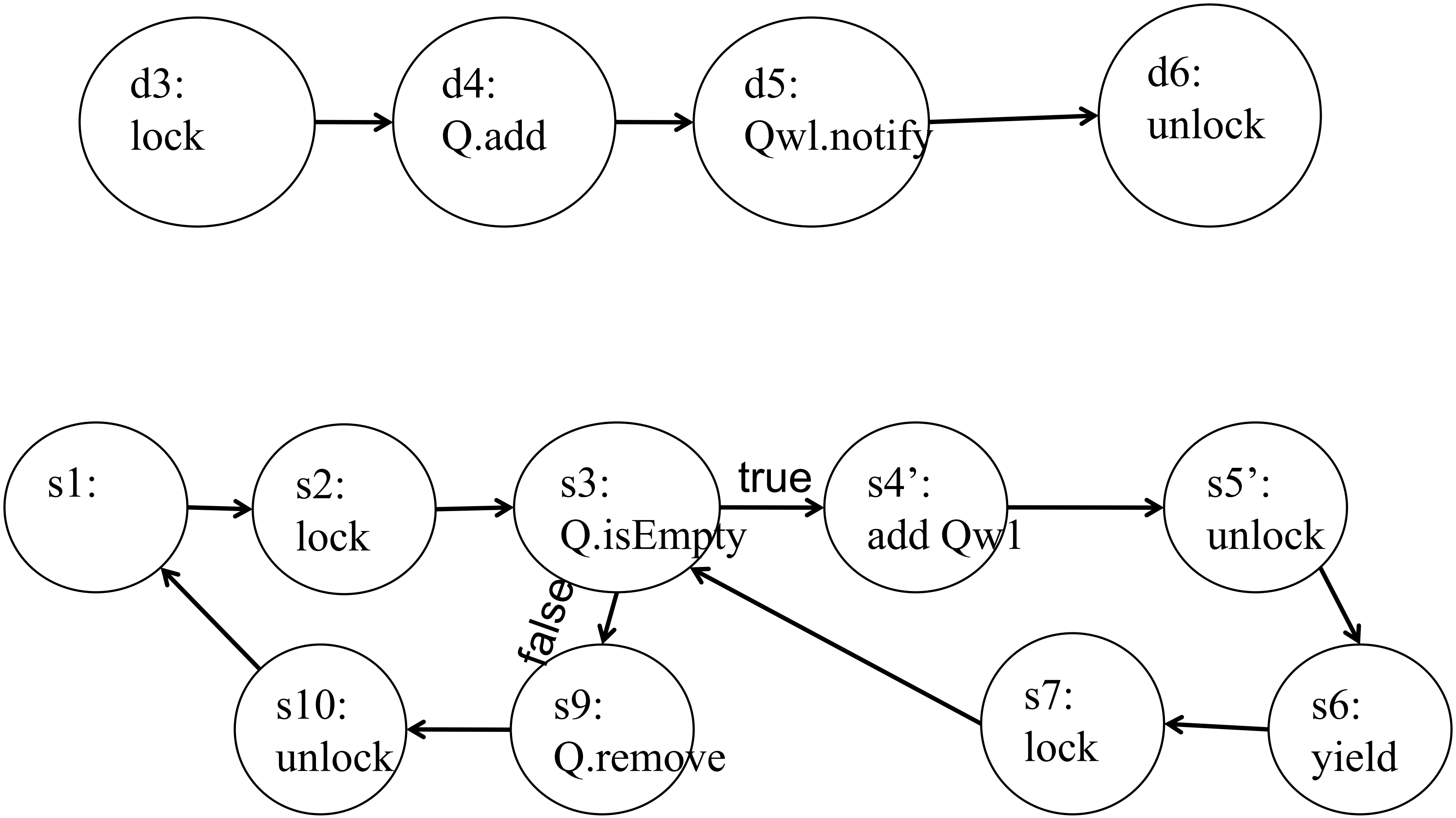
Check Safety



Real Implementation of `wait`

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

Check Safety

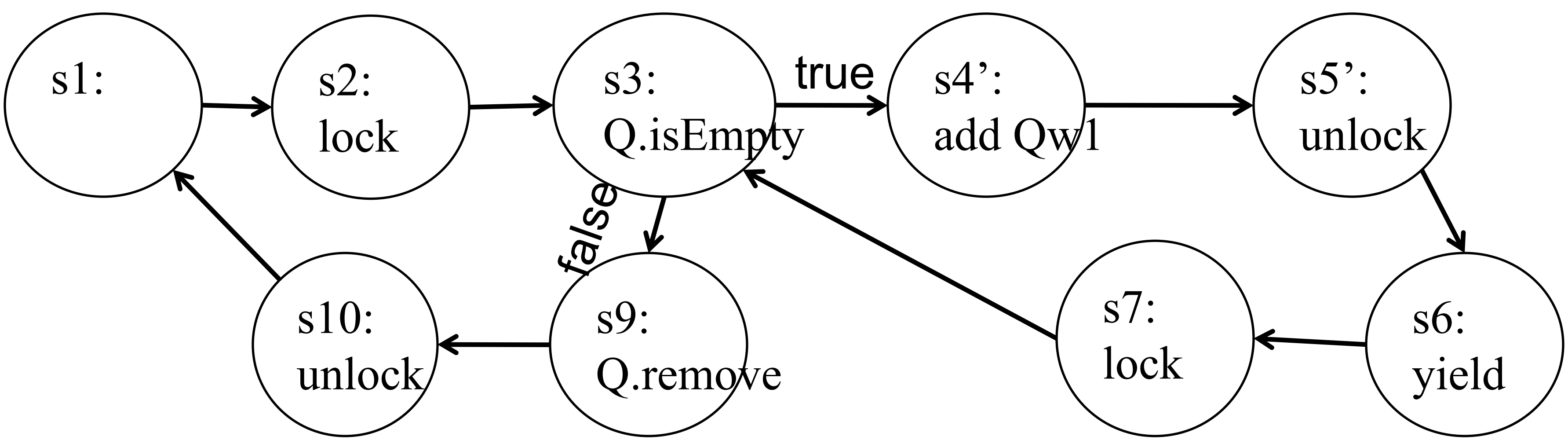


Liveness Properties

- 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

```
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
```

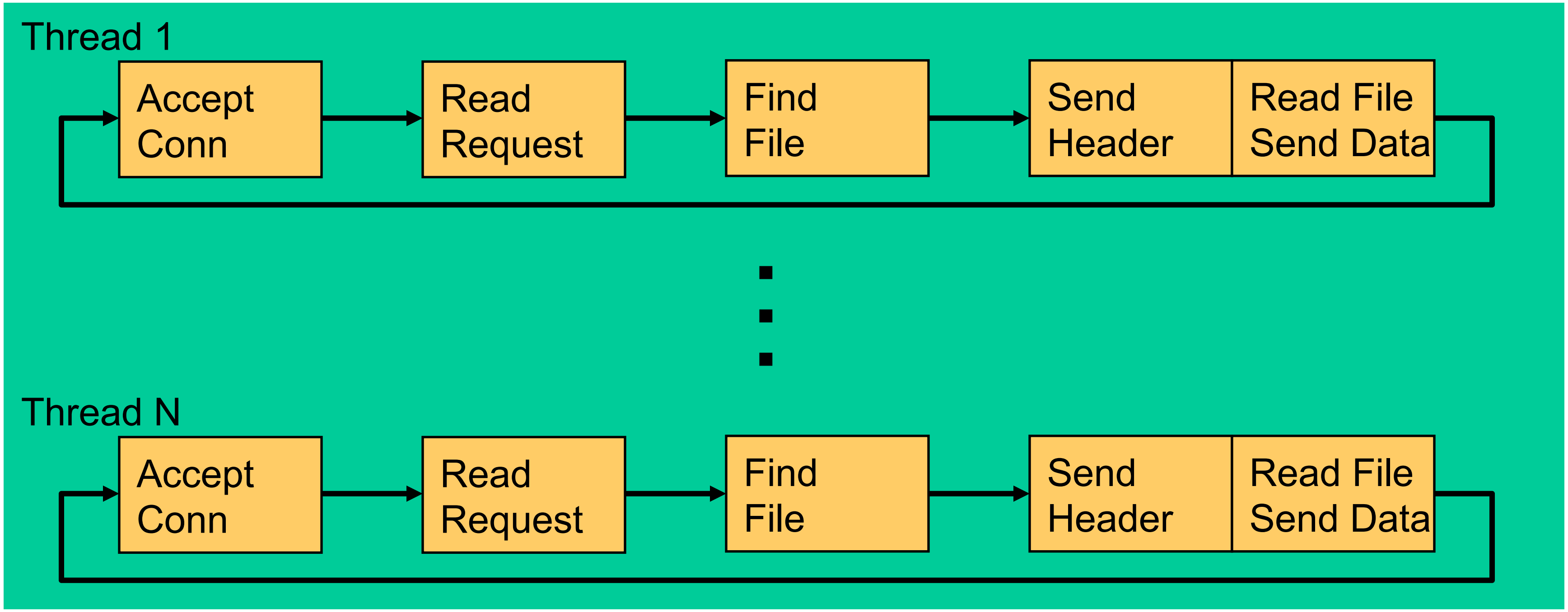
Use Java ThreadPoolExecutor

```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

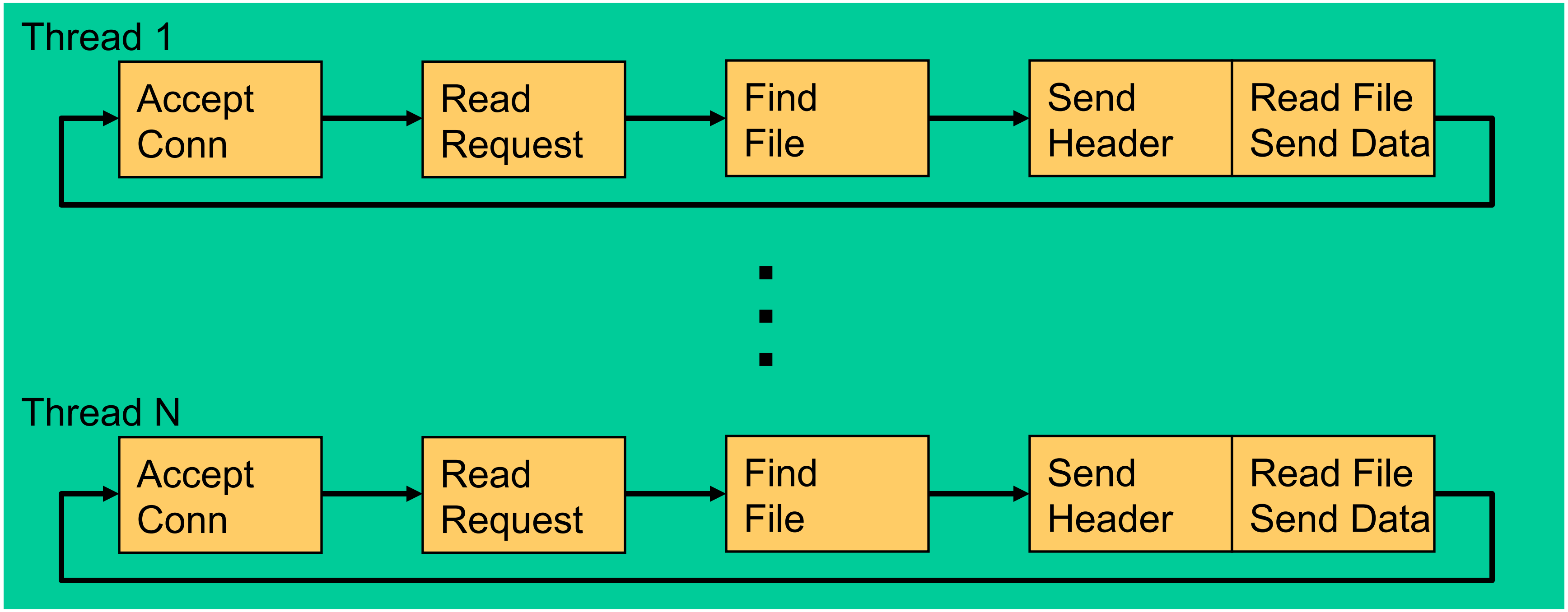
Summary: Thread-Based Network Server

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



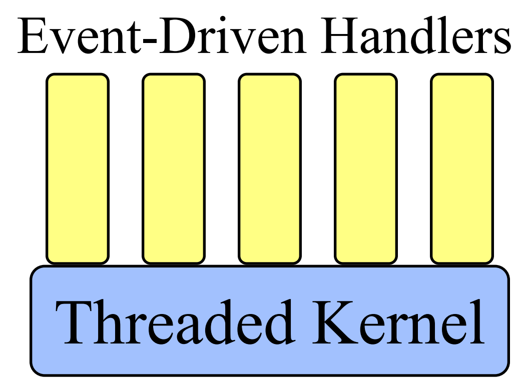
Summary: Thread-Based Network Server

- 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.



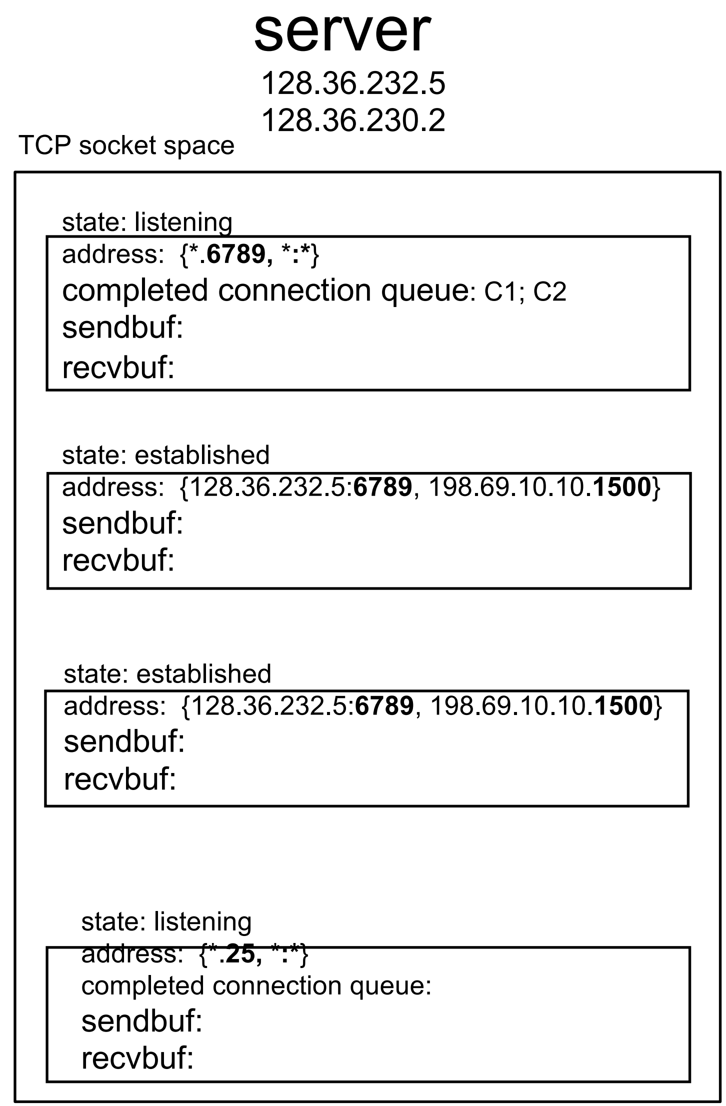
[Ousterhout 1995]

Outline

- Admin and recap
- High performance servers
  - Threaded design
    - Per-request thread
    - Thread pool
      - Busy wait
      - Wait/notify
  - *Select-multiplexing server design*

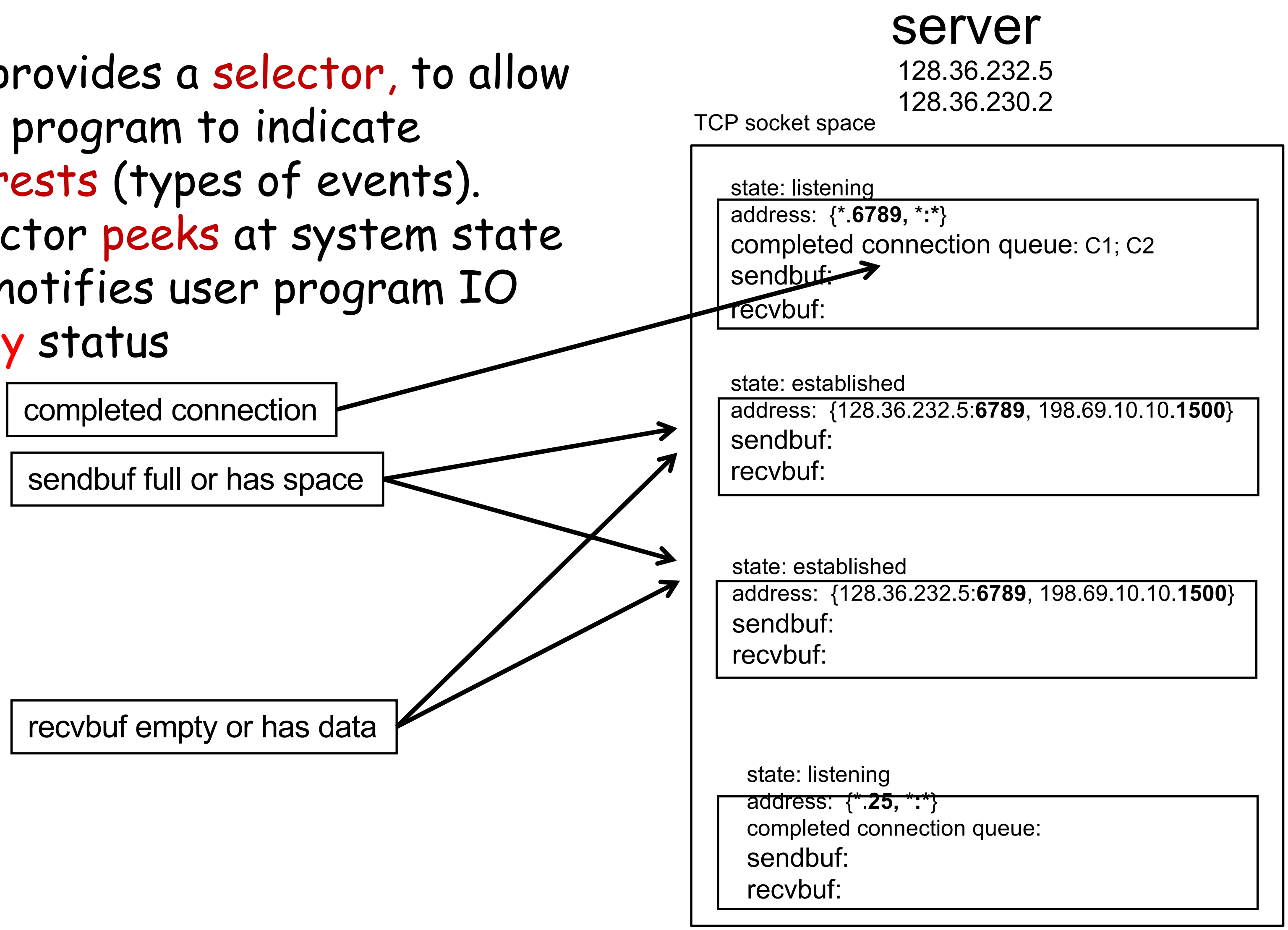
Big Picture: Built on top of Lower-Layer OS Services/Abstractions

- Blocking IO
  - if not ready, block calling thread
  - get data, copy to user space;
- Non-blocking IO (set socket NON_BLOCK) stream
  - return error if not ready (EWOULDBLOCK)
  - after ready, call, OS copy
- Selector (channel) IO [Java NIO; Linux epoll; FreeBSD/Mac kqueue]
  - monitors multiple IO descriptors
- Async IO (Java 7 aio; Linux 2.5 first and then 2.6)
  - aio_read() // after copy to user space
- DMA based (later in course)



Selector Multiplexing Basic Idea

- OS provides a selector, to allow user program to indicate interests (types of events). Selector peeks at system state and notifies user program IO ready status



Background: Linux epoll System Calls

- "… monitoring multiple files to see if IO is possible on any of them…" -- man 7 epoll
- Three basic system calls
  - epoll_create1(2) – create new epoll instance
  - epoll_ctl(2) – manage file descriptors regarding the interested-list
  - epoll_wait(2) – main workhorse, block tasks until IO becomes available
- See SelectEchoServer/epoll_examples.c

Core data structure

```
typedef union epoll_data {
    void        *ptr;
    int          fd;
    uint32_t     u32;
    uint64_t     u64;
} epoll_data_t;

struct epoll_event {
    uint32_t     events;      /* Epoll events */
    epoll_data_t data;        /* User data variable */
};
```

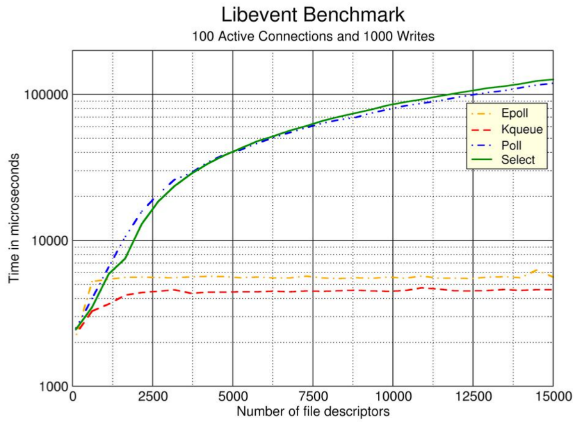
The *data* member of the *epoll_event* structure specifies data that the kernel should save and then return (via epoll_wait(2)) when this file descriptor becomes ready.

https://events19.linuxfoundation.org/wp-content/uploads/2018/07/dbueso-oss-japan19.pdf

Background: Linux epoll Internal

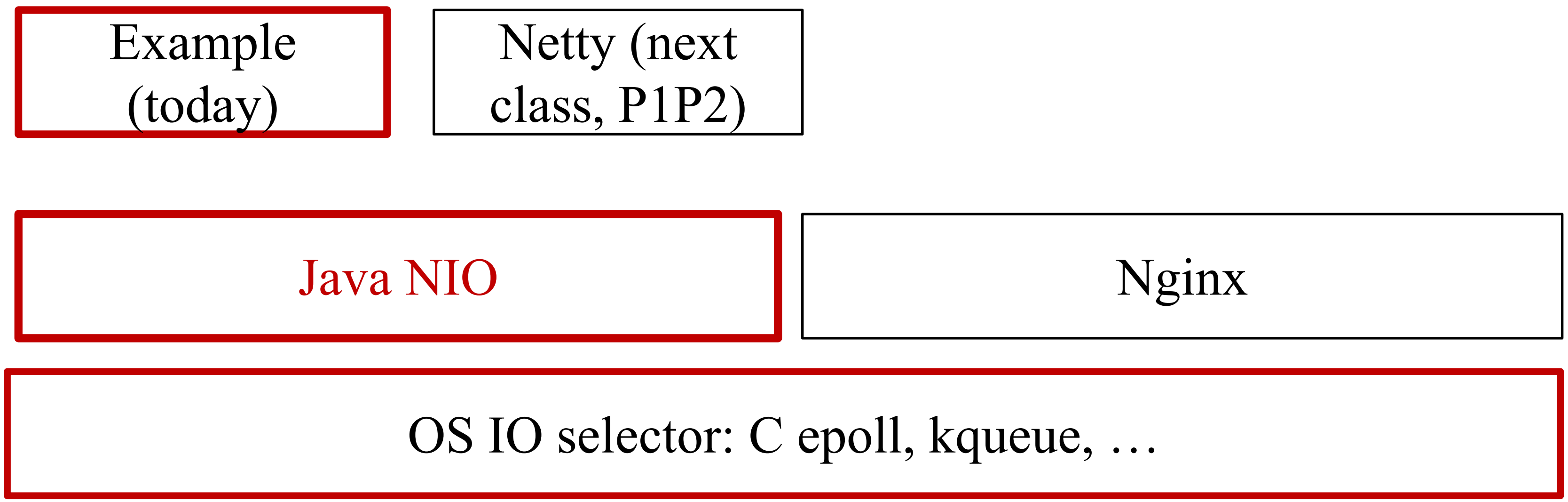
- Before epoll, select/poll is "stateless" but then need O(n) complexity; epoll separates setup and waiting phrases to reach O(n_ready)

- Details see: https://man7.org/linux/man-pages/man7/epoll.7.html



https://events19.linuxfoundation.org/wp-content/uploads/2018/07/dbueso-oss-japan19.pdf

Big Picture



Example (today)
Netty (next class, P1P2)
Java NIO
Nginx
OS IO selector: C epoll, kqueue, …

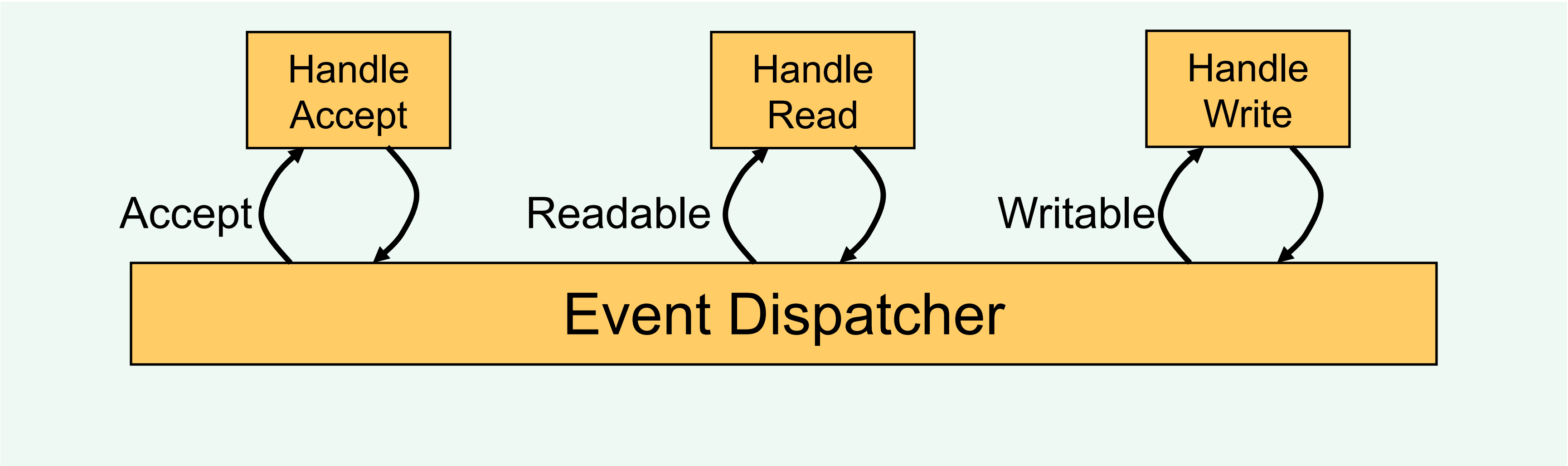
Basic Idea: Asynchronous Initiation and Callback

- Issue of only peek:
  - Cannot handle initiation calls (e.g., read file, initiate a connection by a network client)

- Idea: asynchronous initiation (e.g., aio_read) and program specified completion handler (callback)
  - Also referred to as proactive (Proactor) nonblocking

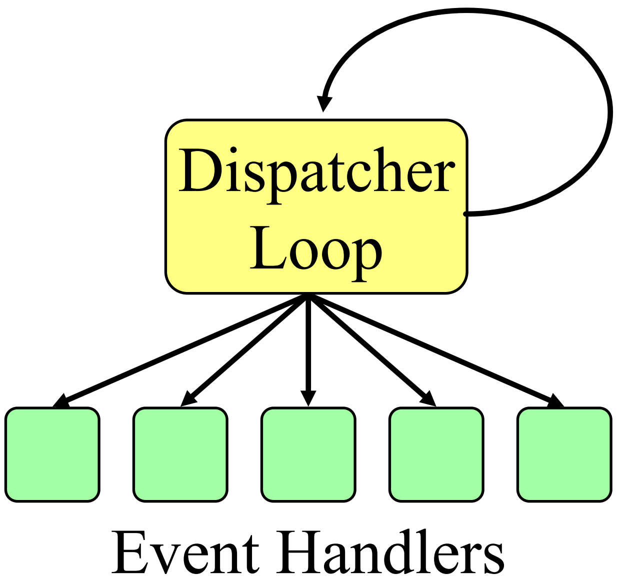
- We focus more on multiplexed, reactive design

Multiplexed, Reactive Server Architecture



- Program registers events (e.g., acceptable, readable, writable) to be monitored and a handler to call when an event is ready
- An infinite dispatcher loop:
  - Dispatcher asks OS to check if any ready event
  - Dispatcher calls (multiplexes) the registered handler of each ready event/source
    - Handler should be non-blocking, to avoid blocking the event loop

Multiplexed, Non-Blocking Network Server




```
// clients register interests/handlers
   on events/sources
while (true)  {
   - ready events = select()
        /* or selectNow(),
           or  select(int timeout) to
           check ready events from the
           registered interests */

  - foreach ready event {
       switch event type:
       accept: call accept handler
       readable: call read handler
       writable: call write handler
     }

  - handle other events
}
```

Main Abstractions

- Main abstractions of multiplexed IO:
  - Channels: represent connections to entities capable of performing I/O operations;
  - Selectors and selection keys: selection facilities;
  - Buffers: containers for data.

- More details see https://docs.oracle.com/javase/8/docs/api/java/nio/package-summary.html

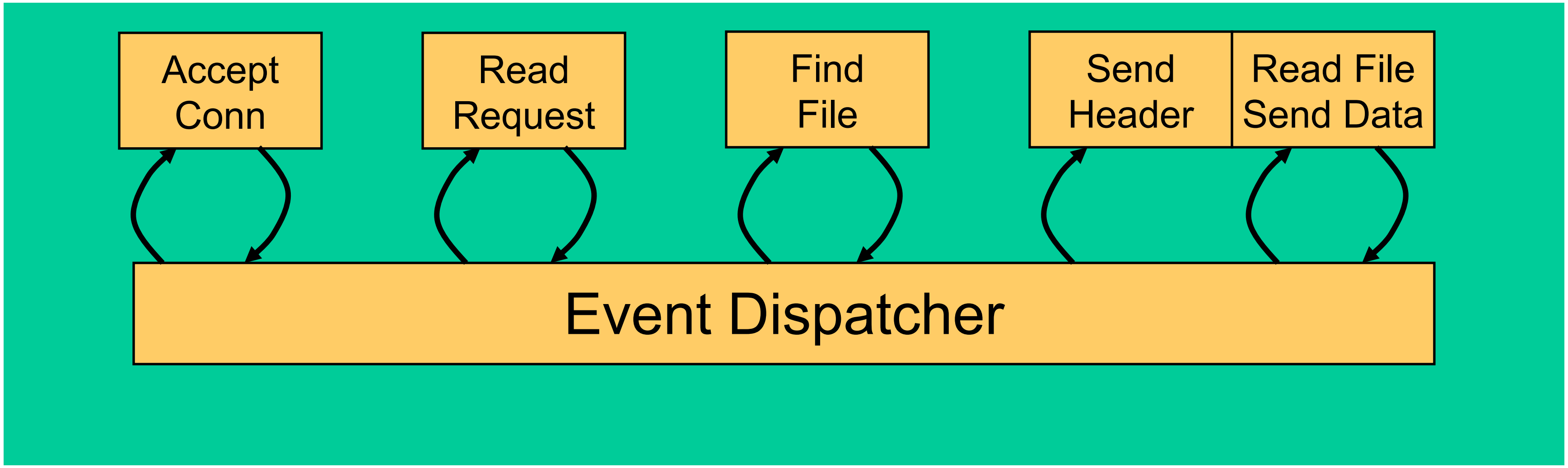
Multiplexed (Selectable), Non-Blocking Channels

SelectableChannel	A channel that can be multiplexed
DatagramChannel	A channel to a datagram-oriented socket
Pipe.SinkChannel	The write end of a pipe
Pipe.SourceChannel	The read end of a pipe
ServerSocketChannel	A channel to a stream-oriented listening socket
SocketChannel	A channel for a stream-oriented connecting socket

- Use `configureBlocking(false)` to make a channel non-blocking
- Note: Java `SelectableChannel` does not include file I/O

Selector

- The class `Selector` is the base of the multiplexer/dispatcher
- Constructor of Selector is protected; create by invoking the `open` method to get a selector (why?)



Selector and Registration

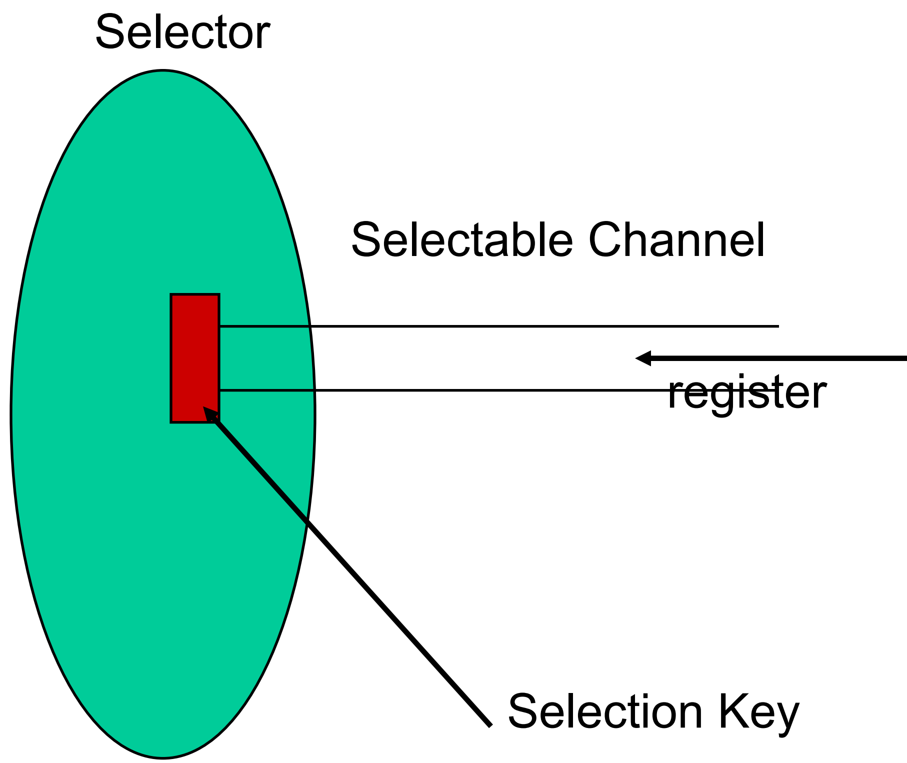
- A selectable channel registers events to be monitored with a `selector` with the `register` method

- The registration returns an object called a `SelectionKey`:

```
SelectionKey key =
    channel.register(selector, ops);
```

Java Selection I/O Structure

- A `SelectionKey` object stores:
  - interest set: events to check: `key.interestOps(ops)`
  - ready set: after calling select, it contains the events that are ready, e.g. `key.isReadable()`
  - an attachment that you can store anything you want `key.attach(myObj)`



Checking Events

- A program calls `select` (or `selectNow()`, or `select(int timeout)`) to check for ready events from the registered SelectableChannels
  - Ready events are called the selected key set

```
selector.select();
Set readyKeys = selector.selectedKeys();
```

- The program iterates over the selected key set to process all ready events

Dispatcher using Select

```
while (true)  {
  - selector.select()
  - Set readyKeys = selector.selectedKeys();

  - foreach key in readyKeys {
        switch event type of key:
         accept: call accept handler
         readable: call read handler
         writable: call write handler
    }
}
```

I/O in Java: ByteBuffer

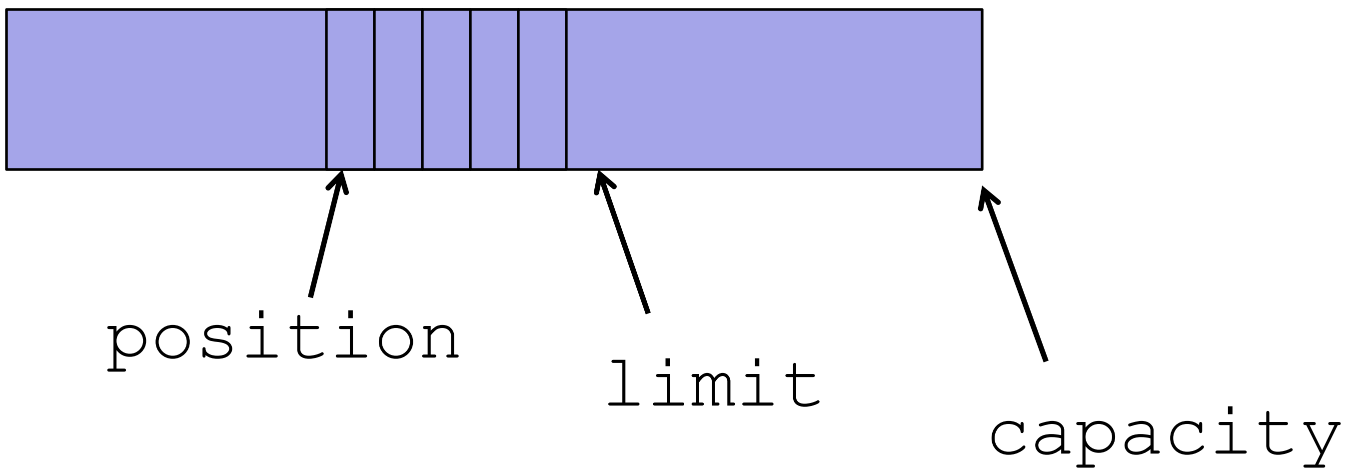
- Java SelectableChannels typically use ByteBuffer for read and write
  - channel.read(byteBuffer);
  - channel.write(byteBuffer);

- ByteBuffer is a powerful class that can be used for both read and write
- It is derived from the class Buffer
- All reasonable network server design should have a good buffer design

Java ByteBuffer Hierarchy

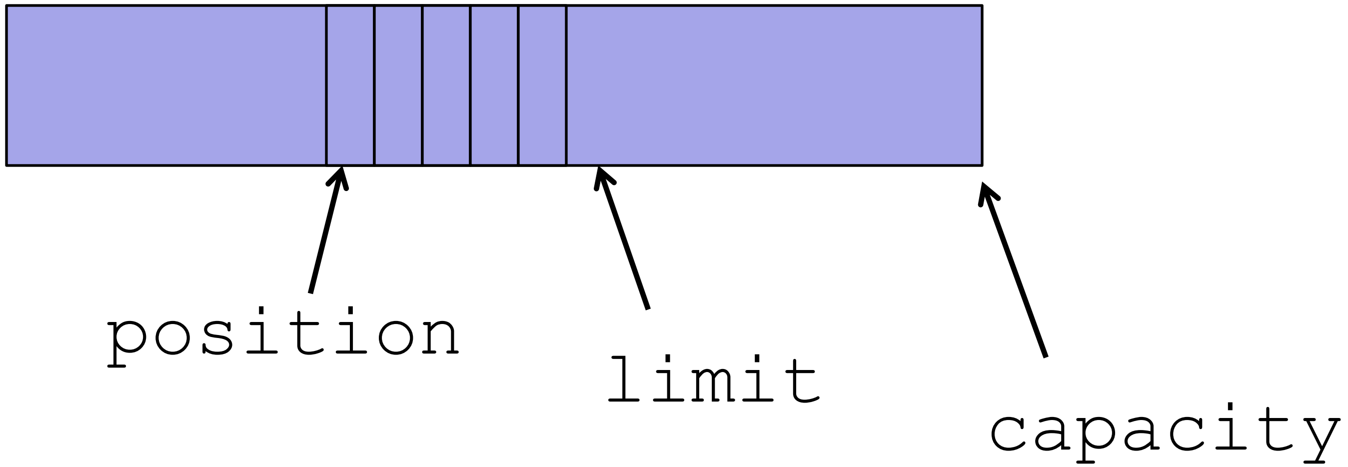
Buffers	Description
Buffer	Position, limit, and capacity; clear, flip, rewind, and mark/reset
ByteBuffer	Get/put, compact, views; allocate, wrap
MappedByteBuffer	A byte buffer mapped to a file
CharBuffer	Get/put, compact; allocate, wrap
DoubleBuffer	' '
FloatBuffer	' '
IntBuffer	' '
LongBuffer	' '
ShortBuffer	' '

Buffer (relative index)



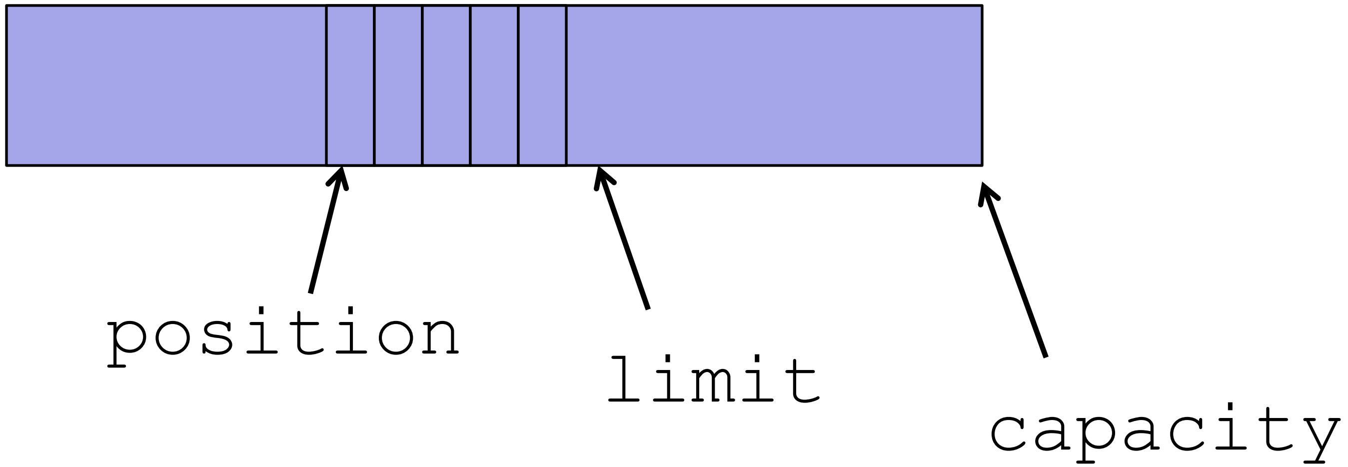
- Each Buffer has three numbers: position, limit, and capacity
  - Invariant: 0 <= *position* <= *limit* <= *capacity*
- `Buffer.clear(): position = 0; limit=capacity`

channel.read(Buffer)



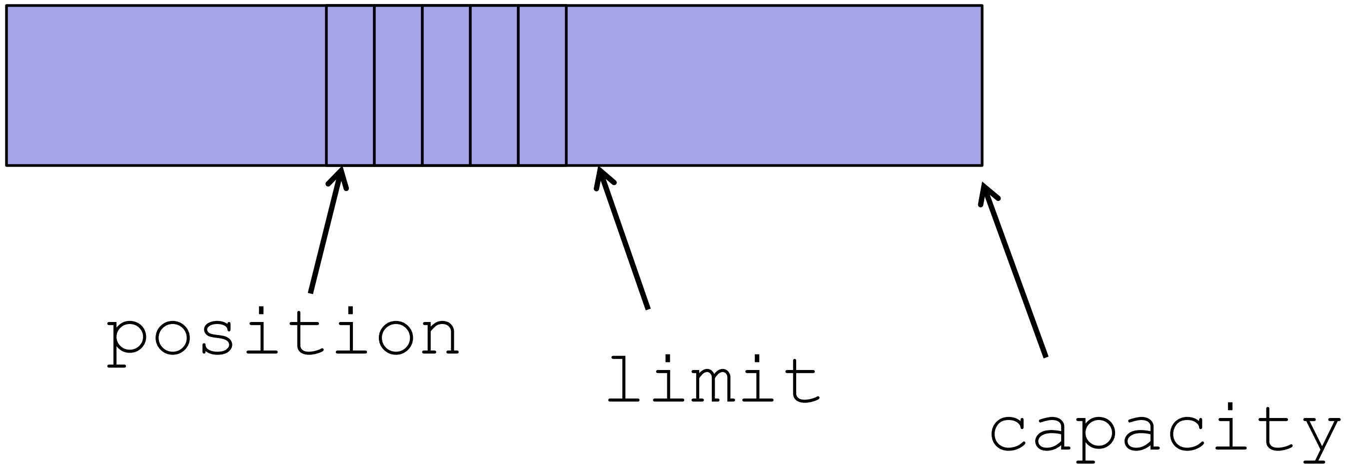
- Put data into Buffer, starting at `position`, not to reach `limit`

channel.write(Buffer)



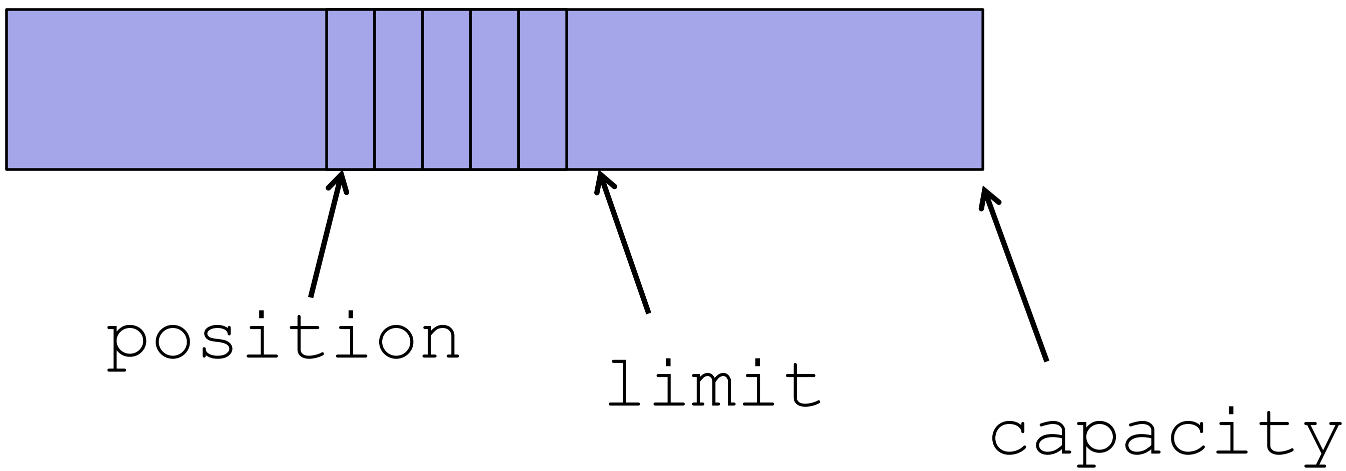
- Move data from Buffer to channel, starting at `position`, not to reach `limit`

Buffer.flip()



- Buffer.flip(): limit=position; position=0
- Why flip: used to switch from preparing data to output, e.g.,
  - buf.put(header); // add header data to buf
  - in.read(buf); // read in data and add to buf
  - buf.flip(); // prepare for write
  - out.write(buf);
- Typical pattern: read, flip, write

Buffer.compact()



- Move [position , limit) to 0
- Set position to limit-position, limit to capacity

```
// typical design pattern
buf.clear(); // Prepare buffer for use
for (;;) {
   if (in.read(buf) < 0 && !buf.hasRemaining())
      break; // No more bytes to transfer
   buf.flip();
   out.write(buf);
   buf.compact(); // In case of partial write
}
```