Network Applications: High-performance Server Design

Y. Richard Yang

http://zoo.cs.yale.edu/classes/cs433/

2/22/2016

Admin

- Assignment three will be posted tomorrow.
- Please start to think about projects
- Dates for exam 1?

Recap: Thread-Based Network Servers

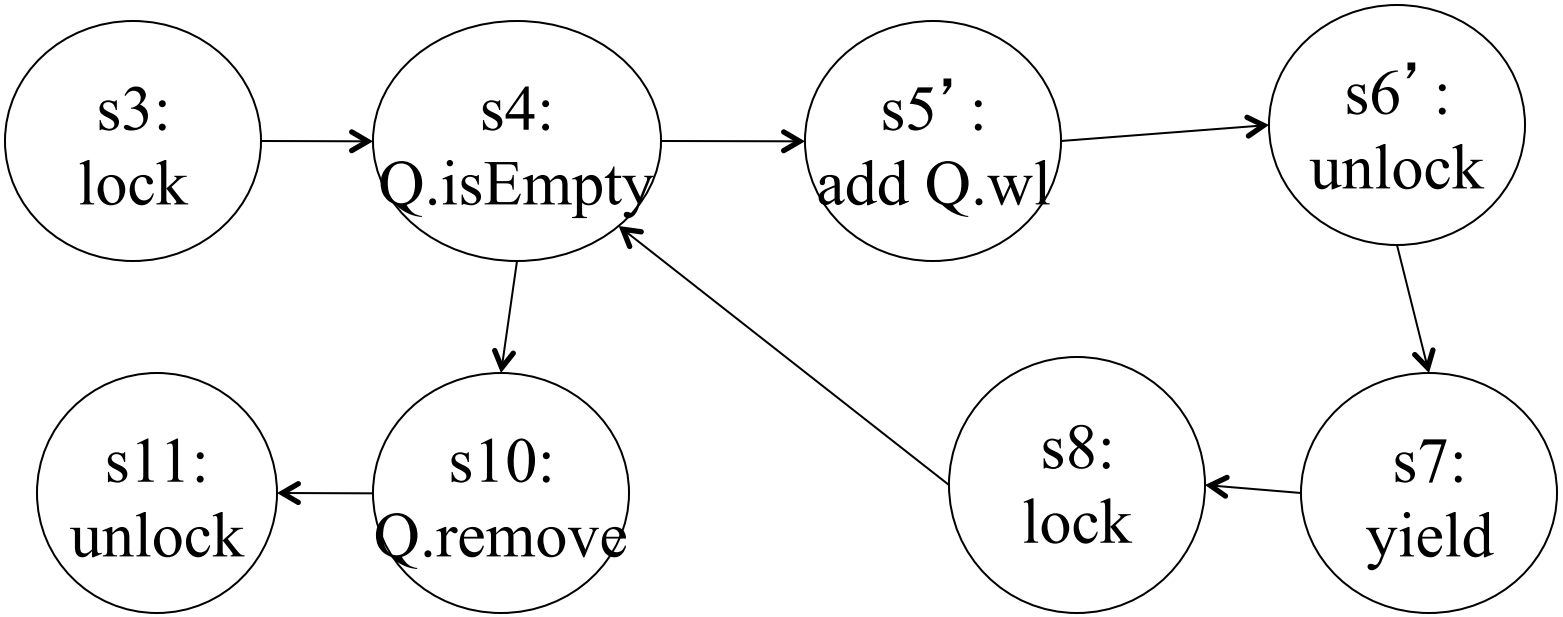
- Why: threads (execution sequences) so that only one thread is blocked
- How:
  - Per-request thread
    - problem: large # of threads and their creations/deletions may let overhead grow out of control
  - Thread pool
    - Design 1: Service threads compete on the welcome socket
    - Design 2: Service threads and the main thread coordinate on the shared queue
      - polling (busy wait)
      - suspension: wait/notify

Recap: 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)
  - main 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

Main thread can always add to Q

- Assume main 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



Each connection in Q is processed

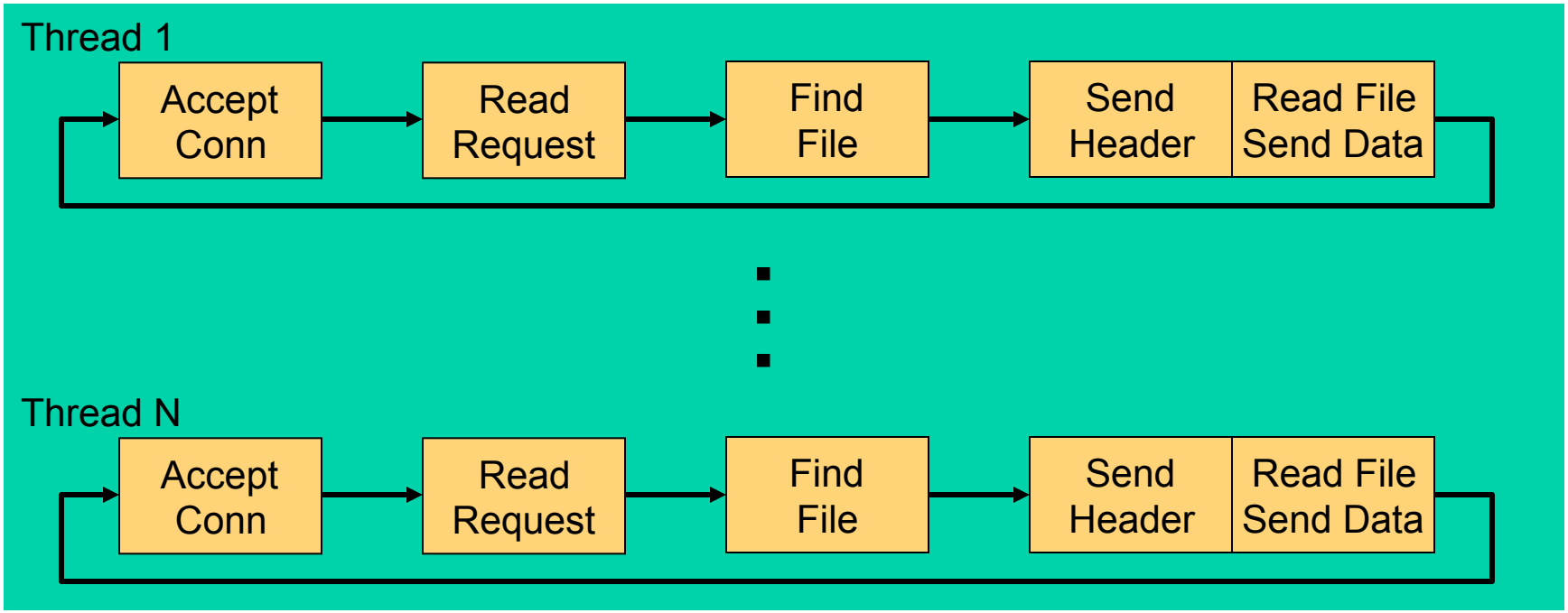
- Cannot be guaranteed unless
  - there is fairness in the thread scheduler, or
  - put a limit on Q size to block the main thread

Blocking Queues in Java

- Design Pattern for producer/consumer pattern with blocking
- 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

Summary: Using Threads

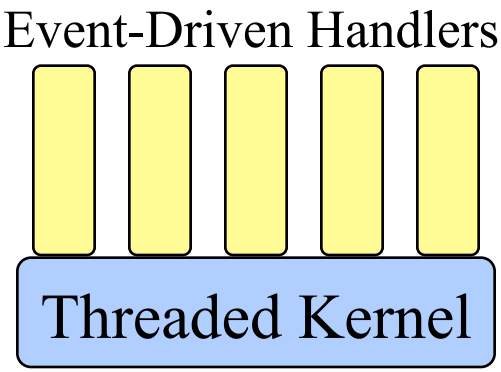


- Advantages
  - Intuitive (sequential) programming model
  - Shared address space simplifies optimizations
- Disadvantages
  - Overhead: thread stacks, synchronization
  - Thread pool parameter (how many threads) difficult to tune

Should You Use Threads?

- Typically avoid threads for io
  - Use event-driven, not threads, for GUIs, distributed systems, low-end servers.

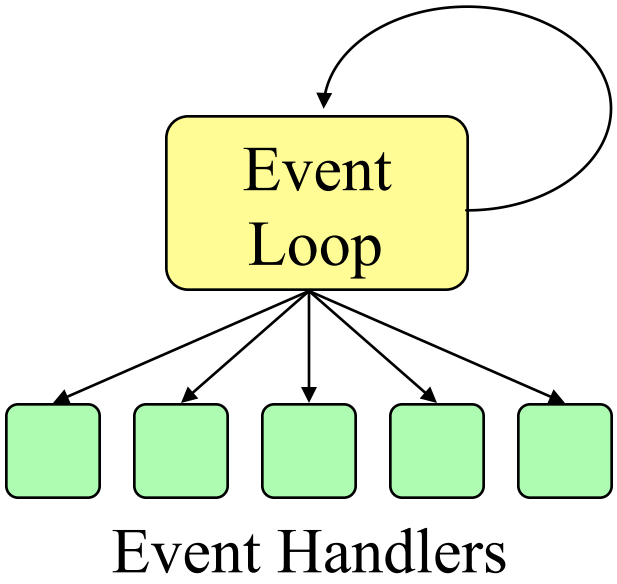
- Use threads where true CPU concurrency is needed.
  - Where threads needed, isolate usage in threaded application kernel: keep most of code single-threaded.



[Ousterhout 1995]

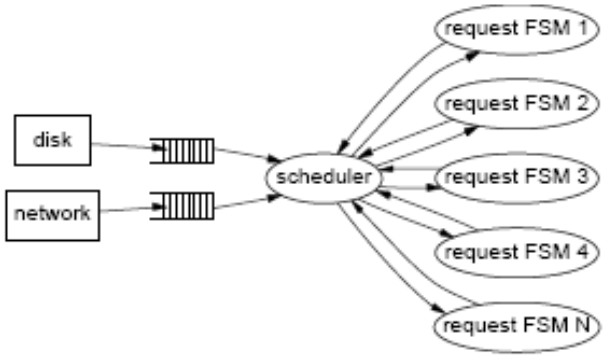
(Extreme) Event-Driven Programming

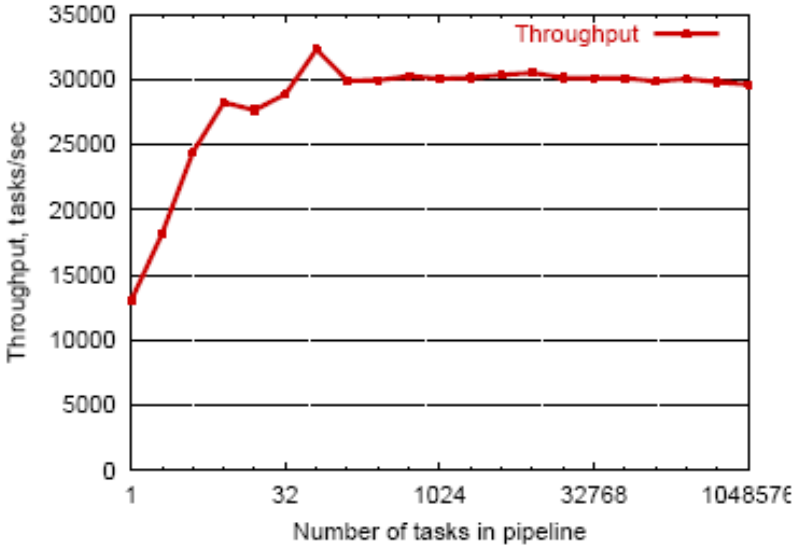
- One execution stream: no CPU concurrency
- A single-thread event loop issues commands, waits for events, invokes handlers (callbacks)
  - Handlers issue asynchronous (non-blocking) I/O
  - No preemption of event handlers (handlers generally short-lived)



[Ousterhout 1995]

Event-Driven Programming



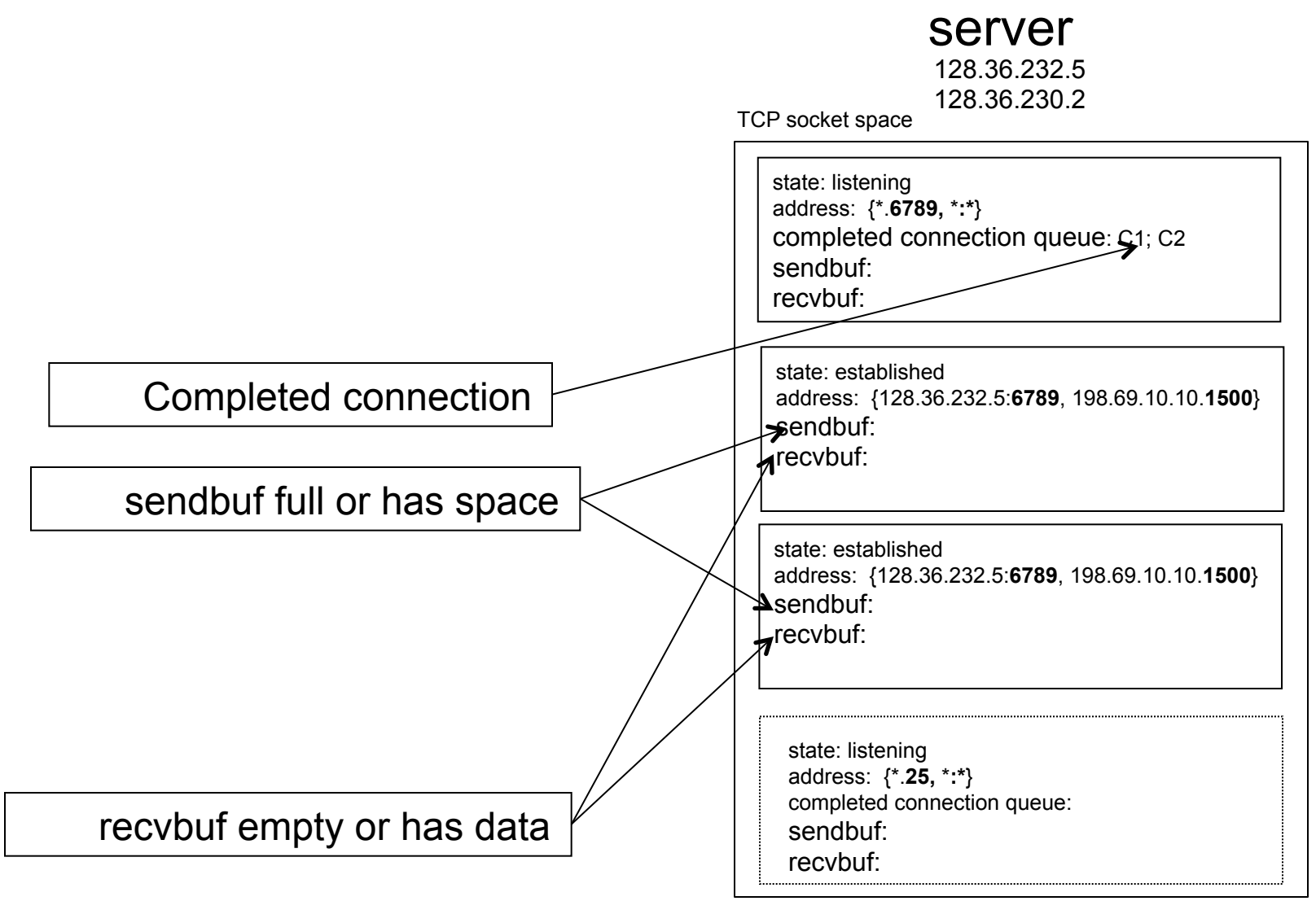


- Advantages
  - Single address space
  - No synchronization
- Disadvantages
  - Program complexity
  - In practice, disk reads/page fault still block
- Many examples: Click router, Flash web server, TP Monitors, NOX controller, Google Chrome (libevent), Dropbox (libevent),

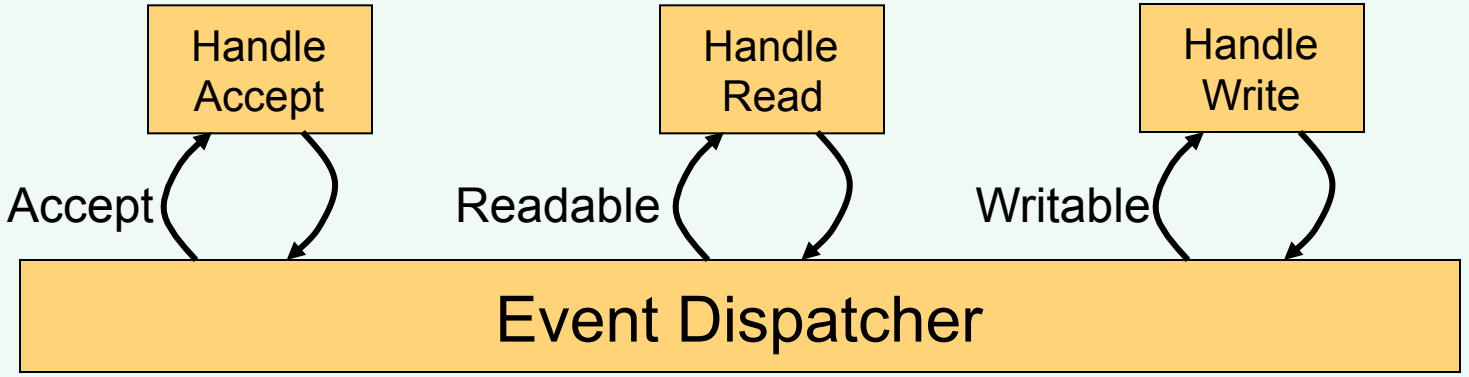
Async Server I/O Basis

- Modern operating systems, such as Windows, Mac and Linux, provide facilities for fast, scalable IO based on the use of asynchronous initiation (e.g., aio_read) and notifications of ready IO operations taking place in the operating system layers.
  - Windows: IO Completion Ports
  - Linux: select, epoll (2.6)
  - Mac/FreeBSD: kqueue
- An Async IO package (e.g., Java Nio, Boost ASOI, Netty) aims to make (some of) these facilities available to applications

Async I/O Example: Ready Operations



Async IO with OS Notification



- Software framework on top of OS notification
  - Register handlers with dispatcher on sources (e.g., which sockets) and events (e.g., acceptable, readable, writable) to monitor
  - Dispatcher asks OS to check if any ready source/event
  - Dispatcher calls the registered handler of each ready event/source

Dispatcher Structure

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

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

Outline

- Admin and recap
- High performance servers
  - Thread
    - Per-request thread
    - Thread pool
      - Busy wait
      - Wait/notify
  - Asynchronous servers
    - Overview
    - Java async io

Async I/O in Java

- Java AIO provides some platform-independent abstractions on top of OS notification mechanisms (e.g., select/epoll)
- A typical network server (or package) builds on top of AIO abstractions
  - Sources (channel)
  - Selector
  - Buffers

Async I/O in Java: Sources

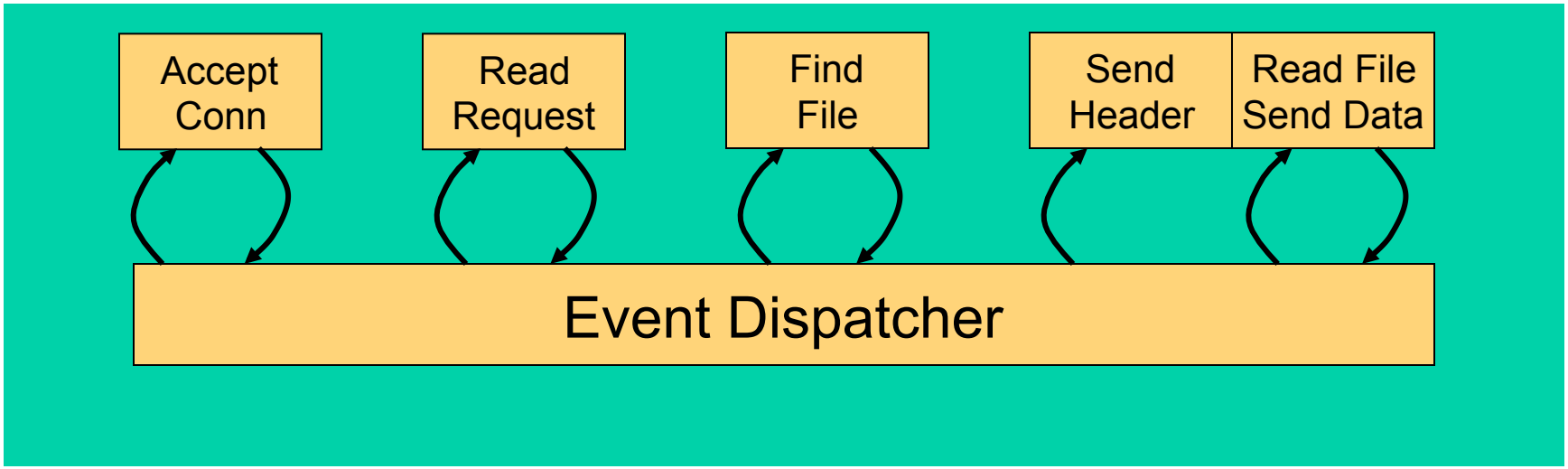
- A source that can generate events in Java is called a `SelectableChannel` object:
  - Example SelectableChannels: `DatagramChannel`, `ServerSocketChannel`, `SocketChannel`, `Pipe.SinkChannel`, `Pipe.SourceChannel`
  - use `configureBlocking(false)` to make a channel non-blocking

- Note: Java `SelectableChannel` does not include file I/O

https://docs.oracle.com/javase/8/docs/api/java/nio/channels/spi/AbstractSelectableChannel.html

Async I/O in Java: Selector

- An important class is the class `Selector`, which is a 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

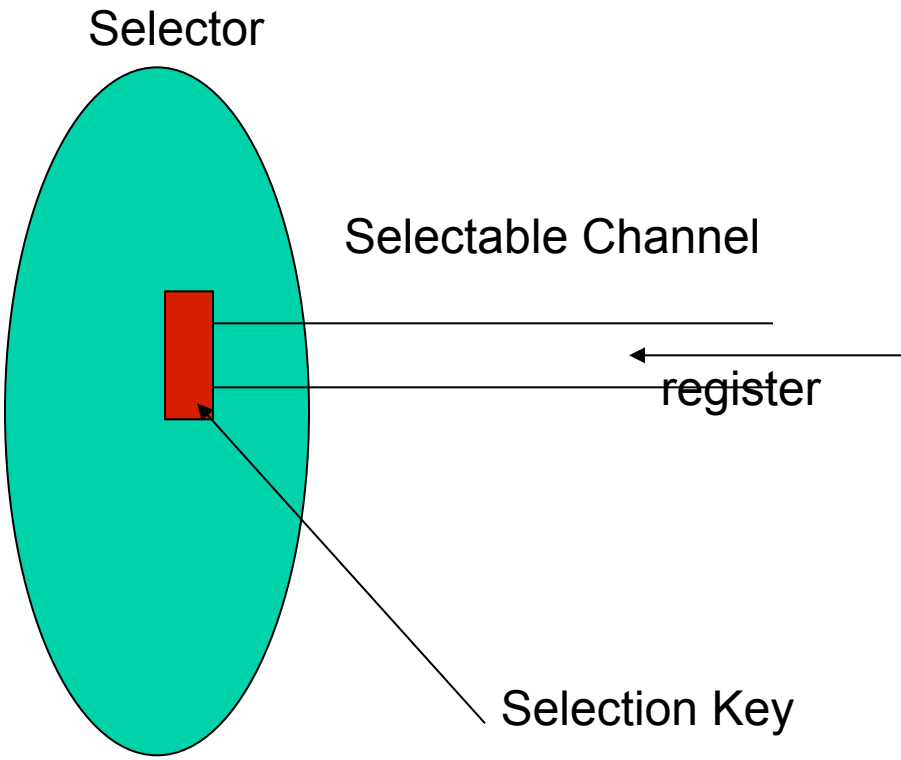
- The registration returns an object called a `SelectionKey`:

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

Java Async 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 Structure

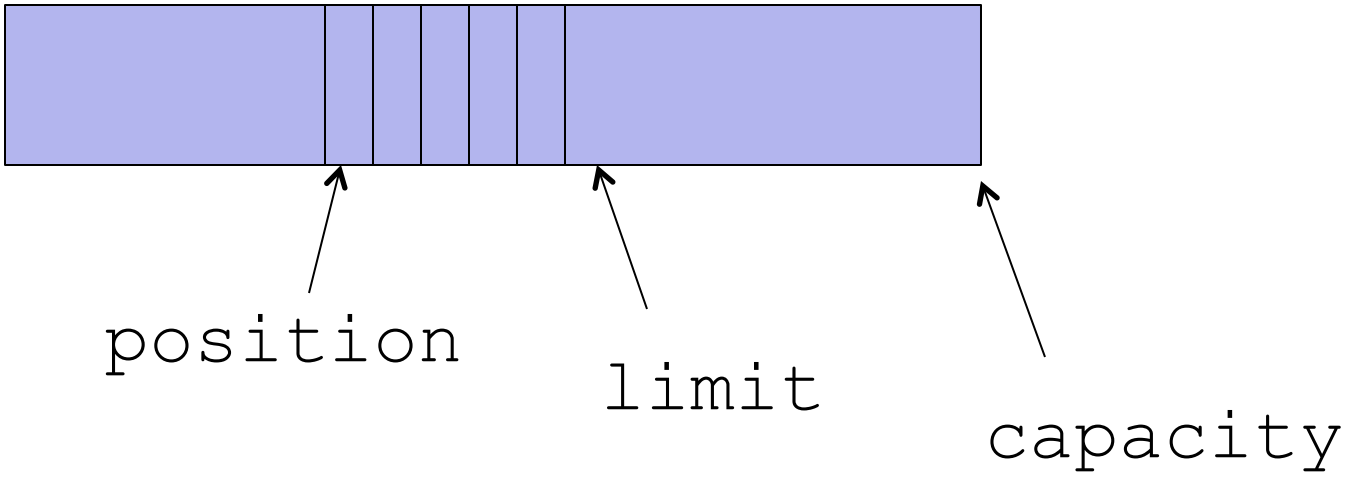
```
while (true)  {
  - selector.select()
  - Set readyKeys = selector.selectedKeys();

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

Async 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
- You can use it either as an absolute indexed or relatively indexed buffer

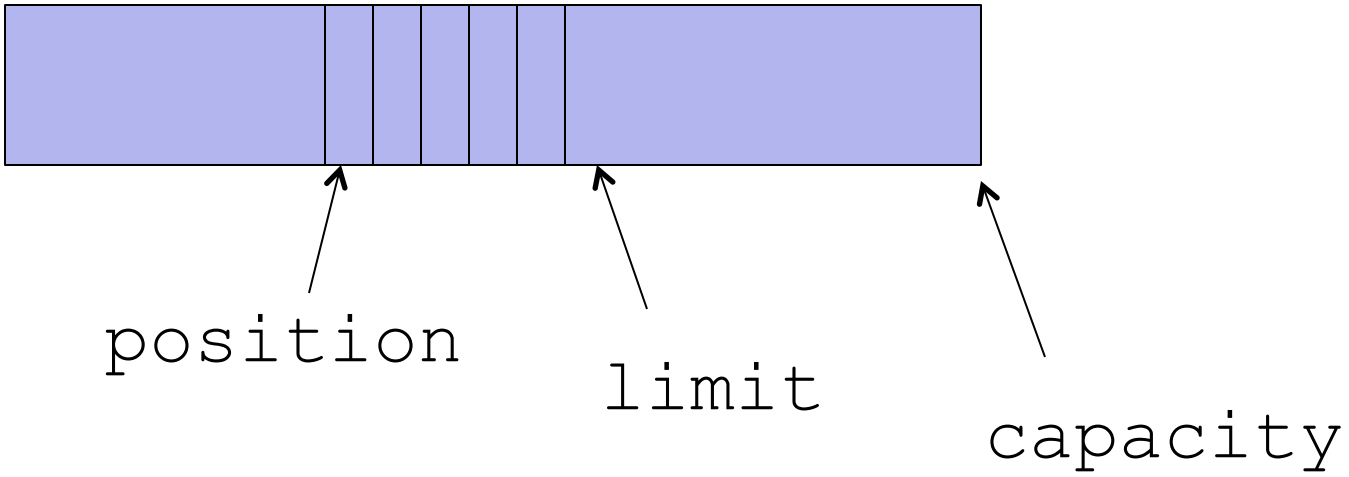
Buffer (relative index)



- Each Buffer has three numbers: position, limit, and capacity
  - Invariant: 0 <= *position* <= *limit* <= *capacity*

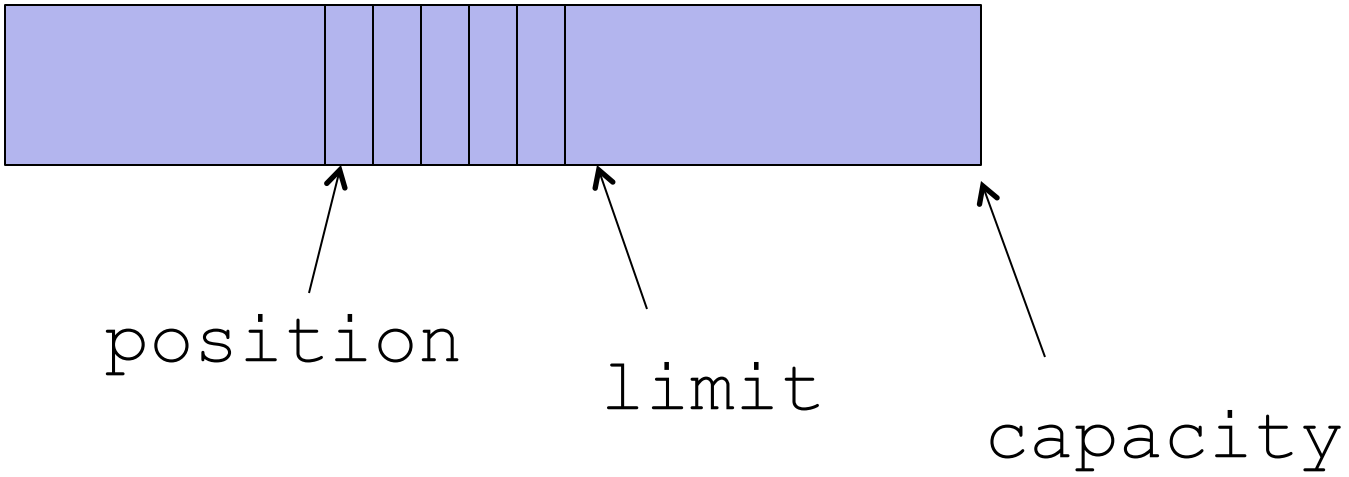
- `Buffer.clear(): position = 0; limit=capacity`

channel.read(Buffer)



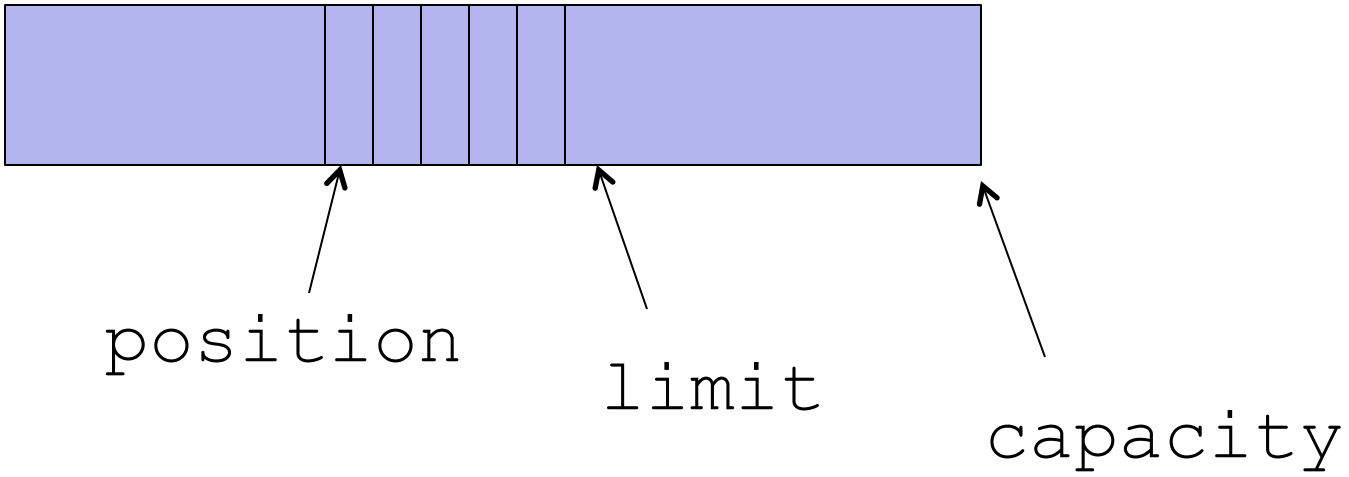
- Put data into Buffer, starting at `position`, not to reach `limit`

channel.write(Buffer)



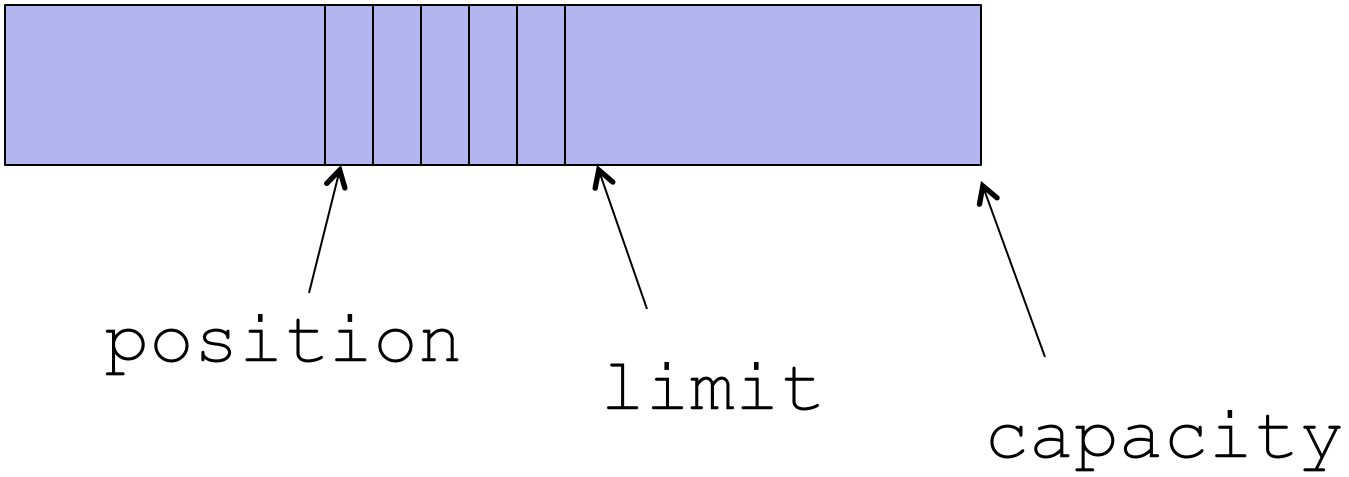
- Move data from Buffer to channel, starting at `position`, not to reach `limit`

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);

Buffer.compact()



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

```
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
}
```

Example

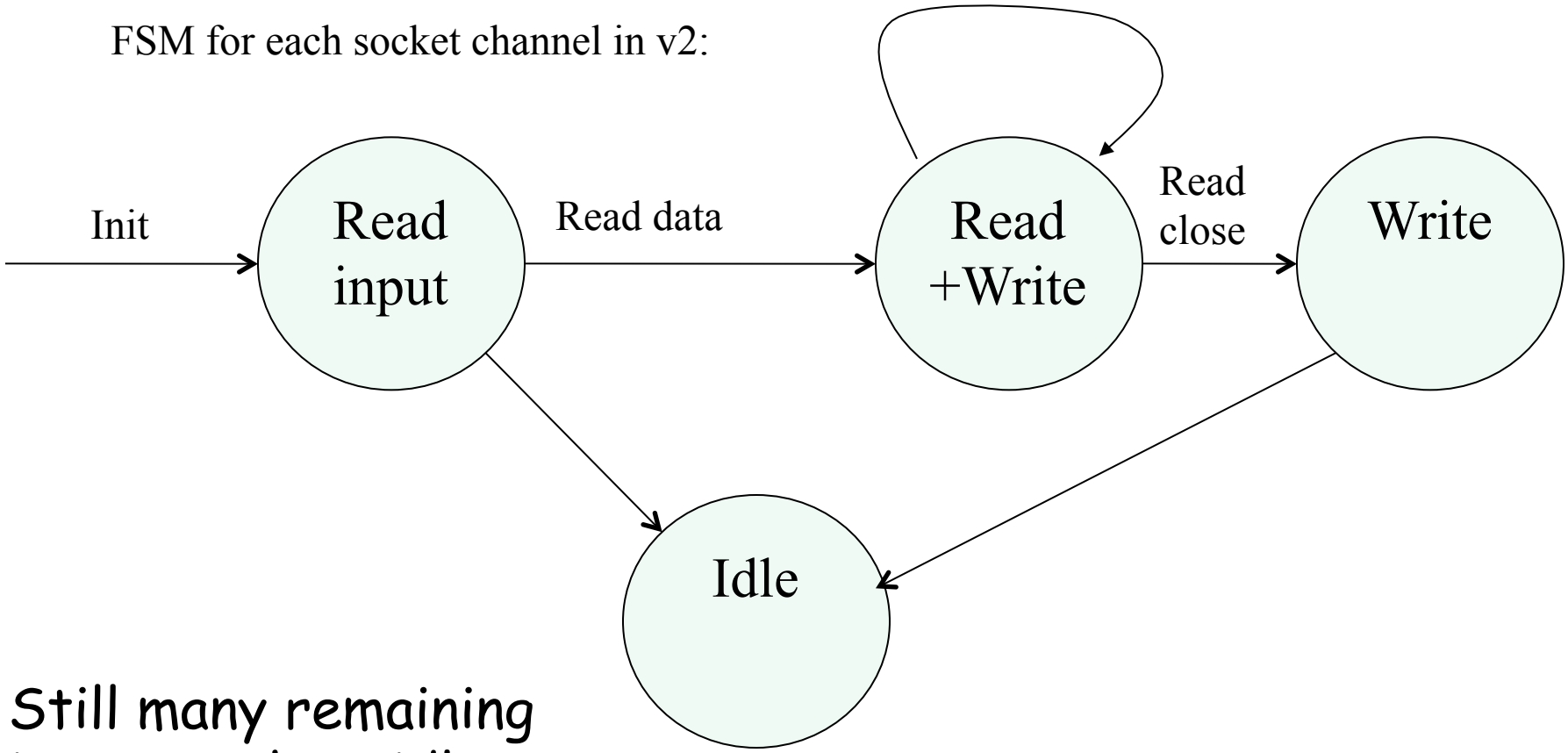
- See AsyncEchoServer/v1-2/EchoServer.java

Problems of Async Echo Server v1

- Empty write: Callback to `handleWrite()` is unnecessary when nothing to write
  - Imagine empty write with 10,000 sockets
  - Solution: initially read only, later allow write

- `handleRead()` still reads after the client closes
  - Solution: after reading end of stream (read returns -1), deregister read interest for the channel

(Partial) Finite State Machine (FSM)



Still many remaining issues such as idle instead of close

Finite-State Machine and Thread

- Why no need to introduce FSM for a thread version?