Network Applications: High-Performance Network Servers

Y. Richard Yang

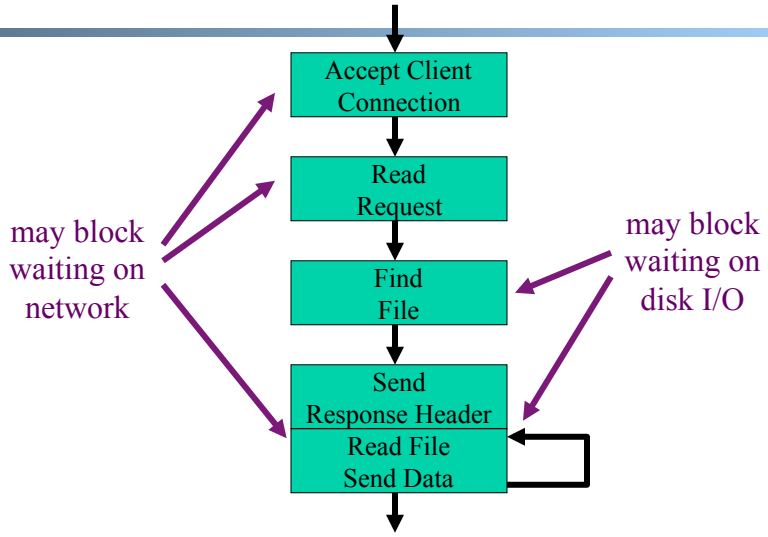
http://zoo.cs.yale.edu/classes/cs433/

10/01/2013

Outline

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

Recap: Server Processing Steps



Want to be able to process requests concurrently.

Recap: Progress So Far

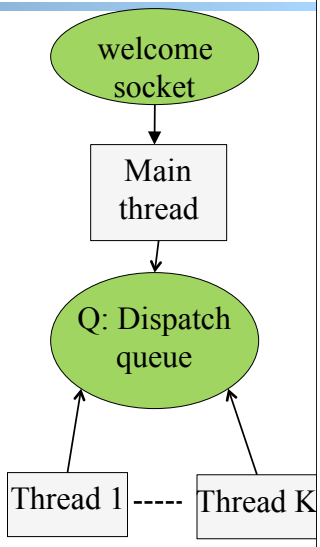
- Avoid blocking the whole program (so that we can reach bottleneck throughput)
  - Introduce threads
- Limit unlimited thread overhead
  - Thread pool (two designs)
- Coordinating data access
  - synchronization (lock, synchronized)
- Coordinating behavior: avoid busy-wait
  - Wait/notify

4

Recap: 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
}
```



5

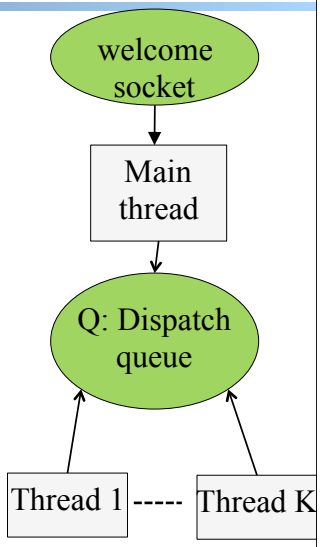
Worker

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
}
```



6

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

7

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

8

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

9

Example: Producer and Consumer using Suspension

```
public class ProducerConsumer {
   private boolean valueReady = false;
   private Object value;
   synchronized void produce(Object o) {
      while (valueReady) wait();
      value = o; valueReady = true;
      notifyAll();
   }
   synchronized Object consume() {
      while (!valueReady) wait();
      valueReady = false;
      Object o = value;
      notifyAll();
      return o;
   }
}
```

Q: which object's lock and waitset does this code use?

10

Note

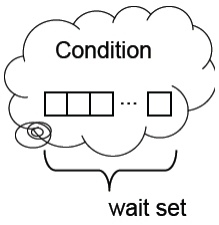
- Use of wait(), notifyAll() and notify() similar to
  - Condition queues of classic Monitors
  - Condition variables of POSIX PThreads API
  - In C# it is called Monitor (http://msdn.microsoft.com/en-us/library/ms173179.aspx)
- Python Thread model in its Standard Library is based on Java model
  - http://docs.python.org/3.3/library/concurrency.html

11

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)

12

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();
}
```

13

Blocking Queues in Java 1.5

- Design Pattern for producer/consumer pattern with blocking

```
interface Queue<E> extends Collection<E> {
   boolean offer(E x); /* produce */
   /* waits for queue to have capacity */

   E remove(); /* consume */
   /* waits for queue to become non-empty */
... }
```

- Two handy implementations
  - LinkedBlockingQueue (FIFO, may be bounded)
  - ArrayBlockingQueue (FIFO, bounded)
  - (plus a couple more)

14

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.

15

Correctness

- How do you analyze that the design is correct?

```
    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
```

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

16

Key Correctness Properties

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

17

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 the program and analyze the properties, but we will use basic analysis
  - This is enough in many cases

18

Make Program Explicit

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

```
1. main {
     void run {
2.     while (true) {
3.        Socket con = welcomeSocket.accept();
4.        lock(Q) {
5.        Q.add(con);
6.        notify Q.wait_list; // Q.notifyAll();
7.        unlock(Q);
8.     } // end of while
9.   }
```

19

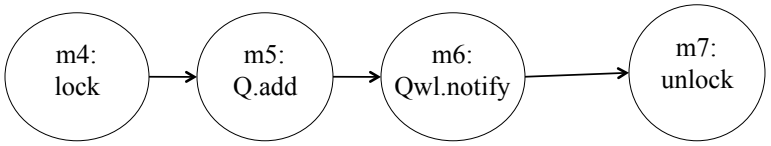
```
    while (true) {
       // get next request
       Socket myConn = null;
       synchronized(Q) {
           while (Q.isEmpty()) {
             Q.wait();
           }
           myConn = Q.remove();
        } // end of sync
        // process request in myConn
    } // end of while
```

```
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.      }
10.    myConn = Q.remove();
11.   unlock(Q);
   // process request in myConn
12.} // end of while
```

20

Statements to State

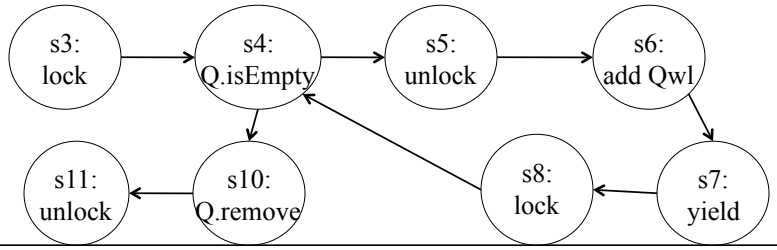
```
1. main {
     void run {
2.     while (true) {
3.        Socket con = welcomeSocket.accept();
4.        lock(Q) {
5.        Q.add(con);
6.        notify Q.wait_list; // Q.notifyAll();
7.        unlock(Q);
8.     } // end of while
9.   }
```



21

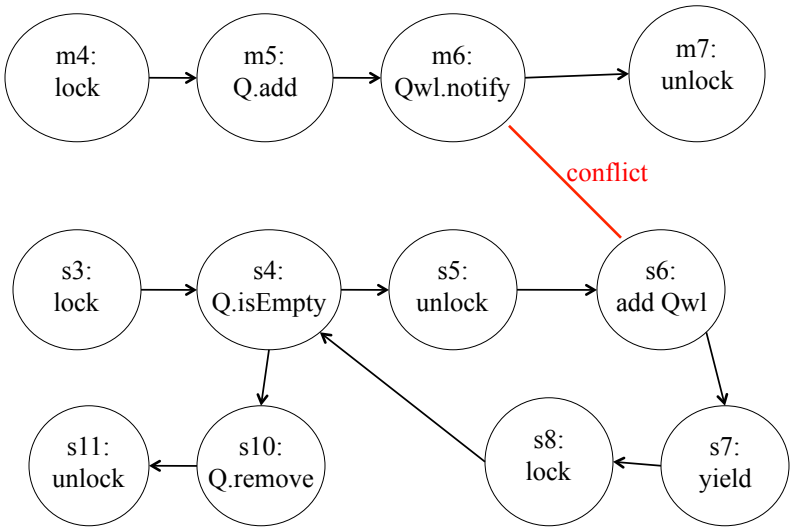
Statements

```
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; //wait
8.        lock(Q);
9.      }
10.    myConn = Q.remove();
11.   unlock(Q);
   // process request in myConn
12.} // end of while
```



22

Check Safety



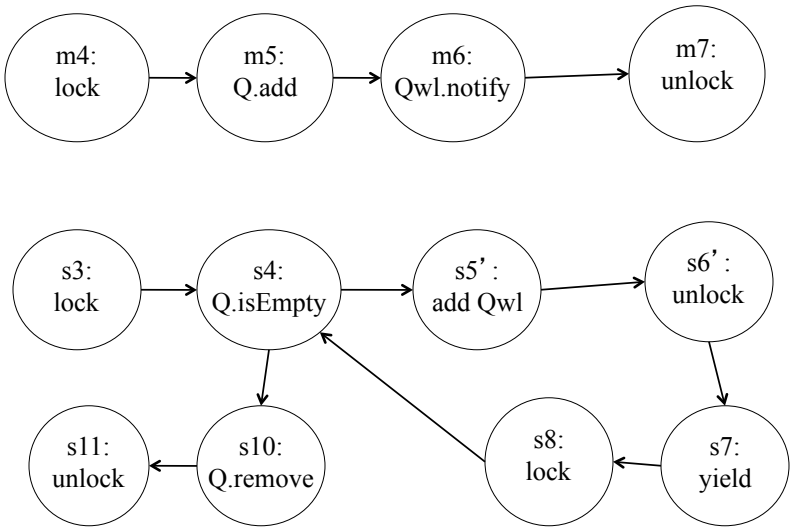
23

Real Implementation of `wait`

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

24

States



25

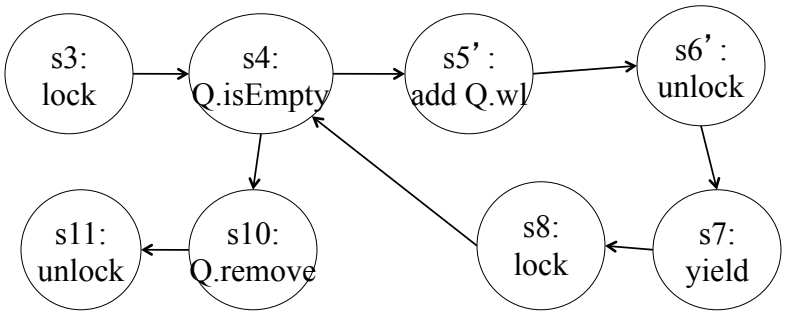
Liveness Properties

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

26

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



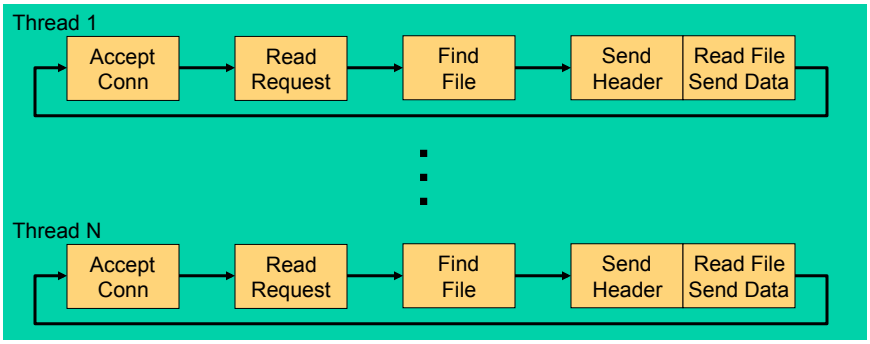
27

All elements in Q can be removed

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

28

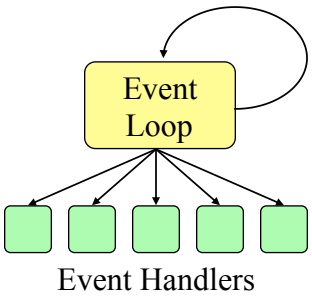
Summary: Using Threads



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

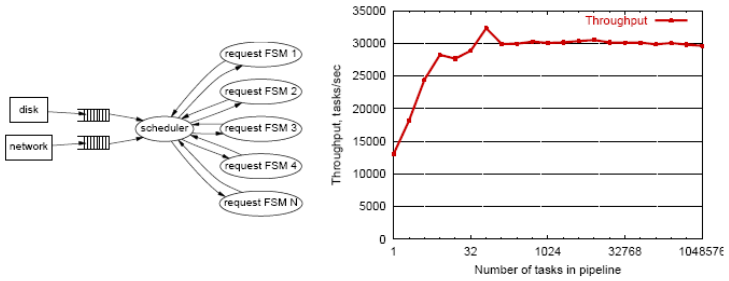
(Extreme) Event-Driven Programming

- 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]

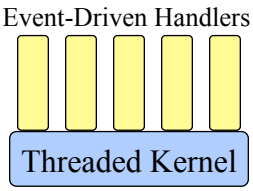
Event-Driven Programming



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

Should You Abandon Threads?

- No: important for high-end servers (e.g., databases).
- But, typically avoid threads
  - Use events, not threads, for GUIs, distributed systems, low-end servers.
  - Only 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]

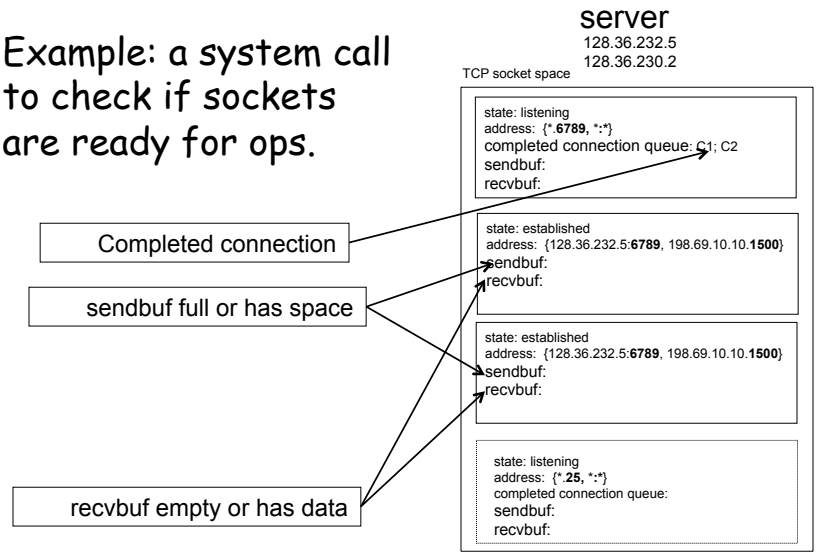
Async Server I/O Basis

- 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) aims to make (some of) these facilities available to applications

33

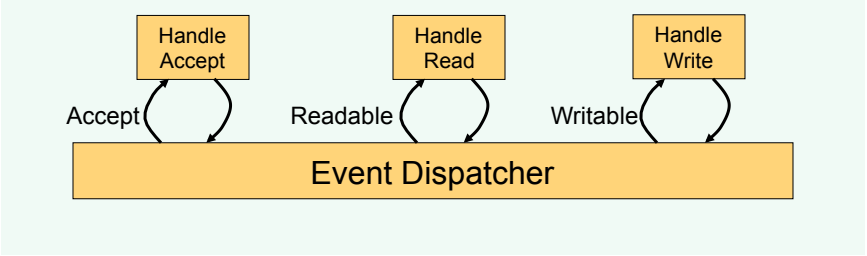
Async I/O Basis: Example

- Example: a system call to check if sockets are ready for ops.



34

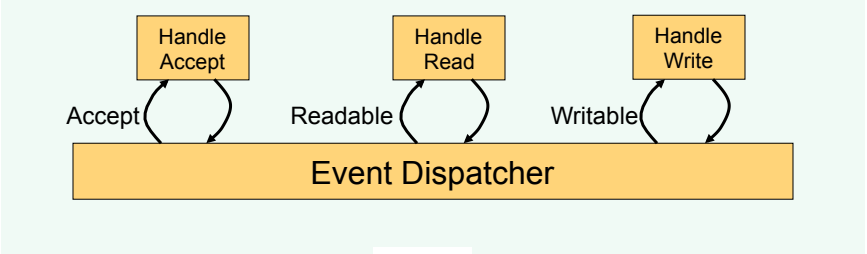
Async IO with OS Notification

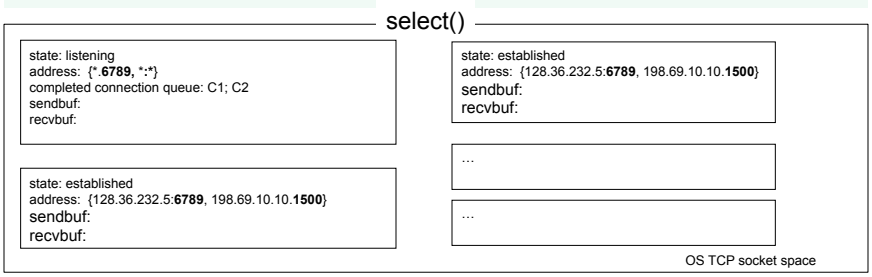


- Software framework on top of OS notification
  - Register handlers with dispatcher on sources (e.g., which sockets) and events (e.g., acceptable, readable, writable) to monitor
  - Dispatcher asks OS to check if any ready source/event
  - Dispatcher calls the registered handler of each ready event/source

35

Async IO Big Picture





36

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

38

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 a complete async io framework on top of the supporting mechanisms

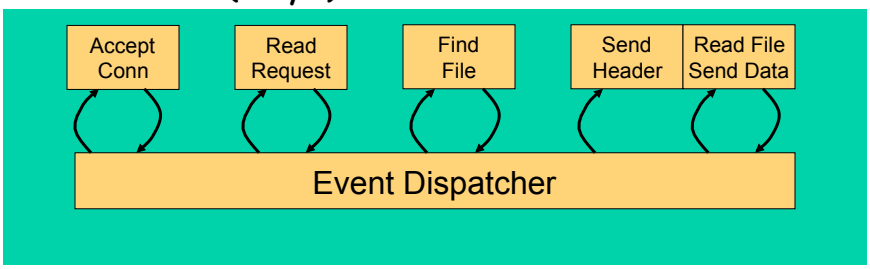
Async I/O in Java: Sources

- 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

http://java.sun.com/j2se/1.5.0/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

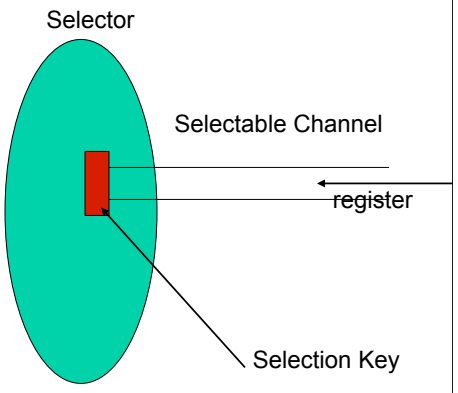
- A selectable channel registers events to be monitored with a `selector` with the `register` method

- 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
    }
}
```

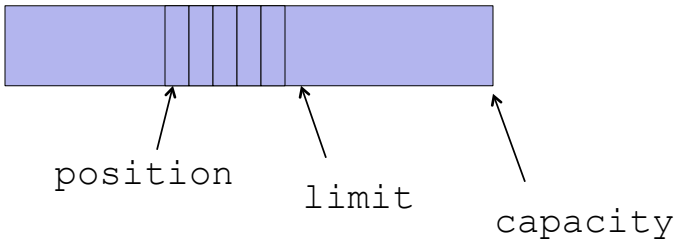
See AsyncEchoServer/v1/EchoServer.java

Short Intro. to 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

46

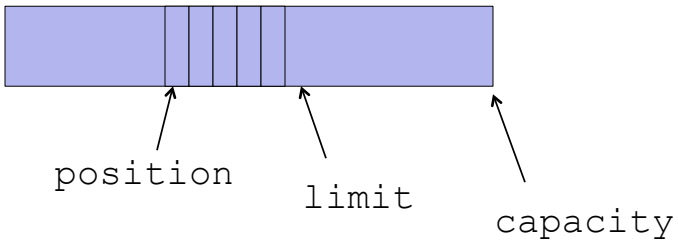
Buffer (relative index)



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

47

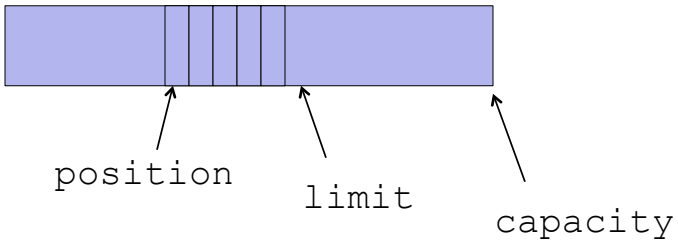
channel.read(Buffer)



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

48

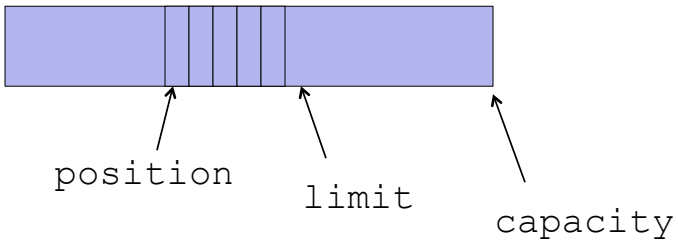
channel.write(Buffer)



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

49

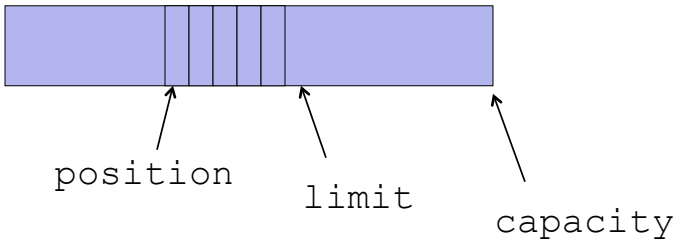
Buffer.flip()



- `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);`

50

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
}
```

51

Example

- See AsyncEchoServer/v1/EchoServer.java

52

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, deregister read interest for the channel

53