COMP202-08B
Computer Communications

Lecture 19

Java – Asynchronous I/O, Part One

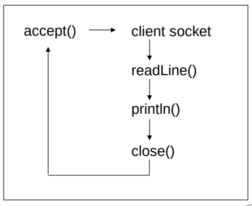
Remember HelloWorldServer?

Note: THIS CODE IS MISSING NECESSARY EXCEPTION HANDLING

```
ServerSocket ss = new ServerSocket(1234);
while(true) {
 Socket client = ss.accept();
 PrintWriter writer = new
    PrintWriter(client.getOutputStream(), true);
 BufferedReader reader = new BufferedReader(
    new InputStreamReader(client.getInputStream()));
 String line = reader.readLine();
 writer.println(“You said: “ + line);
 client.close();
}
```

Remember HelloWorldServer?

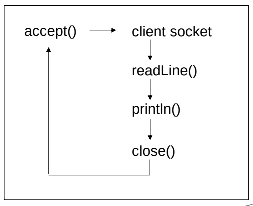
- Main problem is the blocking nature of the readLine():
  - readLine does not return until it has a complete line
  - any new connection to HelloWorldServer will not be serviced until a line has been read



Remember HelloWorldServer?

- Secondary problem is the blocking nature of the println():
  - println does not return until operating system can buffer line internally for transmission
  - any new connection to HelloWorldServer will not be serviced until buffer space has been freed, if necessary

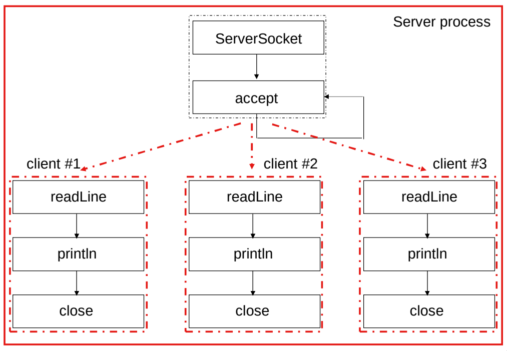
Note: this problem is unlikely to occur with HelloWorldServer, but is more likely to occur with a real-life application that sends lots of data over the network.



Remember HelloWorldServer?

- How did we get around these problems in HelloWorldServer2?

HelloWorldServer2



Threads

- Just as our computer can run multiple programs at the same time, each of which share the processor, we can have two strands of our program run at the same time
- Each strand is a Thread
- Jargon
  - Operating system runs processes
  - Programs have threads
  - Threads can be thought of as light-weight processes
  - Threads can share data structures amongst themselves

HelloWorldServer2.java

```
class HelloWorldServer2 {
 public static void main(String args[])
 {
   try {
    ServerSocket ss = new ServerSocket(1234);
    while(true) {
      Socket client = ss.accept();
      HelloWorldServerThread thread =
        new HelloWorldServerThread(client);
      thread.start();
    }
  }
  catch(Exception e) {
    System.err.println("Exception: " + e);
  }
 }
}
```

HelloWorldServer2.java

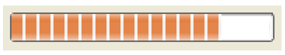
```
class HelloWorldServerThread extends Thread {
  public void run() {
    try {
       PrintWriter writer = new
         PrintWriter(client.getOutputStream(), true);
       BufferedReader reader =
         new BufferedReader(
         new InputStreamReader(client.getInputStream()));
       String line = reader.readLine();
       writer.println("You said: " + line);
       client.close();
    } catch(Exception e) {
       System.err.println("Exception: " + e);
    }
  }
}
```

Threads

- Multiple points of execution
- Shared program variables amongst threads
  - ChatServer.java: list of connected clients and their sockets
- Need for synchronisation
  - ensure only one thread is able to modify data at a time
  - with `synchronize`
- Can be problematic:
  - Need to ensure shared data is correctly locked

What are threads good for?

- Taking advantage of multiple CPUs now present in nearly all computer systems
  - Core2duo 2 CPUs per processor
  - Core2quad 4 CPUs per processor
  - requires programmer ability to break up workload to efficiently make use of available capability
- GUI applications: processing data while keeping user up to date with progress



- … other things too

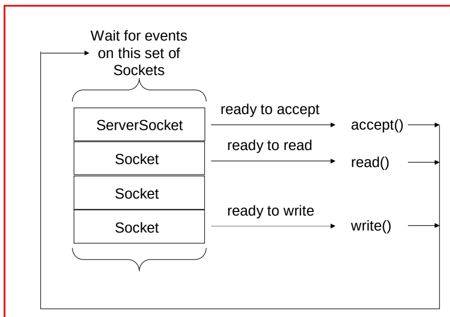
Threads and Sockets

- HelloWorldServer2 (and ChatServer.java) are really good examples of crappy applications that use threads
  - They are not CPU bound!
  - Threads used as a convenient way to get around blocking
- Each time a new connection is established, operating system has to spawn a new thread
  - only scales so far
  - requires operating system to schedule threads to execute
    - overhead
  - threads have their own stack, copies of CPU registers

Alternative: Event-based

- Similar to programming model used when programming GUI applications
  - wait for user to click a button, then do something in response
- Network applications:
  - wait for data to arrive, then do something
  - wait for a new connection to be established, then do something
  - wait for buffer space to come available, then write data
- Faster than spawning threads for each socket:
  - no process scheduling, synchronisation overheads

Event-based network applications



Event based applications in Java

- Until Java 1.4, Sockets + threads was only way to get around blocking
  - 2002
  - though blocking problem has been solved for at least 25 years.
- Java introduced “New I/O”
  - import java.nio.*
  - import java.nio.channels.*
- Selector
- Channels
  - ServerSocketChannel
  - SocketChannel

Selector + Channels : ServerSocket

- Monitor multiple event channels for information simultaneously, using a selector

```
/* create a selector; used to group Channels */
Selector selector = Selector.open();

/* create non-blocking server socket, port 1234 */
ServerSocketChannel ssc;
ssc.socket().bind(new InetSocketAddress(1234));
ssc.configureBlocking(false);

/* ask selector to watch for new connections */
ssc.register(selector, SelectionKey.OP_ACCEPT);

/* block until an event occurs */
selector.select();
```

Selector + Channels : Socket

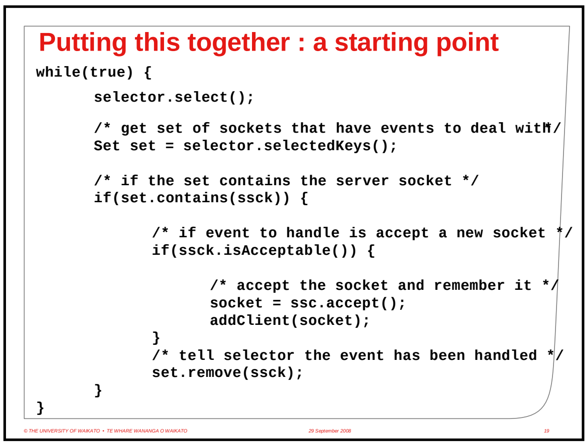
- Code for regular sockets similar:

```
SocketChannel socket;

/*
 * MISSING: accept connection to socket, or
 * connect using socket
 */

/* configure the socket to be non-blocking */
socket.configureBlocking(false);
socket.register(selector, SelectionKey.OP_READ);
```

- Can register socket for OP_WRITE as well.

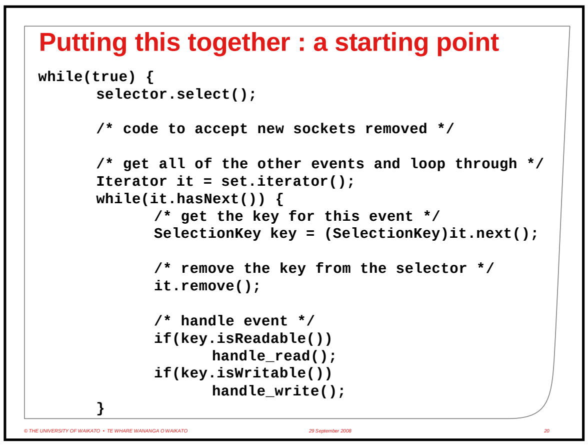
Putting this together : a starting point

```
Selector selector = Selector.open();
ServerSocketChannel ssc;
SocketChannel socket;
SelectionKey ssck;

ssc.socket().bind(new InetSocketAddress(1234));
ssc.configureBlocking(false);
ssck = ssc.register(selector, SelectionKey.OP_ACCEPT

while(true) {
        selector.select();
        Set set = selector.selectedKeys();

        if(set.contains(ssck)) {
                if(ssck.isAcceptable()) {
                        socket = ssc.accept();
                        addClient(socket);
                }
                set.remove(ssck);
        }
}
```



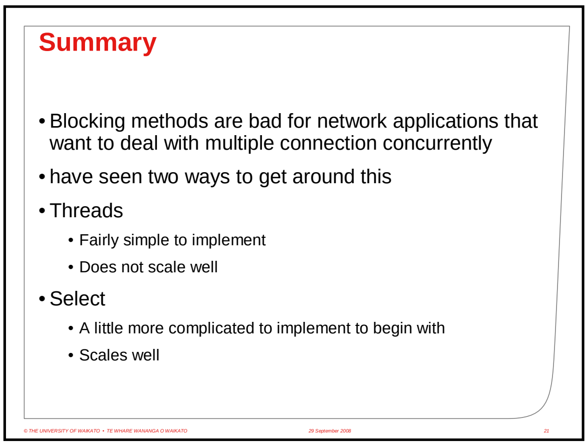
Putting this together : a starting point

```
while(true) {
        selector.select();

        /* get set of sockets that have events to deal with*/
        Set set = selector.selectedKeys();

        /* if the set contains the server socket */
        if(set.contains(ssck)) {

                /* if event to handle is accept a new socket */
                if(ssck.isAcceptable()) {

                        /* accept the socket and remember it */
                        socket = ssc.accept();
                        addClient(socket);
                }
                /* tell selector the event has been handled */
                set.remove(ssck);
        }
}
```



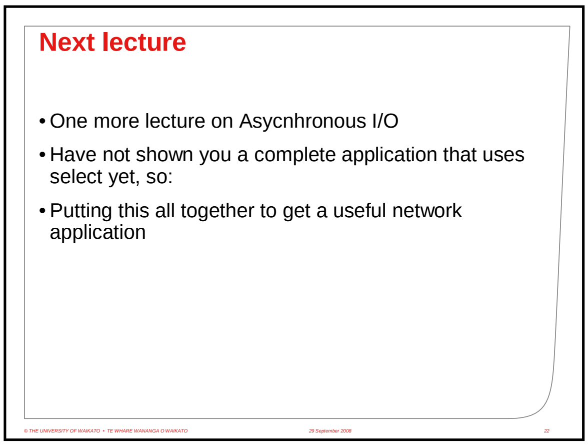
Putting this together : a starting point

```
while(true) {
        selector.select();

        /* code to accept new sockets removed */

        /* get all of the other events and loop through */
        Iterator it = set.iterator();
        while(it.hasNext()) {
                /* get the key for this event */
                SelectionKey key = (SelectionKey)it.next();

                /* remove the key from the selector */
                it.remove();

                /* handle event */
                if(key.isReadable())
                        handle_read();
                if(key.isWritable())
                        handle_write();
        }
```



Summary

- Blocking methods are bad for network applications that want to deal with multiple connection concurrently
- have seen two ways to get around this
- Threads
  - Fairly simple to implement
  - Does not scale well
- Select
  - A little more complicated to implement to begin with
  - Scales well



Next lecture

- One more lecture on Asycnhronous I/O
- Have not shown you a complete application that uses select yet, so:
- Putting this all together to get a useful network application