Java

JavaOne℠

Asynchronous I/O Tricks and Tips

Alan Bateman
Sun Microsystems Inc.

Jeanfrancois Arcand
Sun Microsystems Inc.

Agenda

- Part 1
  - Overview of Asynchronous I/O API
  - Demultiplexing I/O events and thread pools
  - Usage notes and other topics
- Part 2
  - Grizzly Architecture
  - Thread Pool Strategies
  - Tricks
- Conclusion

Concept

- Initiate non-blocking I/O operation
- Notification when I/O completes

Concept

- Initiate non-blocking I/O operation
- Notification when I/O completes
- Compare with non-blocking synchronous I/O
  - notification when channel ready for I/O (Selector)
  - perform non-blocking I/O operation
  - Reactor vs. Proactor pattern

Two forms

- Initiate non-blocking I/O operation
  - Return j.u.c.Future representing pending result

Two forms

- Initiate non-blocking I/O operation
  - Return j.u.c.Future representing pending result
- Initiate non-blocking I/O operation specifying CompletionHandler
  - CompletionHandler invoked when I/O completes

Using Future

```
AsynchronousSocketChannel ch = ...
ByteBuffer buf = ...

Future<Integer> result = ch.read(buf);
```

Using Future

```
AsynchronousSocketChannel ch = ...
ByteBuffer buf = ...

Future<Integer> result = ch.read(buf);

// check if I/O operation has completed
boolean isDone = result.isDone();
```

Using Future

```
AsynchronousSocketChannel ch = ...
ByteBuffer buf = ...

Future<Integer> result = ch.read(buf);

// wait for I/O operation to complete
int nread = result.get();
```

Using Future

```
AsynchronousSocketChannel ch = ...
ByteBuffer buf = ...

Future<Integer> result = ch.read(buf);

// wait for I/O operation to complete with timeout
int nread = result.get(5, TimeUnit.SECONDS);
```

CompletionHandler

```
interface CompletionHandler<V,A> {
    void completed(V result, A attachment);
    void failed(Throwable exc, A attachment);
}
```

- V = type of result value
- A = type of object attached to I/O operation
  - Used to pass context
  - Typically encapsulates connection context
- completed method invoked if success
- failed method invoked if I/O operations fails

Using CompletionHandler

```
class Connection { … }

class Handler implements CompletionHandler<Integer,Connection> {
    public void completed(Integer result, Connection conn) {
        int nread = result;
        // handle result
    }
    public void failed(Throwable exc, Connection conn) {
        // error handling
    }
}
```

Using CompletionHandler

```
class Connection { … }

class Handler implements CompletionHandler<Integer,Connection> {
    public void completed(Integer result, Connection conn) {
        // handle result
    }
    public void failed(Throwable exc, Connection conn) {
        // error handling
    }
}

AsynchronousSocketChannel ch = ...
ByteBuffer buf = ...
Connection conn = ...
Handler handler = ...

ch.read(buf, conn, handler);
```

AsynchronousSocketChannel

- Asynchronous connect
- Asynchronous read/write
- Asynchronous scatter/gather (multiple buffers)
- Read/write operations support timeout
  - failed method invoked with timeout exception
- Implements NetworkChannel
  - for binding, setting socket options, etc.

AsynchronousServerSocketChannel

- Asynchronous accept
  - handler invoked when connection accepted
  - Result is AsynchronousSocketConnection
- Implements NetworkChannel
  - for binding, setting socket options, etc.

AsynchronousDatagramChannel

- Asynchronous read/write (connected)
- Asynchronous receive/send (unconnected)
  - Result of receive is sender address
- Implements NetworkChannel
  - for binding, setting socket options, etc.
- Implements MulticastChannel
  - to join multicast groups

AsynchronousFileChannel

- Asynchronous read/write
- No global file position/offset
  - Each read/write specifies position in file
  - Access different parts of file concurrently

AsynchronousFileChannel

- Asynchronous read/write
- No global file position/offset
  - Each read/write specifies position in file
  - Access different parts of file concurrently

```
Future<Integer> result = channel.write(buf, position);
doSomethingElse();
int nwrote = result.get();
```

AsynchronousFileChannel

- Open method specifies options
  - READ, WRITE, TRUNCATE_EXISTING, …
  - No APPEND
  - Can specify initial attributes when creating file
- Also supports file locking, size, truncate, ...

Groups

- What threads invoke the completion handlers?
- Network oriented channels bound to a group
  - AsynchronousChannelGroup
- Group encapsulates thread pool and other shared resources
- Create group with thread pool
- Default group for simpler applications
- Completion handlers invoked by pooled threads
- AsynchronousFileChannel can be created with its own thread pool (group of one)

Creating a group

```
// fixed thread pool
ThreadFactory myThreadFactory = …
int nthreads = …

AsynchronousChannelGroup group = AsynchronousChannelGroup
    .withFixedThreadPool(nThreads, threadFactory);
```

Creating a group

```
// custom thread pool
ExecutorService pool = ...

AsynchronousChannelGroup group = AsynchronousChannelGroup
    .withThreadPool(pool);
```

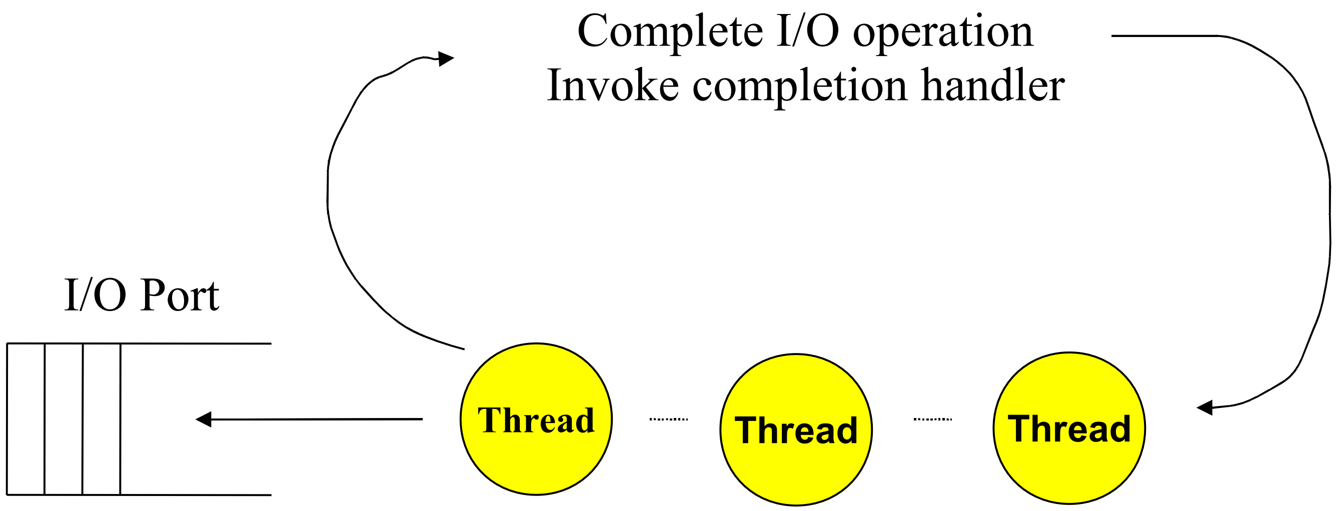
Creating a group

```
// custom thread pool
ExecutorService pool = ...

AsynchronousChannelGroup group = AsynchronousChannelGroup
    .withThreadPool(pool);

AsynchronousSocketChannel channel =
    AsynchronousSocketChannel.open(group);
```

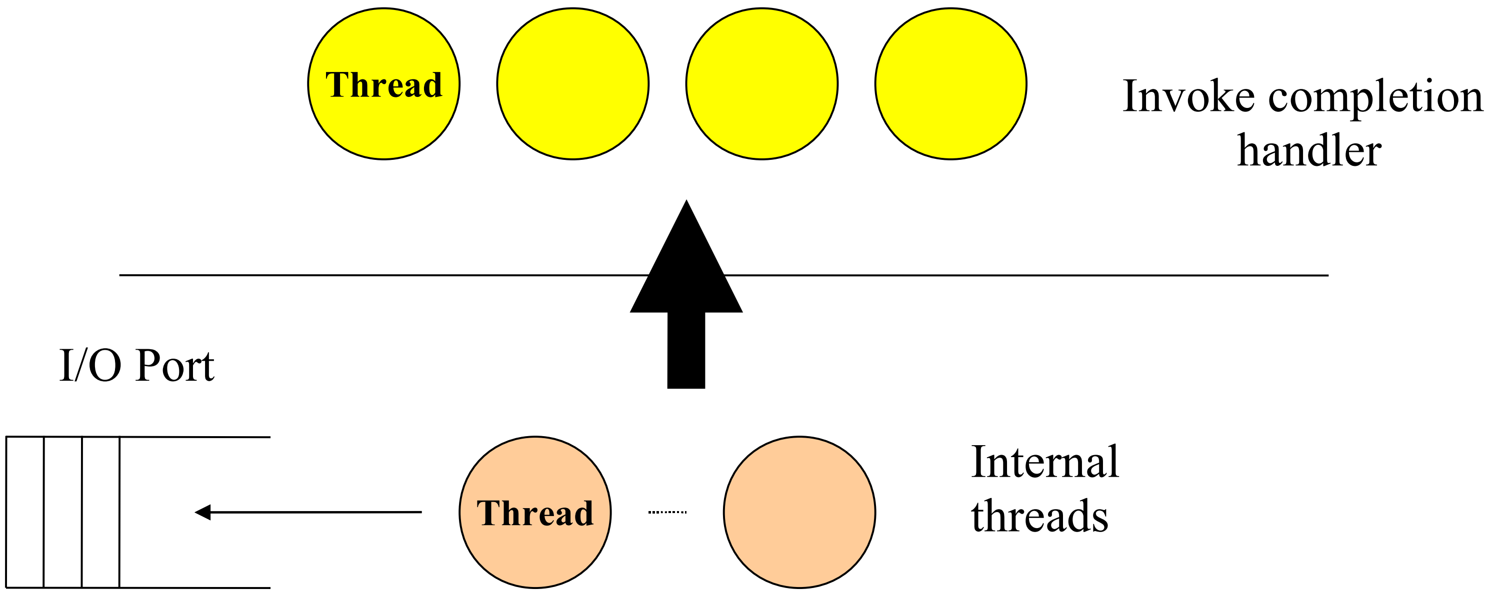
Thread pools

- Fixed thread pool
  - Each thread waits on I/O event
  - do I/O completion
  - invoke completion handler
  - go back to waiting for I/O events
- Cached or custom thread pool
  - Internal threads wait on I/O events
  - Submit tasks to thread pool to dispatch to completion handler

Fixed thread pool



Cached and custom thread pools



More on CompletionHandlers

- Should complete in a timely manner
  - Avoid blocking indefinitely
  - Important for fixed thread pools
- May be invoked directly by initiating thread
  - when I/O operation completes immediately, and
  - initiating thread is pooled thread
  - may have several handler frames on thread stack
  - implementation limit to avoid stack overflow
- Termination due to uncaught error or runtime exception causes pooled thread to exit

ByteBuffers

- Not safe for use by multiple concurrent threads
- When I/O operation is initiated then must take great care not to access buffer until I/O operation completes
- Memory requirements for buffers depends on number of outstanding I/O operations
- Heap buffers incur additional copy per I/O
  - As per SocketChannel API, compare performance
  - Copy performance and temporary direct buffer usage improved

Other topics

- Queuing not supported on stream connections
  - A short-write would corrupt the stream
  - Handlers not guaranteed to be invoked in order
  - Read/WritePendingException to catch bugs
- Asynchronous close
  - Causes all outstanding I/O operations to fail
- Cancellation
  - Future interface defines cancel method
  - Forceful cancel allows to close channel

Grizzly Architecture

Grizzly Comet Framework

HTTP (Sync/Async) | HTTP (Sync/Async)

Grizzly NIO Framework

JDK Kernel NIO | JDK Kernel AIO

Which Thread Pool strategy?

> With AIO, you can configure the thread pool (ExecutorService) used by both the AIO kernel and your application

AsynchronousChannelGroup.withCachedThreadPool
(ExecutorService, initialSize)
AsynchronousChannelGroup.withThreadPool
(ExecutorService)
AsynchronousChannelGroup.withFixedThreadPool
(nThread, ThreadFactory)

…or use the preconfigured/built in Thread Pool that comes by default…

FixedThreadPool

- An asynchronous channel group associated with a fixed thread pool of size N creates N threads that are waiting for already processed I/O events.
- The kernel dispatches events directly to those threads:
    - Thread first completes the I/O operation (like filling a ByteBuffer during a read operation).
    - Next invoke the CompletionHandler.completed() that consumes the result.
    - When the CompletionHandler terminates normally then the thread returns to the thread pool and wait on a next event.

Brrr…It's freezing here!

- What about if all threads "dead lock" inside a CompletionHandler?
  - Bang! your entire application can hang until one thread becomes free to execute again.
  - The kernel is no longer able to EXECUTE anything!
- Hence this is critically important CompletionHandler complete in a timely manner and avoid blocking.
- If all completion handlers are blocked, any new event will be queued until one thread is 'delivered' from the lock.
- Avoid blocking operations inside a completion handler.

Tip # 1 - FixedThreadPool!

Avoid blocking/long lived operations inside a completion handler.

If not possible, either use a CachedThreadPool or another ExecutorService that can be used from a completion handler

CachedThreadPool

> An asynchronous channel group associated with a cached thread pool submits events to the thread pool that simply invoke the user's completion handler.

> Internal kernel's I/O operations are handled by one or more internal threads that are not visible to the user application.

> That means you have one hidden thread pool that dispatches events to a cached thread pool, which in turn invokes completion handler

> Wait! you just win a prize: a thread's context switch for free!!

OOM, here we come!

- Probability of suffering the hang problem compared with the FixedThreadPool is lower.
- Still might grow infinitely…
- At least you guarantee that the kernel will be able to complete its I/O operations (like reading bytes).
- Oops…CachedThreadPool must support unbounded queuing to works properly.
- So you can possibly lock all the threads and feed the queue forever until OOM happens.

Tip # 2 - CachedThreadPool!

Avoid blocking/long lived operations inside a completion handler.

Possibility of OOM if the queue grow indefinitively => monitor the queue

Kernel/default thread pool.

- Hybrid of the above configurations:
  - Cached thread pool that creates threads on demand
  - N threads that dequeue events and dispatch directly to CompletionHandler
- N defaults to the number of hardware threads.
- In addition to N threads, there is one additional internal thread that dequeues events and submits tasks to the thread pool to invoke completion handlers.

Tip # 3 – Kernel Thread Pool

Avoid blocking/long lived operations inside a completion handler.

Grizzly's implementation

- AIOHandler
  - Thread Pool are configurable
  - An application can test which one gives the best scalability/throughput.

AsynchronousSocketChannel.read()

> Once a connection has been accepted, it is now time to read some bytes:

```
AsynchronousSocketChannel.read(ByteBuffer b,
                               Attachement a,
                               CompletionHandler<> c);
```

> Hey Hey → You see the problem, right?
> Who remember when I was scared by the SelectionKey.attach()?

AsynchronousSocketChannel.read()

- Trouble trouble trouble:
  - Let's say you get 10 000 accepted connections
  - Hence 10 000 ByteBuffer created, and the read operations get invoked
  - Now we are waiting, waiting, waiting, waiting for the remote client(s) to send us bytes (slow clients/ network)
  - Another 10 000 requests comes in, and we are again creating 10 000 ByteBuffer and invoke the read() operations.
  - BOOM!

Tip #4: Use ByteBuffer pool & Throttle

- Let's not be too negative here. So far we have tested with more than 20 000 clients without any issues
- But this is still something you have to keep in mind!!
- Might want to throttle the read() operation to avoid the creation of too many ByteBuffer
- We strongly recommend the use of a ByteBuffer pool, specially if you are using Heap ByteBuffer (more on this later).
- Get a ByteBuffer before invoking the read() method, and return it to the pool once the read operations complete.

Blocking AsynchronousSocketChannel.read()

- Hey? Blocking?
- When invoking the read operation, the returned value is a Future:
  - Future readOp = AsynchronousSocketChannel.read(…);
  - readOp.get(30, TimeUnit.SECONDS);
- The Thread will block until the read operation complete or times out.
- Be careful as you might lock your ThreadPool (specially with FixedThreadPool)

Grizzly's implementation

- AIOContext - InputReader
  - Use a ByteBuffer Pool
  - Throttle Read Operations.
  - Use blocking for short read operations.

AsynchronousSocketChannel.write()

- Now let's execute some write operations:

```
AsynchronousSocketChannel.write(ByteBuffer b,
                                Attachement a,
                                CompletionHandler<> c);
```

  - Wait wait wait. Since we are asynchronous, invoking write(..) will not block, so the calling thread can continue its execution.
  - What happens when the calling thread invokes the write method again and the CompletionHandler has not yet been invoked by the previous write call?

AsynchronousSocketChannel.write()

- Aille!! You get a WritePendingException
- Hence when invoking the write operation, make sure the CompletionHandler.complete() has been invoked before initiating another write.
- Better, store ByteBuffer inside a queue and execute write operations only when the previous one has completed (will show code soon)
- As for read, we strongly recommend the use of a ByteBuffer pool for executing write operations. Get one before writing, put it back to the pool after.

Grizzly's implementation

- OutputWriter
  - Use a ByteBuffer Pool
  - FIFO Queue ByteBuffer
  - Allow blocking for write operations.

Damned ByteBuffer!

- If you are using Heap ByteBuffer, be aware the kernel will copy the bytes into a direct ByteBuffer during every write operation:
  - Free byte copy ☺
- Direct ByteBuffer performance have significantly improved with JDK 7, so use them all the time.
- Scattered ByteBuffer write operations still offer you free copy, using direct ByteBuffer or not!

AsynchronousFileChannel.open()

> Before, the nightmare:

```
File f = new File();

FileOutputStream fis = new
    FileOutputStream(f);

FileChannel fc = fis.getChannel();

fc.write(…);
```

…… typing so many lines hurts ☺

AsynchronousFileChannel.open()

- Now, the paradise

```
afc = AsynchronousFileChannel open(Path
   file, OpenOption... options);

afc.write(…);
```

Conclusion

- NIO.2 brings asynchronous I/O to the masses
- You can try it now!
- Try it using Project Grizzly, or look at the implementation to get started.

Companion Session

TS-5052: Hacking the File System with JDK™ Release 7, Thursday @ 10:50, Gateway 102-103.

BOF-5087: All Things I/O with JDK™ Release 7, Thursday @ 6:30pm, Gateway 102-103.

BOF-4611: Grizzly 2.0: Monster Reloaded! Wednesday @ 6:45pm, Hall E 134.

More Information – NIO.2

Open JDK NIO.2 page:
http://openjdk.java.net/projects/nio/

NIO.2 docs
http://openjdk.java.net/projects/nio/javadoc/

NIO.2 mailing list
nio-dev@openjdk.java.net

Alan's blog
http://blogs.sun.com/alanb/

More Information - Grizzly

Project Grizzly:
  http://grizzly.dev.java.net

Join the Grizzly's buzz
  users@grizzly.dev.java.net
  http://twitter.com/project_grizzly

Jeanfrancois' blog
  http://weblogs.java.net/blog/jfarcand/
  http://twitter.com/jfarcand

Grizzly's implementation

- FileWriter (http link will be added later, will use an IDE to show the code during the talk)

JavaOne[SM]

Thank You

Alan Bateman
Jeanfrancois Arcand
Sun Microsystems Inc.