OutOfMemory Errors

OutOfMemory Errors

Well, the most obvious cause of this is memory leaks in your application 🙂 But, if you've thoroughly investigated using tools like jconsole, yourkit, jprofiler or any of the other profiling and analysis tools out there and you can eliminate your code as the source of the problem, read on.

JVM bugs

Garbage Collection Problems

One symptom of a cluster of jvm related memory issues is the OOM exception accompanied by a message such as `"java.lang.OutOfMemoryError: requested xxxx bytes for xxx. Out of swap space?"`.

Sun bug number 4697804 describes how this can happen in the scenario when the garbage collector needs to allocate a bit more space during its run and tries to resize the heap but fails because the machine is out of swap space. One suggested work around is to ensure that the jvm never tries to resize the heap, by setting min heap size to max heap size:

```
java -Xmx1024m -Xms1024m
```

Another workaround is to ensure you have configured sufficient swap space on your device to accommodate all programs you are running concurrently.

Direct ByteBuffers

Another issue related to jvm bugs is the exhaustion of native memory. The symptoms to look out for are the process size growing, but the heap usage remaining relatively constant. Native memory can be consumed by a number of things, the JIT compiler being one, and nio ByteBuffers being another. Sun bug number 6210541 discusses a still-unsolved problem whereby the jvm itself allocates a direct ByteBuffer in some circumstances that is never garbage collected, effectively eating native memory. Guy Korland's blog discusses this problem here and here. As the JIT compiler is one consumer of native memory, the lack of available memory may manifest itself in the JIT as OutOfMemory exceptions such as `"Exception in thread "CompilerThread0" java.lang.OutOfMemoryError: requested xxx bytes for ChunkPool::allocate. Out of swap space?"`

By default, Jetty will allocate and manage its own pool of direct ByteBuffers for io if the nio SelectChannelConnector is configured. It also allocates MappedByteBuffers to memory-map static files via the DefaultServlet settings. However, you could be vulnerable to this jvm ByteBuffer allocation problem if you have disabled either of these options. For example, if you're on Windows, you may have disabled the use of memory-mapped buffers for the static file cache on the DefaultServlet to avoid the file-locking problem.

JSP bugs

Permgen problems

The JSP engine in Jetty is Jasper. This was originally developed under the Apache Tomcat project, but over time has been forked by many different projects. All jetty versions up to 6 used Apache-based Jasper exclusively, with Jetty 6 using Apache Jasper only for JSP2.0. With the advent of JSP 2.1, Jetty 6 switched to using Jasper from Sun's Glassfish project, which is now the reference implementation.

All forks of Jasper suffer from a problem whereby the permgen space can be put under pressure by using jsp tag files. This is because of the classloading architecture of the jsp implementation. Each jsp file is effectively compiled and its class loaded in its own classloader so as to allow for hot replacement. Each jsp that contains references to a tag file will compile the tag if necessary and then load it *using its own classloader*. If you have many jsps that refer to the same tag file, then the tag's class will be loaded over and over again into permgen space, once for each jsp. The relevant Glassfish bug report is bug # 3963, and the equivalent Apache Tomcat bug report is bug # 43878. The Apache Tomcat project has already closed this bug report with status WON'T FIX, however the Glassfish folks still have the bug report open and have scheduled it to be fixed. When the fix becomes available, the Jetty project will pick it up and incorporate into our release program.