ByteBuffers and Off-Heap Memory
64 Bits is BIG

- **16 exabytes aka 16 billion gigabytes**
  - Not big enough to individually address all the atoms in the universe

- **Current processors can’t actually address it all**
  - And operating systems impose their own limits

- **64-bit machines are ubiquitous**
  - 2012 Consumer PC: 8 Gb main memory, < $500
  - 1977: Cray-1: 64 bit data bus, 24-bit address bus, 8 Mb main memory, $8.8MM
InfoGraphic: What Are We Doing With All This Memory?

- Nuclear Weapons Research: 1%
- Large Datasets: 10%
- Web-app Caches: 10%
- FarmVille: 80%
The JVM Memory Map

```
java -Xms1024m -Xmx4096m com.example.Hello
```

```text
Reserved

/usr/bin/java

Java Heap
Internal Data

130+ Tb

Thread Stacks
Shared Libraries
Mapped JARs

```

```text
0000000004000000 36K r-x-- /usr/local/java/jdk-1.6-x64/bin/java
0000000004010000 8K rwx-- /usr/local/java/jdk-1.6-x64/bin/java
00000000040eba000 676K rwx-- [ anon ]
0000000006fae0000 21248K rwx-- [ anon ]
0000000006fc2c0000 62720K rwx-- [ anon ]
000000000700000000 699072K rwx-- [ anon ]
00000000072aab0000 2097152K rwx-- [ anon ]
0000000007aabb0000 349504K rwx-- [ anon ]
0000000007c0000000 1048576K rwx-- [ anon ]
00000007fae7bf0000 4K ------ [ anon ]
00000007fae7bf0000 1024K rwx-- [ anon ]
00000007fae7cf0000 12K ------ [ anon ]
00000007fae7d0000 1016K rwx-- [ anon ]
00000007fae7df0000 12K ------ [ anon ]
00000007faed000000 1652K r-x-- /usr/local/java/jdk-1.6-x64/jre/lib/rt.jar
... 00000007faef34aa0000 1576K r-x-- /lib/x86_64-linux-gnu/libc-2.13.so
00000007faef36340000 2044K ------ /lib/x86_64-linux-gnu/libc-2.13.so
00000007faef38330000 16K r-x-- /lib/x86_64-linux-gnu/libc-2.13.so
00000007faef39370000 4K rwx-- /lib/x86_64-linux-gnu/libc-2.13.so
... 00000007faef3e800000 4K r-x-- /lib/x86_64-linux-gnu/ld-2.13.so
00000007faef3e810000 8K rwx-- /lib/x86_64-linux-gnu/ld-2.13.so
00000007ffffffd5b0000 132K rwx-- [ stack ]
00000007ffffffd5ff0000 4K r-x-- [ anon ]
7fffffff600000 4K r-x-- [ anon ]
```

```
total 4478020K
```
java.nio.ByteBuffer

• Part of the JDK since 2002 (1.4)
  – Channel selectors got all the press

• Three flavors
  – non-direct (on-heap)
  – direct (off heap)
  – mapped (off heap, contents backed by file)

• References an unstructured block of memory
  – No pointers, must use indexes

• Limited to 2Gb each

• Not thread-safe
byte[] data = new byte[1024];
ByteBuffer buf1 = ByteBuffer.wrap(data);
ByteBuffer buf2 = ByteBuffer.allocate(1024);
ByteBuffer buf3 = ByteBuffer.allocateDirect(1024);

buf1.position(12);
buf1.putInt(0x12345678);
int x = buf1.getInt();

int y = buf1.getInt(12);
buf1.putInt(12, x + 231);
Practical Use: Off-Heap Web-App Cache

- **Cache stands between response body and database**
  - Entire pages
  - Pieces of pages
  - Data used to construct pages

- **Options**
  - External (Akamai, mod_cache)
  - Internal (servlet filter, map in application scope)
  - Distributed (Memcached, Terracotta, Coherence)

- **What we want**
  - In-process, to minimize cost of access
  - Off-heap, to minimize impact on garbage collector
How the Garbage Collector Works
The Problem with Internal Caches

• **Big Heap == Slow GC**
  – “Mark” phase increased by number of live objects
  – “Compact” phase increased by holes, size of remaining objects
  – If heap is almost full, may get into cycle of constant GC

• **Paging is BAD**
  – And you can’t control it
Solution: Off-Heap Cache

• **Heap just holds operational data**
  – Can be much smaller
  – More frequent collections, but less gets collected

• **Garbage collector doesn’t touch cache**
  – If data gets paged to disk, it stays there until needed

• **Cache can be larger than physical memory**
Issues in Implementing an Off-Heap Cache

- Accessing Non-Heap Memory
- Memory Management / Fragmentation
- Managing Cache
- Marshalling/UnmarshallingCached Data
Memory Management

• Explicitly assign/release memory
• Manage the freelist
• Prevent fragmentation
public class OffHeapStringCache {
    private int _maxLength;
    private LinkedHashMap<String, CharBuffer> _map;
    private LinkedList<CharBuffer> _freeList;

    public OffHeapStringCache(int size, int maxLength) {
        _maxLength = maxLength;

        _map = new LinkedHashMap<String, CharBuffer>(size, .75f, true);

        _freeList = new LinkedList<CharBuffer>();
        for (int ii = 0; ii < size; ii++) {
            CharBuffer buf = ByteBuffer.allocateDirect(2 * maxLength).asCharBuffer();
            _freeList.add(buf);
        }
    }
}
Managing Cached Data

• A cache acts like a Map
• But has fixed size

• And an expiration (eviction) strategy
  – Simple strategy: least recently used
  – More complex: least frequently used
  – Even more complex: least cost to recreate
public synchronized void put(String key, String value) {
    if (value.length() > _maxLength)
        throw new IllegalArgumentException("string too long: " + value.length());

    CharBuffer buf = _map.get(key);
    Don’t want to leak buffers!

    if (buf == null)
        buf = (_freeList.size() > 0) ? _freeList.removeFirst() : null;

    if (buf == null)
    {
        Entry<String, CharBuffer> eldest = _map.entrySet().iterator().next();
        buf = eldest.getValue();
        _map.remove(eldest.getKey());
        This just resets the buffer’s position/limit
    }

    buf.clear();
    buf.put(value);
    buf.limit(value.length());
    Must explicitly restrict the “active” portion of the buffer

    _map.put(key, buf);
}

Map iterates in LRU order
public synchronized String get(String key) {
    CharBuffer buf = _map.get(key);
    if (buf == null)
        return null;
    buf.position(0);
    return buf.toString();
}

Position will be updated by any read/write, so must be reset each time
Reconstituting Cached Data

• Simply copying data from cache to heap negates benefit of off-heap cache
  – Large arrays are stored directly in tenured generation
  – But it will be almost immediately eligible for collection

• Solution: move directly from cache to output
  – void write(String key, Writer out)
  – Alternative: return InputStream that wraps buffer

• Think about scope of synchronization!
public synchronized boolean write(String key, Writer writer) throws IOException {
    CharBuffer buf = _map.get(key);
    if ((buf == null) || (buf.limit() == 0))
        return false;

    buf.position(0);
    for (int ii = 0 ; ii < buf.limit() ; ii++)
        writer.write(buf.get(ii));

    return true;
}
Practical Use: Memory-Mapped Files

• File’s contents are mapped into process address space
  – Once page is loaded, all access is in-process

• Operating System loads/writes pages as needed
  – Unlike RandomAccessFile, which requires context switch

• Most useful when you access relatively small areas of the file repeatedly
  – Especially if file is large vis-à-vis available RAM
Example: Memory-Mapped JARs

- JAR directory is at end of file
  - Directory references individual entries by offset
  - Fast to copy entry data into array, pass to classloader

- Alternative: read file from start to finish
Using Memory-Mapped Files

- **Best for data that is structured as an array**
  - Or that has fixed-size offsets

- **Create Java class that acts as “view” on ByteBuffer**
  - Getter/Setter methods that use absolute buffer positions

- **Mapping is multi-step process**
  - Create Channel first, then map

- **ByteBuffer is limited to 2Gb, files can be bigger**
  - Option 1: individual mappings for sections of files
  - Option 2: create “megabuffer” that combines buffers, has similar API
Code Sample: Mapping a File

RandomAccessFile raf = new RandomAccessFile("/tmp/example.dat", "rw");
try {
    FileChannel channel = raf.getChannel();
    MappedByteBuffer buf = channel.map(MapMode.READ_WRITE, 8L, 16);
    // do something with buf
}
finally {
    raf.close();
}

RandomAccessFile allows both read and write

Section of file to map

Mapping could be read-only even if RAF is opened read-write

Mapping remains until file is closed
public class TempNodeData {
    private static int OFF_PVID = 0; // long
    private static int OFF_LAT = 8; // float
    private static int OFF_LON = 12; // float
    private static int OFF_MORTON = 16; // int

    public static int RECORDSIZE = 20;

    private ByteBuffer buffer;

    public TempNodeData(ByteBuffer srcBuf, int index) {
        srcBuf.position(index * RECORDSIZE);
        buffer = srcBuf.slice();
    }

    public long getPVID() {
        return buffer.getLong(OFF_PVID);
    }

    // ...

    // not thread safe!

    // slice() creates new buffer with same backing store
Example: Scaling Traffic Applications

- Navteq is integrating GPS/cellphone probe data into traffic model
  - Map-match, discover route, apply travel time to road segments

- Problem: size x volume
  - North America, South America, Europe total 100MM road segments
  - Peak design volume is 3 billion probe points/day

- Solution: files sorted by location hash
  - Nearby locations will be physically colocated on disk
  - Can limit probe traffic to small geographic area
  - Operating system loads only those pages that are needed
For More Information