INSIDE THE JAVA VIRTUAL MACHINE

Memory Management and Troubleshooting

Filip Hanik

Covalent Technologies

August 29, 2007

Who am I?

- fhanik@apache.org
- Tomcat Committer / ASF member
- Co-designed the Comet implementation
- Implemented NIO connector in 6
- Responsible for session replication and clustering
- Been involved with ASF since 2001
- Member, Covalent Technical Team

What are we Talking About?

- Internals of Java Memory
- Spoken from a Java developer’s standpoint
- For other Java developers and system administrators

Agenda

- Understanding the Java Memory Layout
- Out Of Memory Errors
  - Causes
  - Solution
- Garbage Collection Basics
- Java Tuning Options – Time Constraint
- Questions and Answers

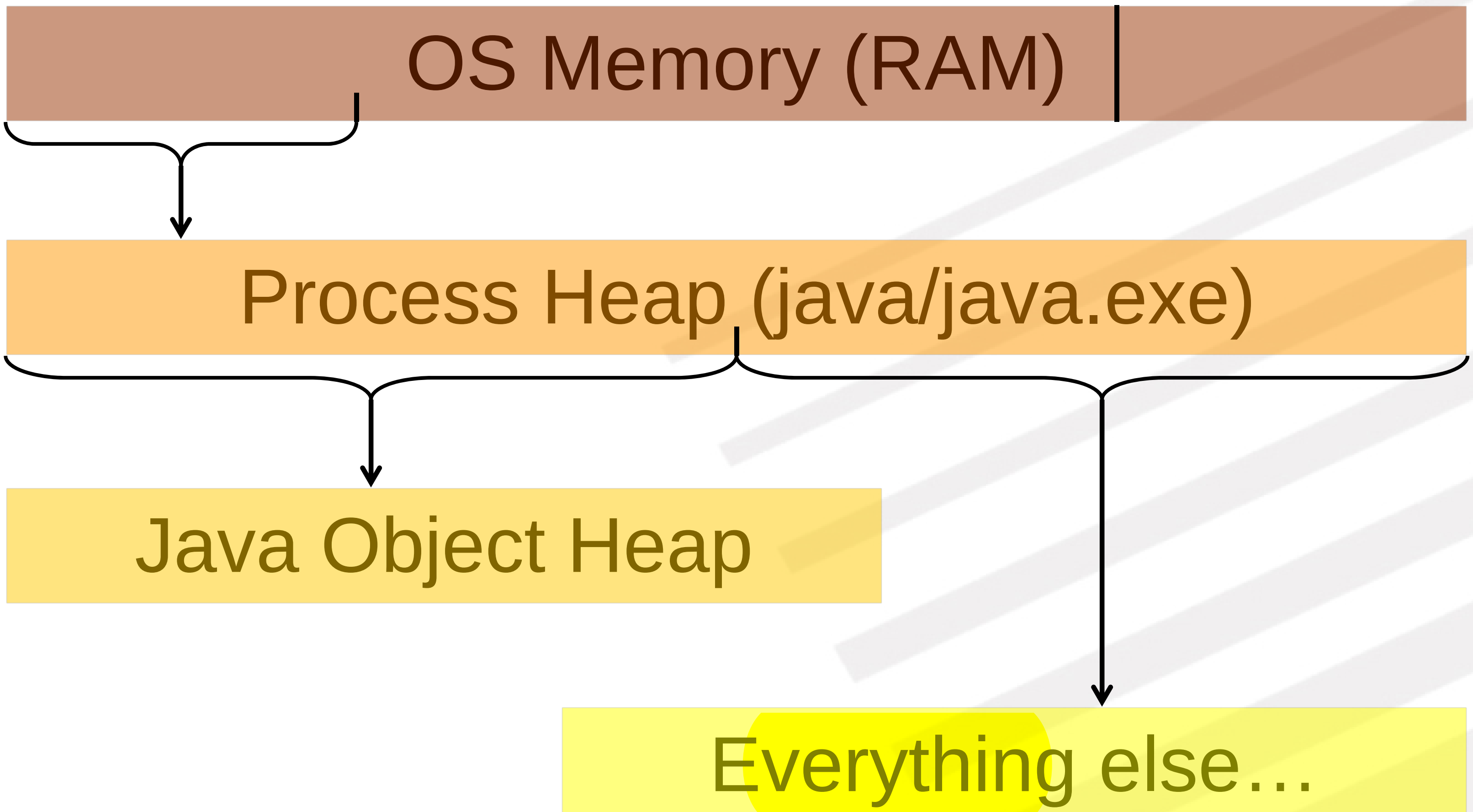
Storing Data in Memory

- Java runs as a single process
  - Does not share memory with other processes
- Each process allocates memory
  - We call this process heap
- Ways to allocate memory in a process
  - C (malloc and free)
  - C++ (new and delete)
  - Java (new and dereference -> Garbage Collection)

Storing Data in Memory

- JVM manages the process heap
  - In most cases
  - JNI managed memory would be an exception, and there are others
- No shared memory between processes
  - At least not available through the Java API
- JVM creates a Java Heap
  - Part of the process heap
  - Configured through –Xmx and –Xms settings

The JVM Process Heap



JVM Process Heap

- Maximum size is limited
  - 32 bit size, roughly 2GB
  - 64 bit, much much larger ☺
- If 2GB is the max for the process
  - -Xmx1800m –Xms1800m – not very good
  - Leaves no room for anything else

Java Object Heap

- Also referred to as Java Heap
  - Often confused with JVM process heap
- Stores Java Objects
  - instances of classes
  - and the data the objects contain
    - Primitives
    - References

Benefits of the Java Heap

- Pre-allocate large blocks of memory
- Allocation of small amounts of memory is very fast
- No need to fish for a free memory segment in RAM
- No fragmentation
- Continuous memory blocks can make a big difference
- NullPointerException vs. General Access Fault
  - NPE runtime error
  - GAF crash the process

Gotcha #1

- -Xmx, -Xms and –Xmn
  - Only controls the Java Object Heap
  - Often misunderstood to control the process heap
- Confusion leads to incorrect tuning
  - And in some cases, the situation worsens

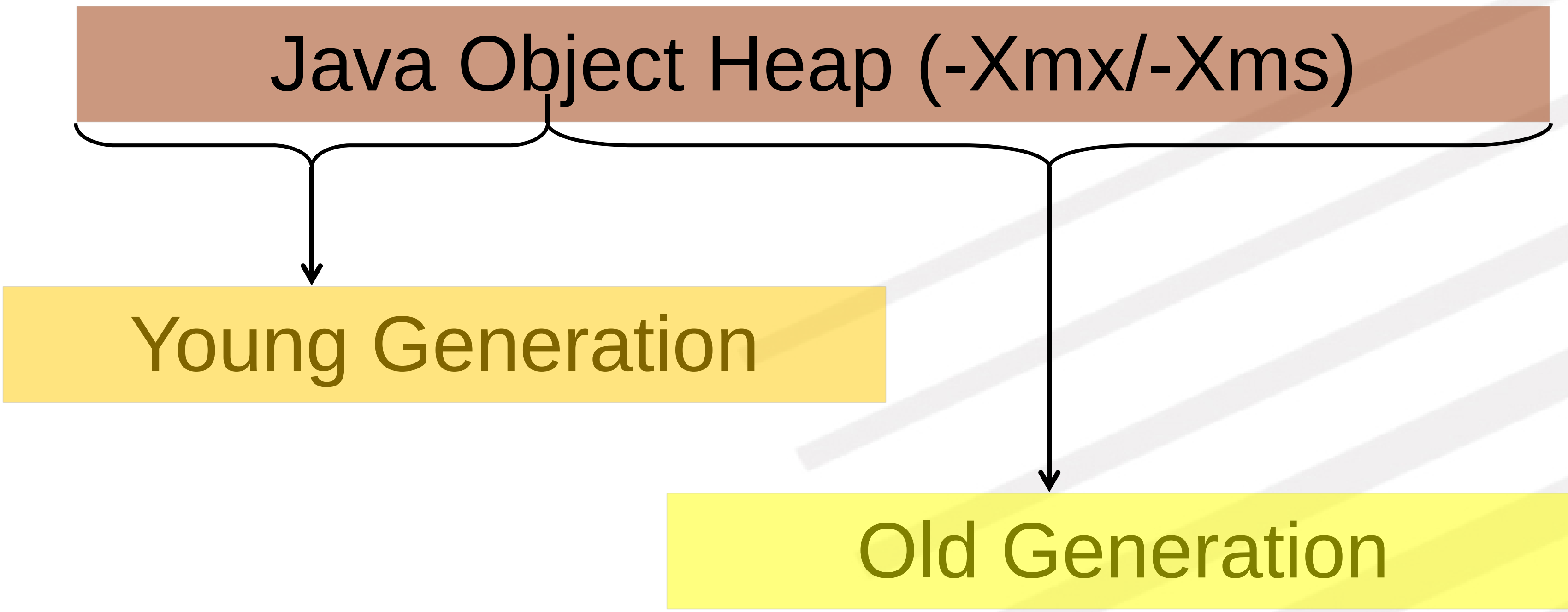
Java Object Heap

- So how is the Java Heap allocated?
- -XX:MinHeapFreeRatio=
  - Default is 40 (40%)
  - When the JVM allocates memory, it allocates enough to get 40% free
  - Huge chunks, very large default
  - Not important when –Xms == -Xmx
- -XX:MaxHeapFreeRatio=
  - Default 70%
  - To avoid over allocation
  - To give back memory not used
- As you can see, to provide performance and avoid fragmentation, excessively large blocks are allocated each time

Java Object Heap

- Object allocation statistics
  - 80-98% of newly allocated are extremely short lived (few million instructions)
  - 80-98% die before another megabyte has been allocated
  - Typical programs
- Tomcat Core (no webapps)
  - Lots of long lived objects
  - Still a small memory footprint

Java Object Heap

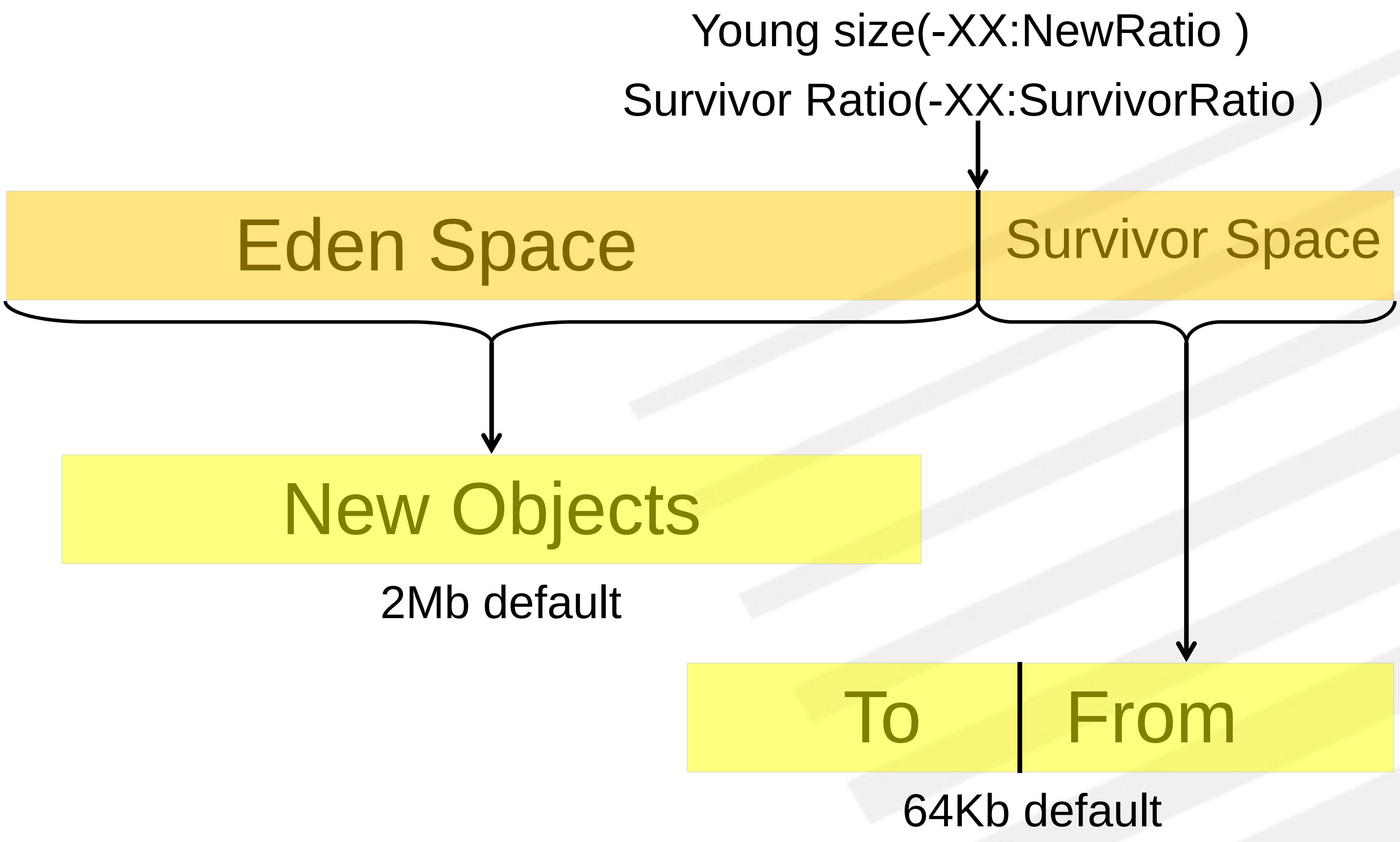


A good size for the YG is 33% of the total heap

Java Object Heap

- Young Generation
  - All new objects are created here
  - Only moved to Old Gen if they survive one or more minor GC
  - Sized Using
    - -Xmn – not preferred (fixed value)
    - -XX:NewRatio=<value> - preferred (dynamic)
- Survivor Spaces
  - 2, used during the GC algorithm (minor collections)

Young Generation



Gotcha #2

- Problem
  - Multithreaded apps create new objects at the same time
  - New objects are always created in the EDEN space
  - During object creation, memory is locked
  - On a multi CPU machine (threads run concurrently) there can be contention

Gotcha #2

- Solution
  - Allow each thread to have a private piece of the EDEN space
- Thread Local Allocation Buffer
  - -XX:+UseTLAB
  - -XX:TLABSize=<size in kb>
  - -XX:+ResizeTLAB
  - (On by default on multi CPU machines and newer JDK)
- Analyse TLAB usage
  - -XX:+PrintTLAB
- JDK 1.5 and higher (GC ergonomics)
  - Dynamic sizing algorithm, tuned to each thread

Old Generation

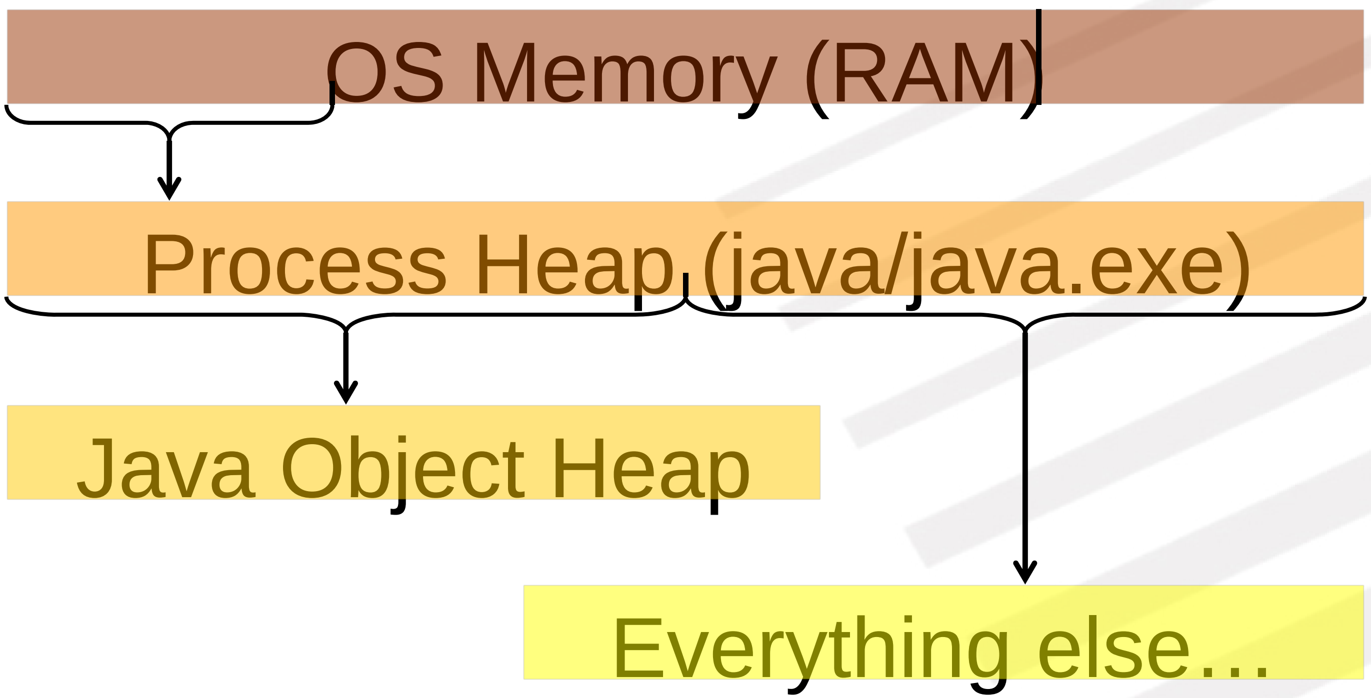
Tenured Space

5Mb min 44Mb max (default)

Garbage collection presentation will explain in detail how these spaces are used during the GC process.

JVM Process Heap

- Java Object Heap
  - A handful, but a small part of the story



JVM Process Heap
Everything else…

- Permanent Space
- Code Generation
- Socket Buffers
- Thread Stacks
- Direct Memory Space
- JNI Code
- Garbage Collection
- JNI Allocated Memory

Permanent Space

- Permanent Generation
  - Permanent Space (name for it)
  - 4Mb initial, 64Mb max
  - Stores classes, methods and other meta data
  - -XX:PermSize=<value> (initial)
  - -XX:MaxPermSize=<value> (max)
- Common OOM for webapp reloads
- Separate space for pre-historic reasons
  - Early days of Java, class GC was not common, reduces size of the Java Heap

Gotcha #3

- Permanent Space Memory Errors
  - Too many classes loaded
  - Classes are not being GC:ed
  - Unaffected by –Xmx flag
- Identified by
  - java.lang.OutOfMemoryError: PermGen space
- Many situations, increasing max perm size will help
  - i.e., no leak, but just not enough memory
  - Others will require to fix the leak

JVM Process Heap
Everything else…

- Permanent Space
- Code Generation
- Socket Buffers
- Thread Stacks
- Direct Memory Space
- JNI Code
- Garbage Collection
- JNI Allocated Memory

Code Generation

- Converting byte code into native code
- Very rare to cause memory problems
- JVM will most likely crash if it doesn’t have enough mem for this operation
  - Never seen it though

JVM Process Heap
Everything else…

- Permanent Space
- Code Generation
- Socket Buffers
- Thread Stacks
- Direct Memory Space
- JNI Code
- Garbage Collection
- JNI Allocated Memory

TCP connections

- Each connection contains two buffers
  - Receive buffer ~37k
  - Send buffer ~25k
- Configured in Java code
  - So might not be exposed through applications configuration
- Usually hit other limits than memory before an error happen
  - IOException: Too many open files (for example)

JVM Process Heap

Everything else…

- Permanent Space
- Code Generation
- Socket Buffers
- Thread Stacks
- Direct Memory Space
- JNI Code
- Garbage Collection
- JNI Allocated Memory

Thread Stacks

- Each thread has a separate memory space called “thread stack”
- Configured by –Xss
- Default value depends on OS/JVM
- As number of threads increase, memory usage increases

Gotcha #4

- java.lang.OutOfMemoryError:
  unable to create new native thread
- Solution
  - Decrease –Xmx and/or
  - Decrease –Xss
  - Or, you have a thread leak, fix the program
- Gotcha
  - Increasing –Xmx (32bit systems) will leave less room for threads if it is being used, hence the opposite of the solution
  - Too low –Xss value can cause java.lang.StackOverflowError

JVM Process Heap
Everything else…

- Permanent Space
- Code Generation
- Socket Buffers
- Thread stacks
- Direct Memory Space
- JNI Code
- Garbage Collection
- JNI Allocated Memory

Direct Memory Space

- Ability to let Java developers map memory outside the Java Object Heap
- java.nio.ByteBuffer.allocateDirect
- java.lang.OutOfMemoryError:
  Direct buffer memory
- Adjusted by
  - -XX:MaxDirectMemorySize=<value>

JVM Process Heap
Everything else…

- Permanent Space
- Code Generation
- Socket Buffers
- Thread stacks
- Direct Memory Space
- JNI Code
- Garbage Collection
- JNI Allocated Memory

JNI

- Code needs memory
  - Usually very little
- JNI programs also allocate memory
  - Error allocating memory.[NativeMemory.c] (my code)
  - JVM goes berserk or crashes or if the JNI code can handle it gracefully, you’re lucky
- Linux way of dealing with mem leak
  - Kill the process!

JVM Process Heap
Everything else…

- Permanent Space
- Code Generation
- Socket Buffers
- Thread stacks
- Direct Memory Space
- JNI Code
- Garbage Collection
- JNI allocated memory

Garbage Collection

- Also uses memory
  - Threads
  - Memory to store GC info
- If there isn't enough memory for GC, then the system will not be functioning at all

GC History

- First time around 1959 – LISP language
- The idea
  - automatic memory cleanup
  - Easier to write code
  - Easier to debug
- What it does
  - Maps memory in memory
  - The Java Object Heap is such a map

Phases of GC

- Lock it down
  - All objects that are to take part in the GC must be locked, so that they don't mutate
- Mark
  - Iterate through all objects
  - Mark the "unreachable" as garbage
- Sweep
  - Remove all previously marked objects
  - Reclaim memory

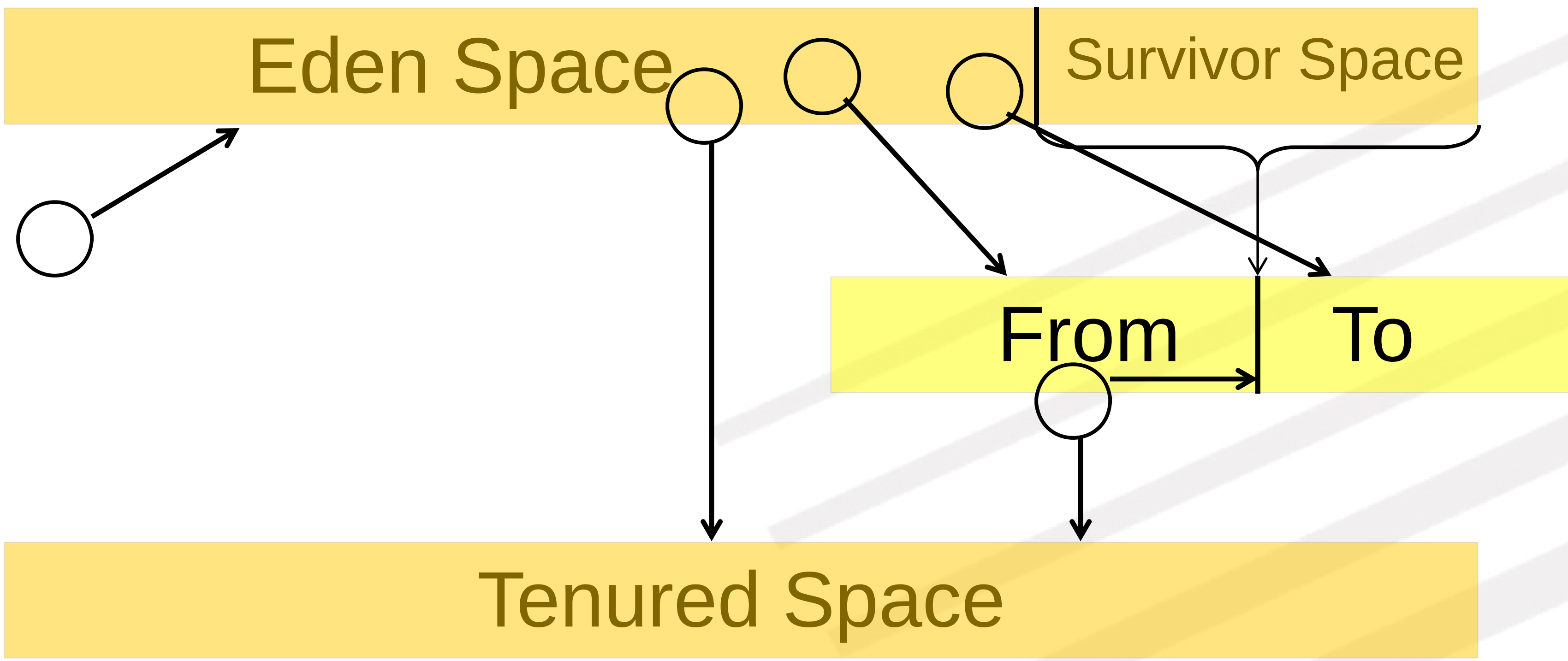
Early Version of Java

- Garbage Collector wasn't very well tuned
- Only one algorithm was available
- Mark and Sweep entire heap
  - Takes a very long time
  - Time spent is dependent on the size of the heap
  - That is why the "Permanent Space" was invented
    - And cause un/reloading of classes wasn't very common either
- Also known as "stop-the-world" gc
  - The entire JVM is locked down

Strategies

- Stop The World
- Incremental
  - Time GC with new object creation
  - If GC runs, suspend new allocation
- Concurrent/Parallel
  - Allocation happens at the same time as GC
  - Very complex locking regimes
  - Generations/Spaces make it easier
- CMS stands for
  - Concurrent
  - Mark
  - Sweep

How It Works



1. New objects created
2. When EDEN is full – minor collection
3. Copy surviving objects into 1st survivor space
4. Next time Eden fills
   Copy from Eden to 2nd
   Copy from 1st to 2nd
5. If 2nd fills and objects remain in Eden or 1st
   These get copied to the tenured

covalent

How it Works

- One survivor space is always empty
  - Serves as destination for minor collections
- Objects get copied to the tenured space when the 2nd survivor space fills up
- Major collections occur when the tenured space fills up
  - Major collections free up Eden and both survivor spaces

New and Fancy

- Concurrent/Parallel Garbage Collection
- -XX:+UseParNewGC
  - Parallel GC in the New(Young) Generation
- -XX:+UseConcMarkSweepGC
  - Concurrent in the Old generation
- Use these two combined
  - Multi CPU box can take advantage of this

Sun Recommended

- GC Settings
  - -XX:+UseConcMarkSweepGC
  - -XX:+CMSIncrementalMode
  - -XX:+CMSIncrementalPacing
  - -XX:CMSIncrementalDutyCycleMin=0
  - -XX:+CMSIncrementalDutyCycle=10
  - -XX:+UseParNewGC
  - -XX:+CMSPermGenSweepingEnabled
- To analyze what is going on
  - -XX:+PrintGCDetails
  - -XX:+PrintGCTimeStamps
  - -XX:-TraceClassUnloading

Minor Notes

- -XX:+UseParallelGC <> -XX:+UseParNewGC
- -XX:ParallelGCThreads=<nr of cpu>
  - Use with ParallelGC setting
- If you have 4 cpus and 1 JVM
  - Set value to 4
- If you have 4 cpus and 2 JVM
  - Set value to 2
- If you have 4 cpus and 6 JVM
  - Set value to 2

GC Ergonomics

- Started with JDK 1.5
- JVM is self trained and GC strategy adapts
- Rules/Guidelines can be set using command line options
  - Max pause time goal
    - The longest pause time to suspend application
  - Throughput goal
    - Time spent GC vs. time spent outside GC
- Not guaranteed

Out Of Memory Errors

- There is a seamless way to get info
  - -XX:+HeapDumpOnOutOfMemoryError
- No performance impact during runtime
- Dumping a –Xmx512m heap
  - Create a 512MB .hprof file
  - JVM is “dead” during dumping
  - Restarting JVM during this dump will cause unusable .hprof file

Gotcha’s

- Major collections don’t run until tenured is full
- What does that mean?
  - -Xmx1024m
  - Current heap could be 750MB
  - 500MB of “dead” objects
  - If VM is idle, could stay like that for a very long time
  - Wasting 500MB of RAM for an idle JVM

Monitoring Agents

- Monitor memory usage
- If system is idle, force a GC
- Can be done automatically and with remote agents
- Example would be:
- www.yourkit.com
- And script the client to take telemetry readings from an embedded agent

Thank You

- fhanik@covalent.net
- For support information contact Covalent
  - support@covalent.com, sales@covalent.com
  - www.covalent.com
  - 800/444-1935 or 925/974-8800
- Question and Answer time!!

covalent™

The Enterprise Open Source Support Company