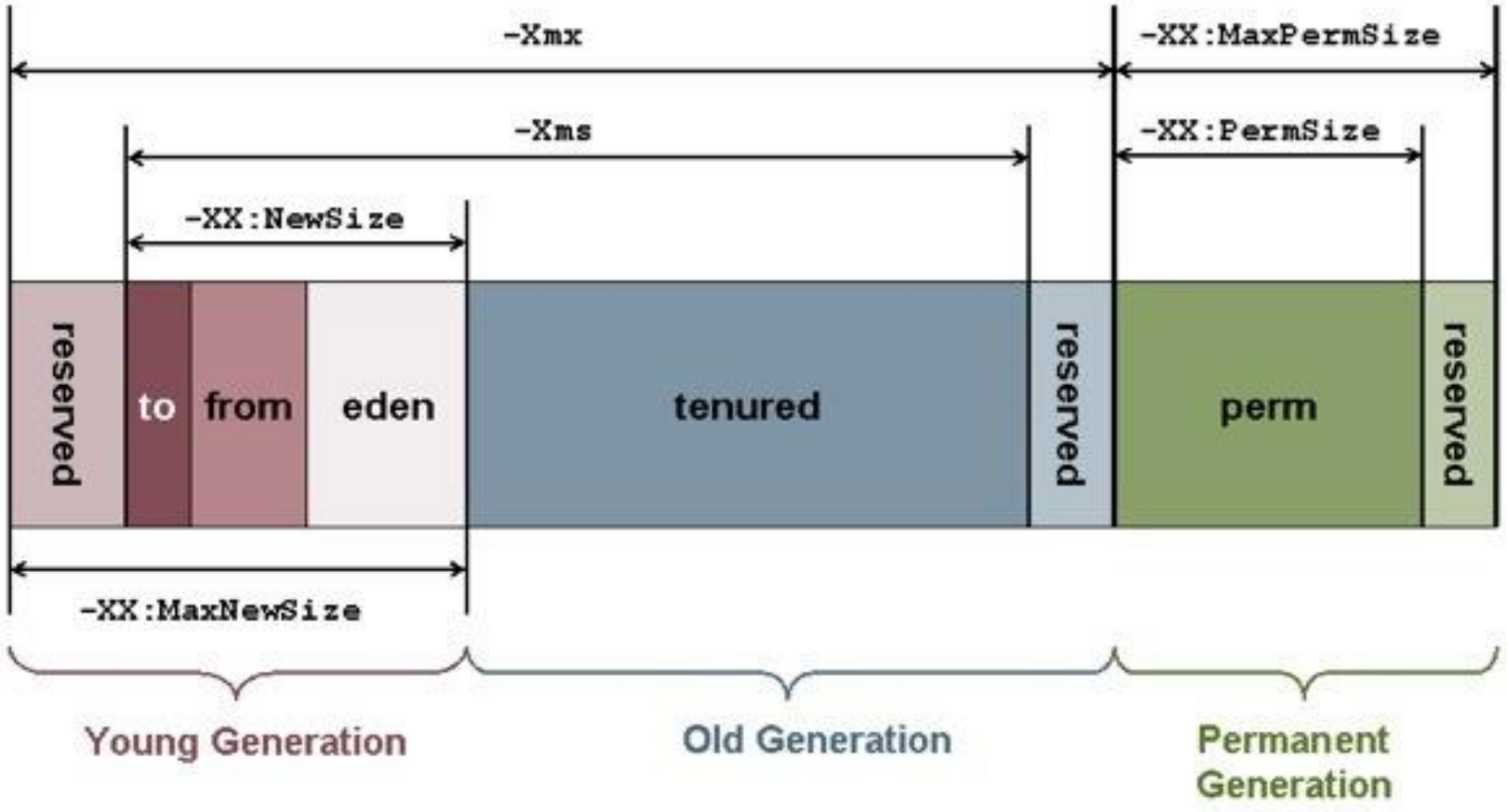
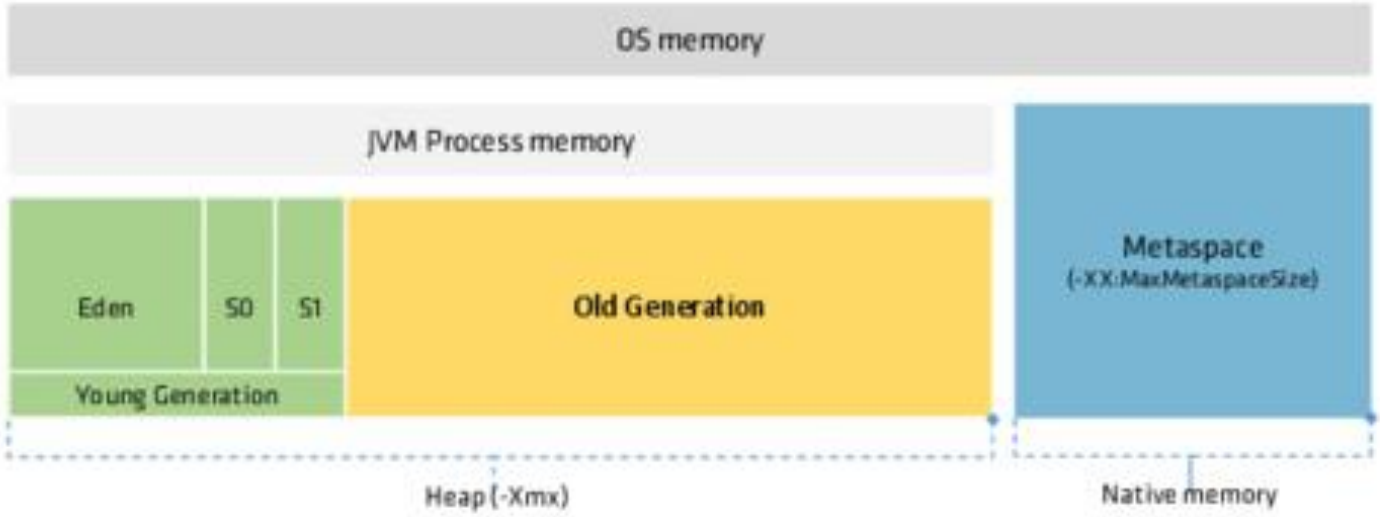
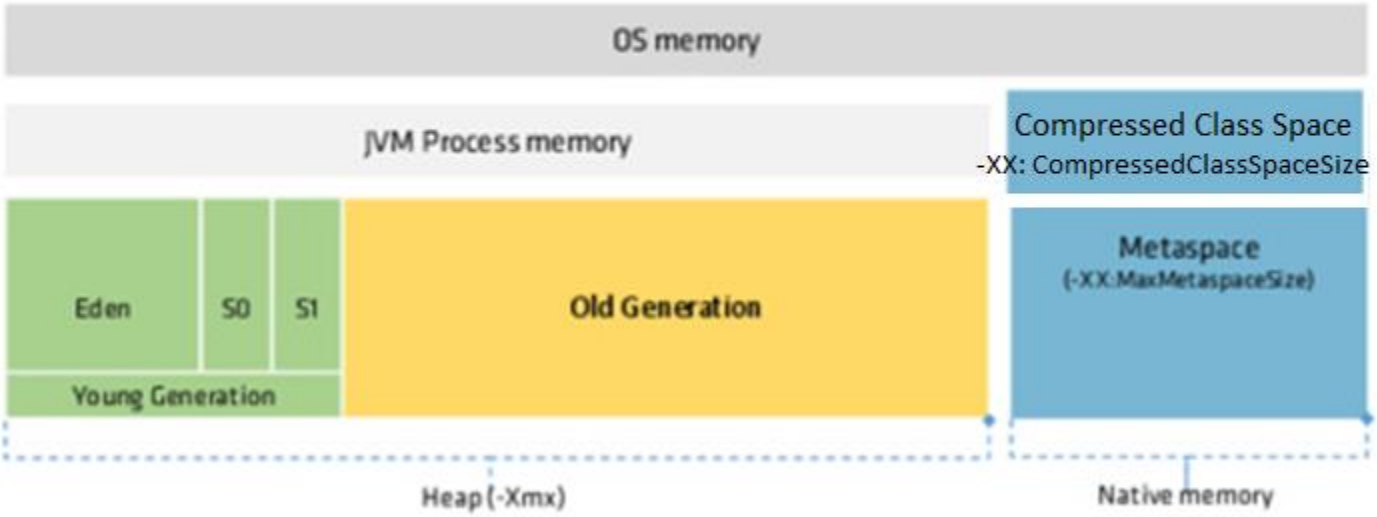
Java Memory Management


-Xmx
-XX:MaxPermSize
-Xms
-XX:PermSize
-XX:NewSize
reserved
to
from
eden
tenured
reserved
perm
reserved
-XX:MaxNewSize
Young Generation
Old Generation
Permanent Generation

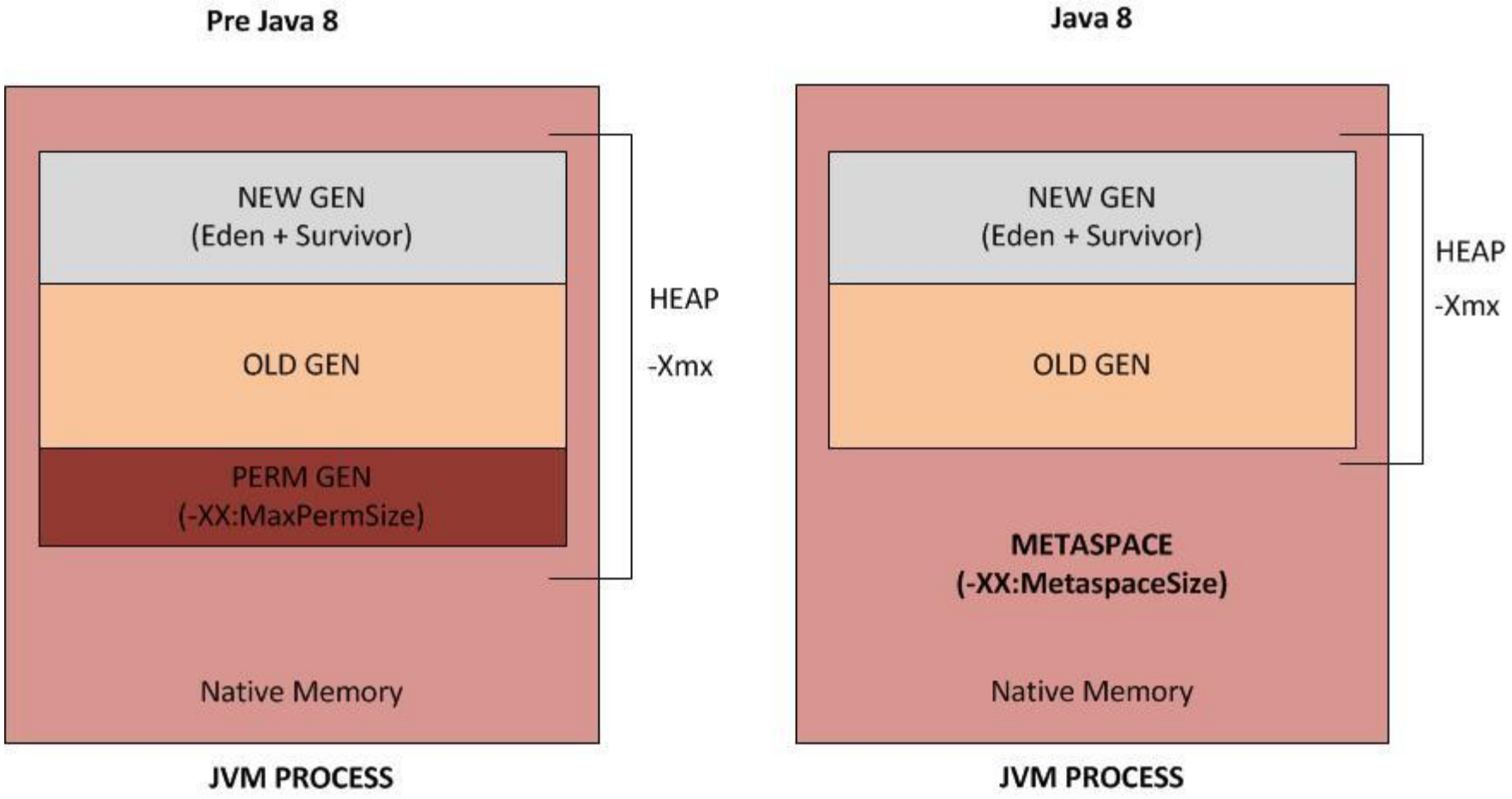
JVM memory
JDK8



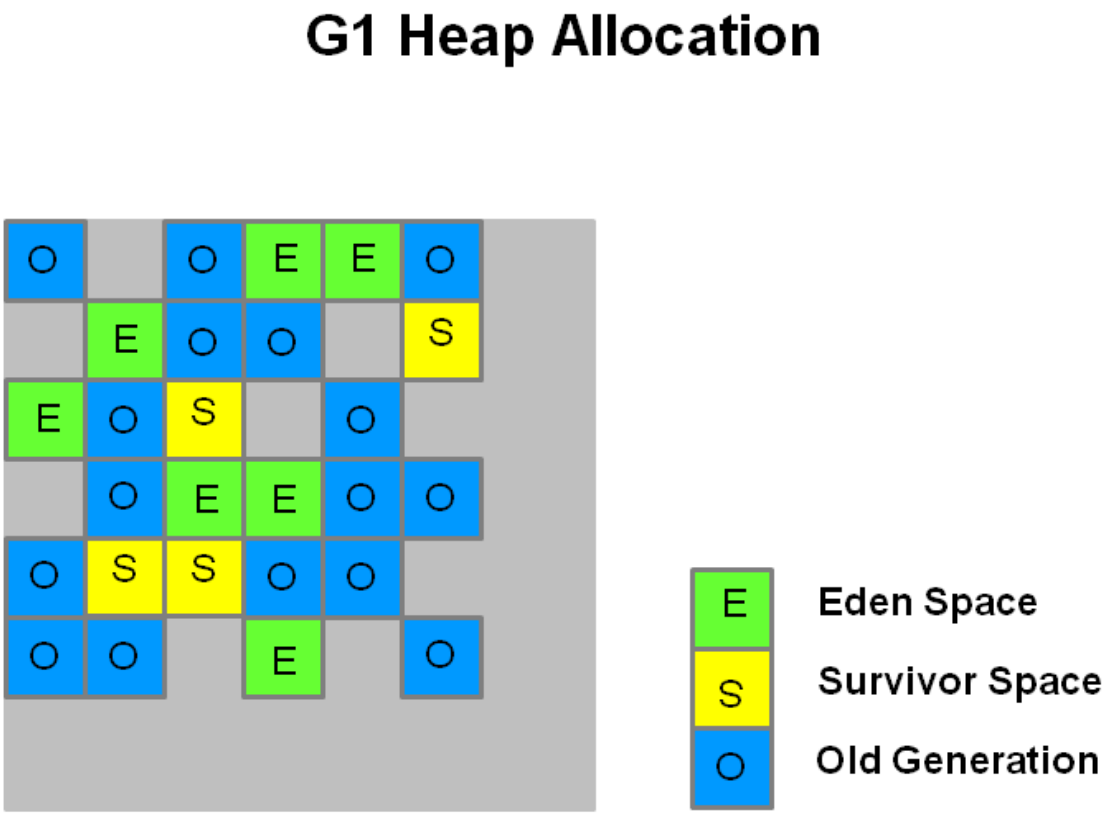
JVM memory
JDK8



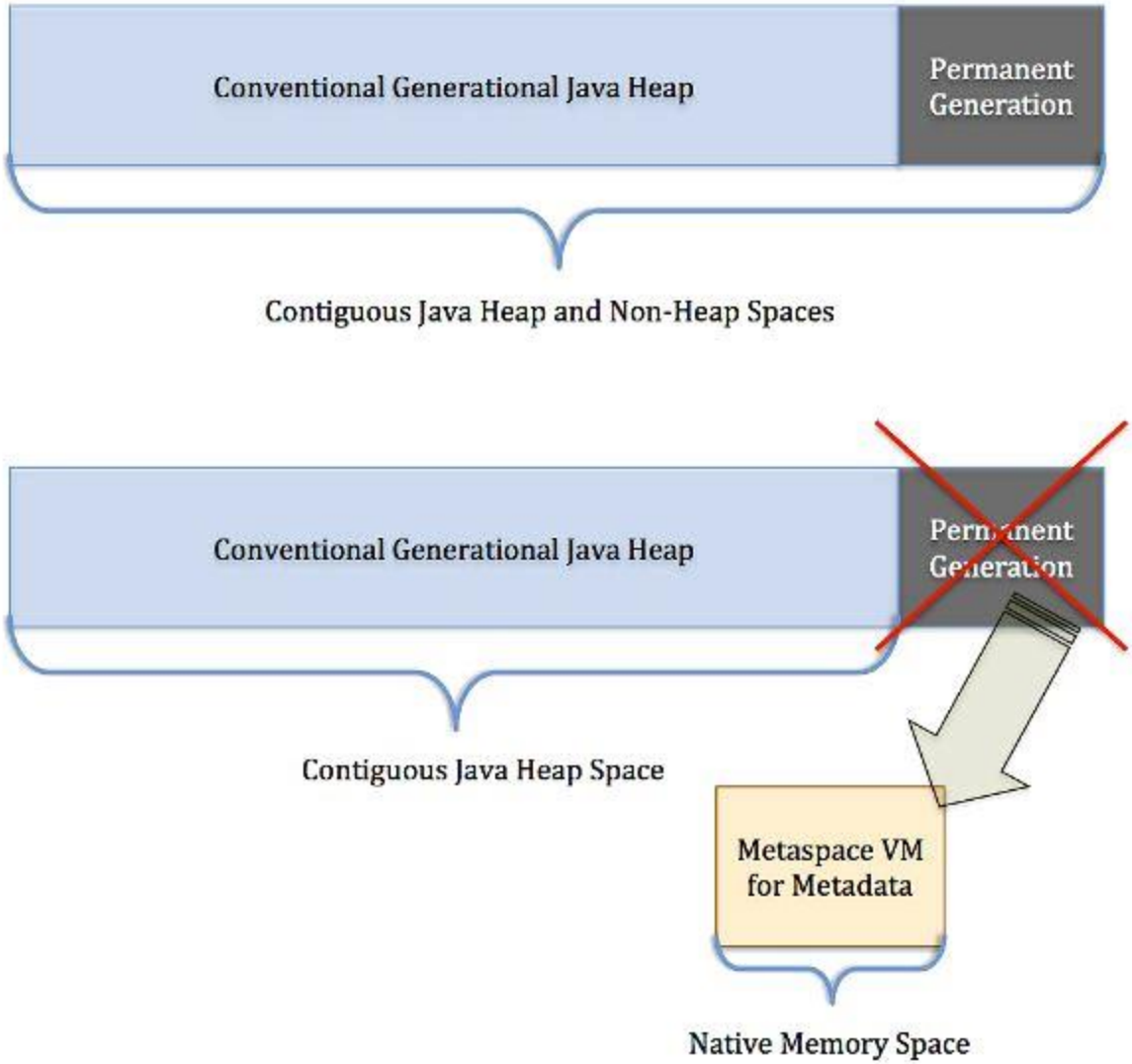
JAVA 8 MEMORY MANAGEMENT

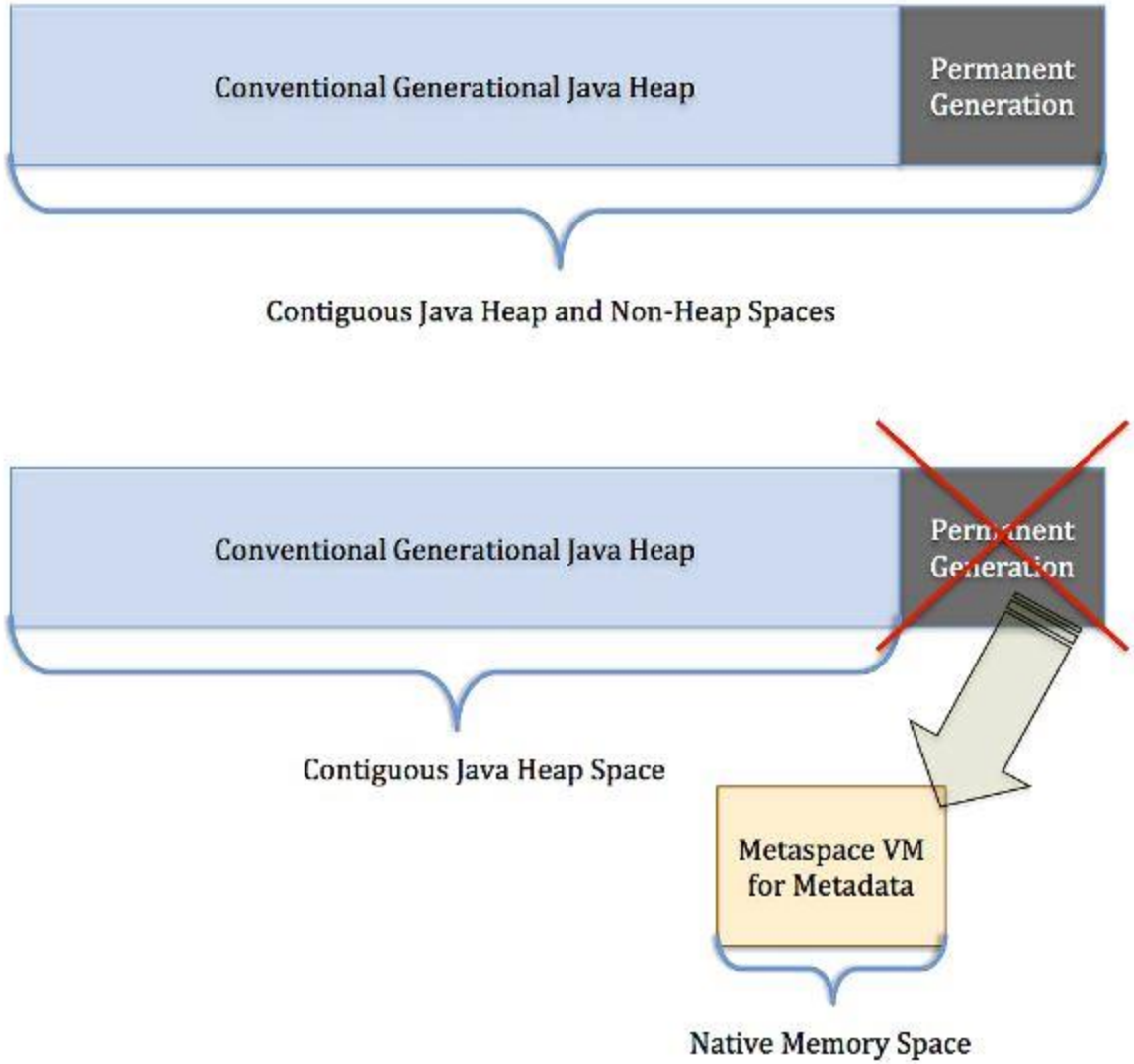


The heap is one memory area split into many fixed sized regions.



PermGen vs MetaSpace


Conventional Generational Java Heap
Permanent Generation
Contiguous Java Heap and Non-Heap Spaces
Conventional Generational Java Heap
Contiguous Java Heap Space
Metaspace VM for Metadata
Native Memory Space


Conventional Generational Java Heap
Permanent Generation
Contiguous Java Heap and Non-Heap Spaces
Conventional Generational Java Heap
Contiguous Java Heap Space
Metaspace VM for Metadata
Native Memory Space

What Does The Removal Of Permanent Space Mean To The End Users?

Since the class metadata is allocated out of native memory, the max available space is the total available system memory. Thus, you will no longer encounter OOM errors

jmap –clstats <PID>: prints class loader statistics.

Java Heap Space and Stack Memory

Java Heap Space

Java Heap space is used by java runtime to allocate memory to Objects and JRE classes. Whenever we create any object, it's always created in the Heap space.

Garbage Collection runs on the heap memory to free the memory used by objects that doesn't have any reference.

Any object created in the heap space has global access and can be referenced from anywhere of the application.

Java Stack Memory

Java Stack memory is used for execution of a thread.

They contain method specific values that are short-lived and references to other objects in the heap that are getting referred from the method.

Stack memory is always referenced in LIFO (Last-In-First-Out) order.

Whenever a method is invoked, a new block is created in the stack memory for the method to hold local primitive values and reference to other objects in the method.

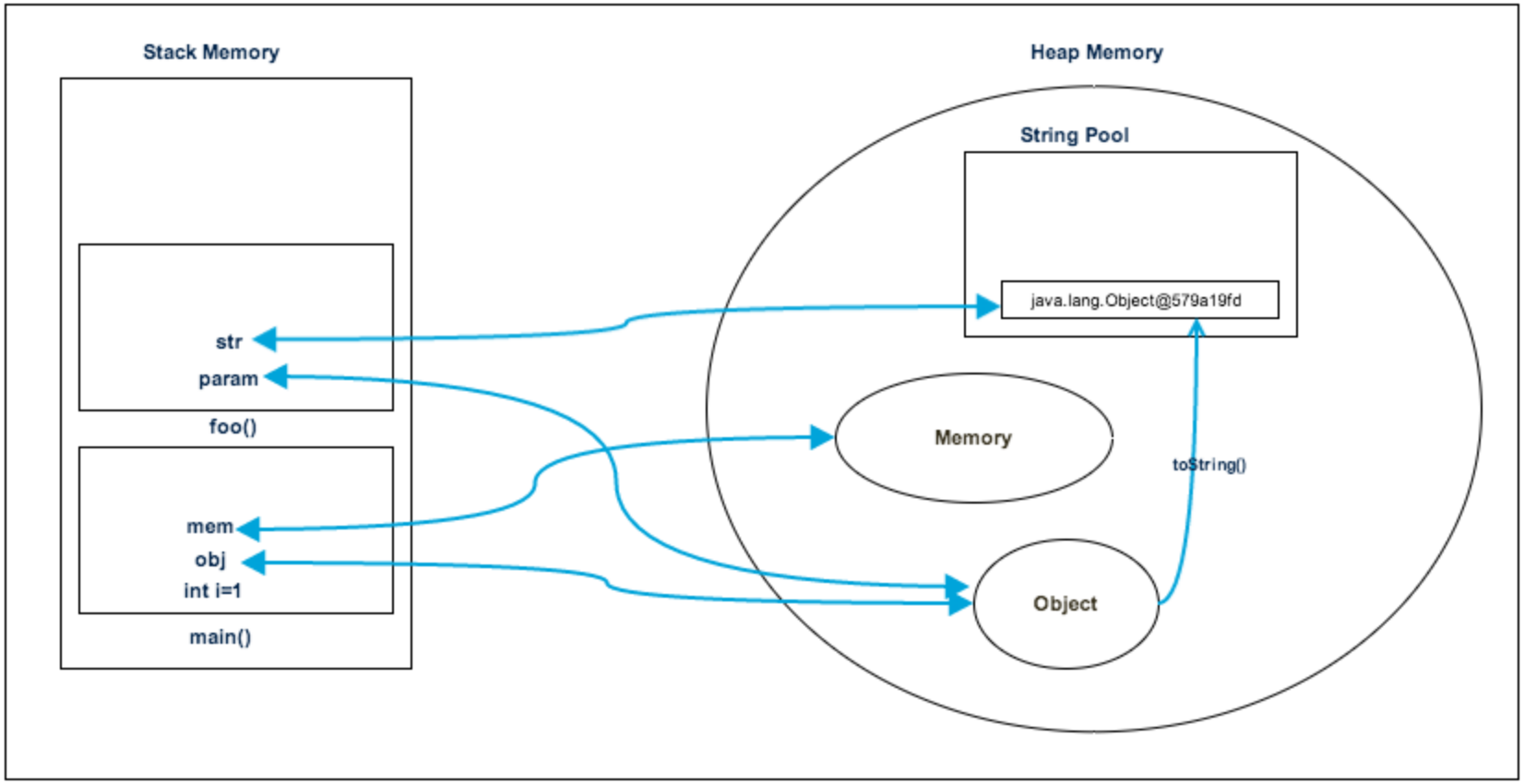
As soon as method ends, the block becomes unused and become available for next method.
Stack memory size is very less compared to Heap memory.

```
public class Memory {

        public static void main(String[] args) { // Line 1
                int i=1; // Line 2
                Object obj = new Object(); // Line 3
                Memory mem = new Memory(); // Line 4
                mem.foo(obj); // Line 5
        } // Line 9

        private void foo(Object param) { // Line 6
                String str = param.toString(); //// Line 7
                System.out.println(str);
        } // Line 8

}
```


Stack Memory
Heap Memory
String Pool
java.lang.Object@579a19fd
str
param
foo()
Memory
toString()
mem
obj
int i=1
main()
Object
Java Runtime Memory

steps of execution of the program

As soon as we run the program, it loads all the Runtime classes into the Heap space. When main() method is found at line 1, Java Runtime creates stack memory to be used by main() method thread.

We are creating primitive local variable at line 2, so it's created and stored in the stack memory of main() method. Since we are creating an Object in line 3, it's created in Heap memory and stack memory contains the reference for it. Similar process occurs when we create Memory object in line 4.

Now when we call foo() method in line 5, a block in the top of the stack is created to be used by foo() method. Since Java is pass by value, a new reference to Object is created in the foo() stack block in line 6.

A string is created in line 7, it goes in the String Pool in the heap space and a reference is created in the foo() stack space for it.

foo() method is terminated in line 8, at this time memory block allocated for foo() in stack becomes free.

In line 9, main() method terminates and the stack memory created for main() method is destroyed. Also the program ends at this line, hence Java Runtime frees all the memory and end the execution of the program.

Difference between Java Heap Space and Stack Memory

Heap memory is used by all the parts of the application whereas stack memory is used only by one thread of execution.

Whenever an object is created, it’s always stored in the Heap space and stack memory contains the reference to it. Stack memory only contains local primitive variables and reference variables to objects in heap space.

Objects stored in the heap are globally accessible whereas stack memory can’t be accessed by other threads.

Memory management in stack is done in LIFO manner whereas it’s more complex in Heap memory because it’s used globally. Heap memory is divided into Young-Generation, Old-Generation etc, more details at Java Garbage Collection.

Stack memory is short-lived whereas heap memory lives from the start till the end of application execution.

We can use -Xms and -Xmx JVM option to define the startup size and maximum size of heap memory. We can use -Xss to define the stack memory size.

When stack memory is full, Java runtime throws java.lang.StackOverFlowError whereas if heap memory is full, it throws java.lang.OutOfMemoryError: Java Heap Space error.

Stack memory size is very less when compared to Heap memory. Because of simplicity in memory allocation (LIFO), stack memory is very fast when compared to heap memory.

Native Memory

Native Memory is an area which is usually used by the JVM for it’s internal operations and to execute the JNI codes.

The JVM Uses Native Memory for Code Optimization and for loading the classes and libraries along with the intermediate code generation.

The Size of the Native Memory depends on the Architecture of the Operating System and the amount of memory which is already commited to the Java Heap.

Native memory is an Process Area where the JNI codes gets loaded or JVM Libraries gets loaded or the native Performance packs and the Proxy Modules gets loaded. There is no JVM Option available to size the Native Area. but we can calculate it approximately using the following formula:

NativeMemory = (ProcessSize – MaxHeapSize – MaxPermSize)

How to Use Native Memory Tracking

First enable NMT and then use jcmd to access the data collected thus far.

Enable NMT

Enable NMT using the following command line. Note that enabling this will cause 5-10% performance overhead.

-XX:NativeMemoryTracking=[off | summary | detail]

Use jcmd to Access NMT Data

Use jcmd to dump the data collected and optionally compare it to the last baseline.

jcmd <pid> VM.native_memory [summary | detail | baseline | summary.diff | detail.diff | shutdown] [scale= KB | MB | GB]

summary	Print a summary aggregated by category.
detail	•Print memory usage aggregated by category •Print virtual memory map •Print memory usage aggregated by call site
baseline	Create a new memory usage snapshot to diff against.
summary.diff	Print a new summary report against the last baseline.
detail.diff	Print a new detail report against the last baseline.
shutdown	Shutdown NMT.

Category	Description
Java Heap	The heap where your objects live
Class	Class meta data
Code	Generated code
GC	data use by the GC, such as card table
Compiler	Memory used by the compiler when generating code
Symbol	Symbols
Memory Tracking	Memory used by NMT itself
Pooled Free Chunks	Memory used by chunks in the arena chunk pool
Shared space for classes	Memory mapped to class data sharing archive

Thread	Memory used by threads, including thread data structure, resource area and handle area and so on.
Thread stack	Thread stack. It is marked as committed memory, but it might not be completely committed by the OS
Internal	Memory that does not fit the previous categories, such as the memory used by the command line parser, JVMTI, properties and so on.
Unknown	When memory category can not be determined. Arena: When arena is used as a stack or value object Virtual Memory: When type information has not yet arrived

NMT over time
(Use NMT to Detect a Memory Leak**)**

NMT also allows you to track how memory allocations occur over time. After the JVM is started with NMT enabled, we can establish a baseline for memory usage with this command:

% **jcmd process_id VM.native_memory baseline**
That causes the JVM to mark its current memory allocations. Later, wecan compare the current memory usage to that mark

% **jcmd process_id VM.native_memory summary.diff**

Native Memory Tracking:
Total: reserved=5896078KB -3655KB,
committed=2358357KB -448047KB
- Java Heap (reserved=4194304KB, committed=1920512KB
-444927KB)
(mmap: reserved=4194304KB, committed=1920512KB -
444927KB)

In this case, the JVM has reserved 5.8 GB of memory and is presently using 2.3 GB.

That committed size is 448 MB less than when the baseline was established.

Similarly, the committed memory used by the heap has declined by 444 MB (and the rest of the output could be inspected to see where else the memory use declined to account for the remaining 4 MB).

This is a very useful technique to examine the footprint of the JVM over time

QUICK SUMMARY

1. Available in Java 8, Native Memory Tracking (NMT) provides details about the native memory usage of the JVM. From an operating system perspective, that includes the JVM heap (which to the OS is just a section of native memory).

2. The summary mode of NMT is sufficient for most analysis, and allows you to determine how much memory the JVM has committed (and what that memory is used for).

C library function - malloc()
(Java appliations make extensive use of malloc)

The C library function **void *malloc(size_t_size)** allocates the requested memory and returns a pointer to it.

Java appliations make extensive use of malloc, either in Java Native Interface(JNI) code that is part of application itself or in the java class libraries, or in binary code that is part of the software development kit (SDK)

Memory-mapped I/O (mmap - map pages of memory)

On modern operating systems, it is possible to *mmap* a file to a region of memory. When this is done, the file can be accessed just like an array in the program.
Note : mmap() is a “C” language function

Java NIO – Memory-Mapped Files with MappedByteBuffer

Memory-mapped I/O uses the filesystem to establish a virtual memory mapping from user space directly to the applicable filesystem pages. With a memory-mapped file, you can pretend that the entire file is in memory and that you can access it by simply treating it as a very large array. This approach greatly simplifies the code you write in order to modify the file.

ReservedCodeCacheSize and CompileThreshold

When we specify too low "-XX:CompileThreshold" and too much bytecode gets compiled by HotSpot and the following warning was seen:

VM warning: CodeCache is full. Compiler has been disabled.

CompileThreshold

By default, CompileThreshold is set to be 10,000:

Very often, we set the threshold lower.
For example : -XX:CompileThreshold=8000

Since the JIT compiler does not have time to compile every single method in an application, all code starts out initially running in the interpreter, and once it becomes hot enough it gets scheduled for compilation.

ReservedCodeCacheSize

A code cache is where JVM uses to store the native code generated for compiled methods. To improve an application's performance, we can set the "reserved" code cache size:

-XX:ReservedCodeCacheSize=256m

Basic JIT Compilation

Java HotSpot VM automatically monitors which methods are being executed. Once a method has become eligible (by meeting some criteria, such as being called often), it is scheduled for compilation into machine code, and it is then known as a hot method.

The compilation into machine code happens on a separate JVM thread and will not interrupt the execution of the program.

In fact, even while the compiler thread is compiling a hot method, the Java Virtual Machine (JVM) will keep on using the original, interpreted version of the method until the compiled version is ready.

In JDK 8,When tiered compilation is enabled, two things happen:

CompileThreshold is ignored

A bigger code cache is needed. Internally, HotSpot will set it to be 240 MB (i.e., 48 MB * 5)

Note:

in JDK8, the below options are not required

-XX:ReservedCodeCacheSize=256m -XX:+TieredCompilation
or
-XX:CompileThreshold=8000

“reserved” code cache is just an address space reservation, it does not really consume any additional physical memory unless it’s used.

AlwaysPreTouch

-XX:+AlwaysPreTouch. This loads all the pages into memory at start-up. That is, there is an upfront cost.

Without the -XX:+AlwaysPreTouch option the JVM max heap is allocated in virtual memory, not physical memory: it is recorded in an internal data structure to avoid it being used by any other process. Not even a single page will be allocated in physical memory until it's actually accessed. When the JVM needs memory, the operating system will allocate pages as needed.

With the -XX:+AlwaysPreTouch option the JVM touches every single byte of the max heap size with a '0', resulting in the memory being allocated in the physical memory in addition to being reserved in the internal data structure (virtual memory).

Large memory pages

Large memory pages

The default memory page size in most operating systems is 4 kilobytes (kb).

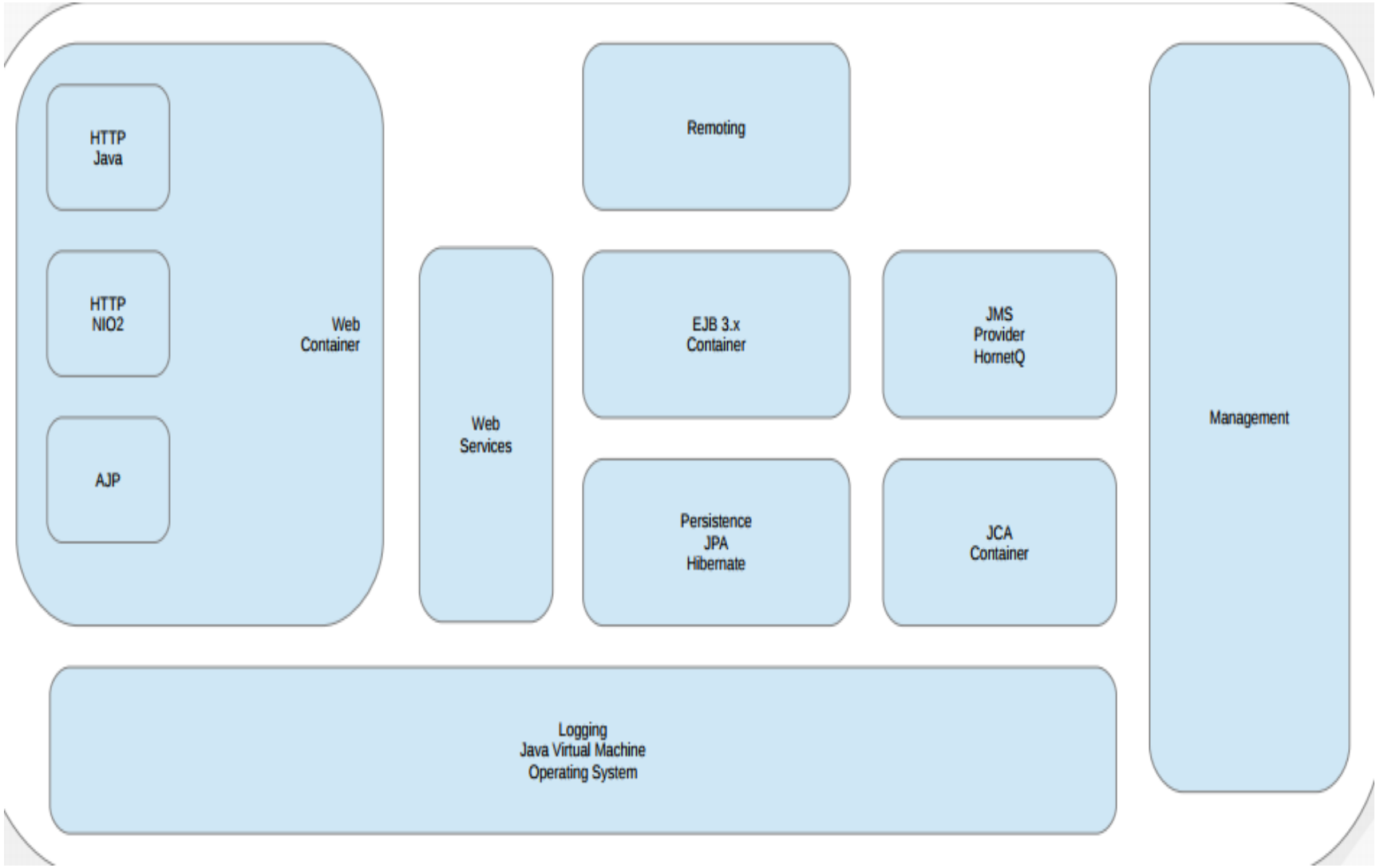
Large memory pages are pages of memory which are significantly larger than 4 kb, usually 2 Mb. In some instances it's configurable, from 2MB to 256MB.

To enable large page memory , use :

-XX:+UseLargePages

Note : Large memory pages are locked in memory, and cannot be swapped to disk like regular memory pages which has both advantages and disadvantages. The advantage is that if the heap is using large page memory it can not be paged or swapped to disk so it's always readily available.

JavaEE Architecture



JBoss Web – Connectors/Thread Pools

Three different use cases determine which web connector to use:

- Low number of connections (i.e. in the hundreds) with high concurrency.

  Use the Java I/O connector, which is synchronous blocking I/O based.

- High number of connections (i.e. in the thousands, tens of thousands or even hundreds of thousands).

  Use the NIO2 based connector (available since EAP 6.1), which is asynchronous non-blocking I/O based.

- Application server fronted by Apache HTTPD server.

  Use AJP connector.

- For all connectors use a defined executor, and never use the default thread pool.
- Set the max connections on your connector configuration.
- Set the protocol parameter to enable the NIO2 based connector.
- Set up multiple connectors if your deployment or deployments have multiple http based points of connection.

  For example, an application or applications deployed that have a web front end, but also a web service based, whether JAX-WS and/or JAX-RS, API.

  Having multiple connectors defined will allow you to configure specific thread pools for each connector, and fine tune the resources consumed.

JBoss Web – Connector/Thread Pool Example

```
<subsystem xmlns="urn:jboss:domain:threads:1.1">

 <unbounded-queue-thread-pool name="JBossWeb">
 <max-threads count="62"/>
 <keepalive-time time="75" unit="minutes"/>
 </unbounded-queue-thread-pool>
 <unbounded-queue-thread-pool name="JBossWebWs">
 <max-threads count="38"/>
 <keepalive-time time="75" unit="minutes"/>
 </unbounded-queue-thread-pool>

</subsystem>
```

```
<subsystem xmlns="urn:jboss:domain:web:2.2" default-virtual-server="default-host"
native="false">
 <configuration>
 <jsp-configuration trim-spaces="true" smap="false" generate-strings-as-char-
arrays="true"/>
 </configuration>

 <connector name="http" protocol="org.apache.coyote.http11.Http11NioProtocol"
scheme="http" socketbinding="http“ executor="JBossWeb"
max-c0nnections="56096"/>

<connector name="httpWs" protocol="org.apache.coyote.http11.Http11NioProtocol"
scheme="http" socketbinding="httpWs“ executor="JBossWebWs"
max-connections="10240"/>

<connector name="ajp" protocol="AJP/1.3" scheme="http" socket-binding="ajp"/>
 <virtual-server name="default-host" enable-welcome-root="true">
 <alias name="benchserver2"/>
 <alias name="benchserver2G1"/>
 </virtual-server>
</subsystem>
```

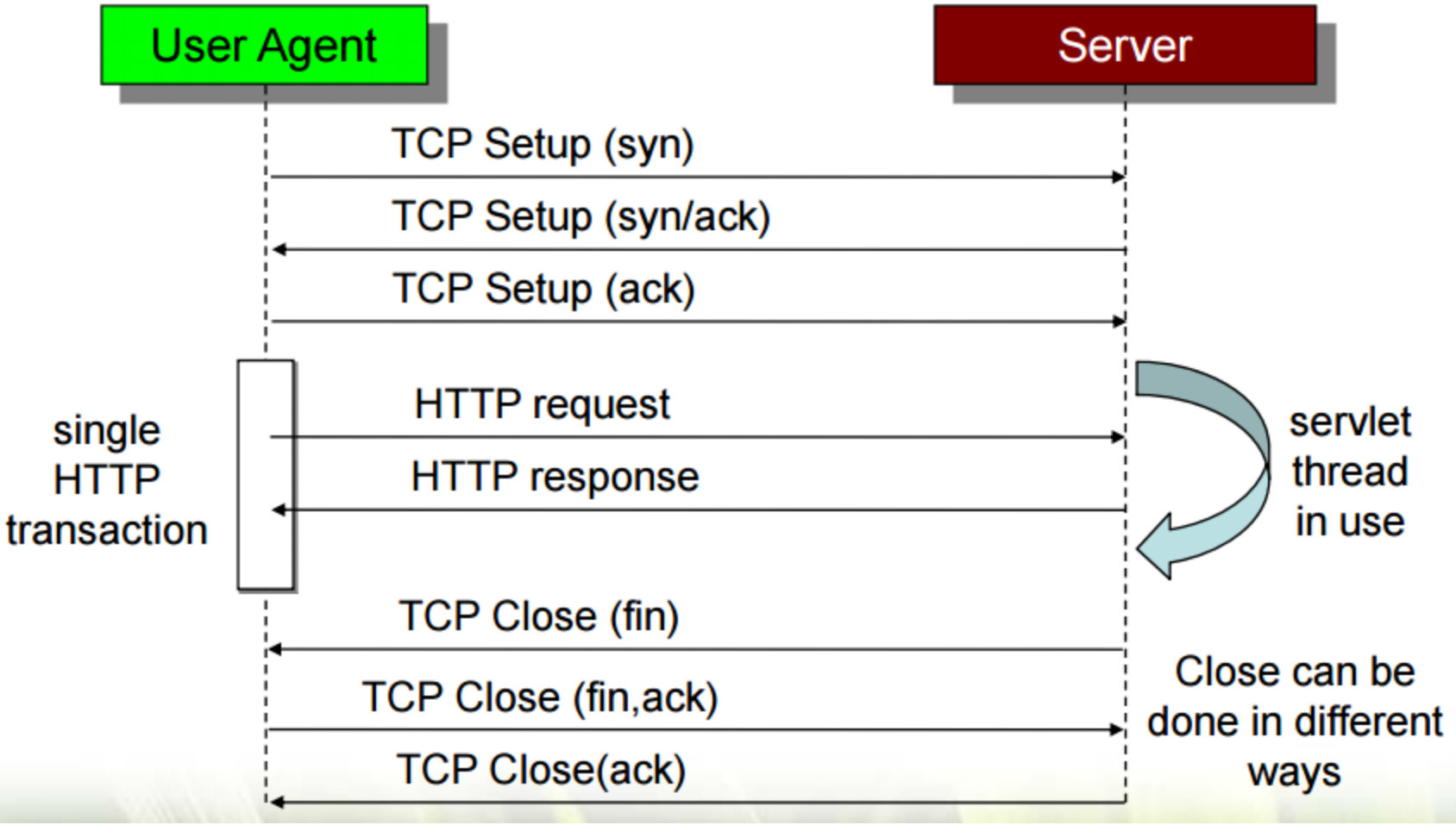
connectionTimeout ＆ keepAliveTimeout

connectionTimeout defines how long Server will wait for a client request after client connected.

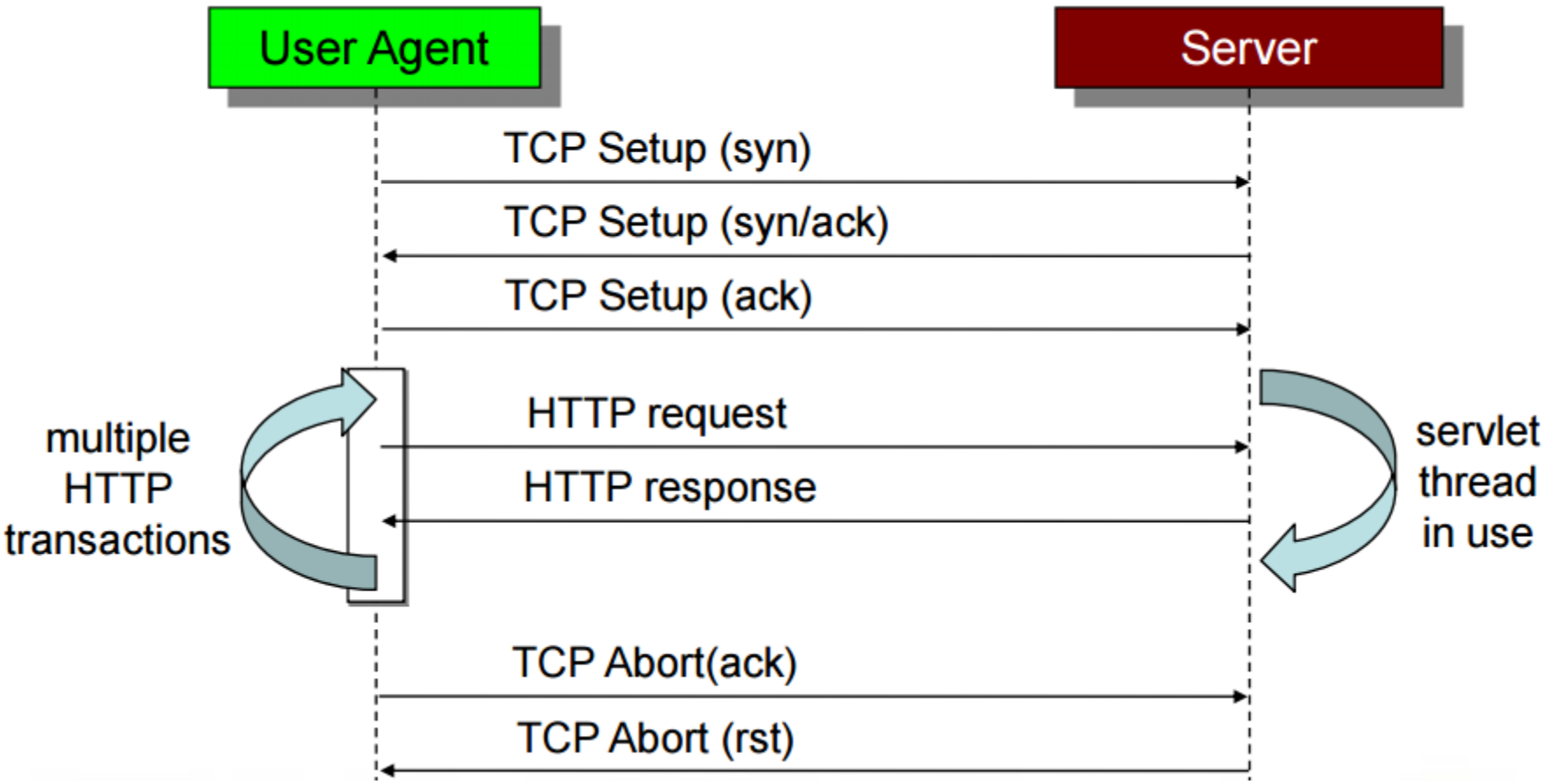
keepAliveTimeout is how long Server will wait for another request before closing the connection

In other words, connectionTimeout is how long Server will wait for the first HTTP request to be sent after TCP connection was established, keepAliveTimeout is how long it will wait for another request to be sent over the same connection

TCP -> No Keep -Alive



TCP -> Keep -Alive



System property values via standalone.conf file for JBoss

org.apache.coyote.http11.DEFAULT_CONNECTION_TIMEOUT Default socket timeout. The default value is 60000 ms.

org.apache.coyote.http11.DEFAULT_KEEP_ALIVE_TIMEOUT Default socket timeout for keep alive. The default value is -1 ms, which means it will use the default socket timeout.

JBoss Web – JSP

- If we use JSP, we have found the following JSP configuration parameters to optimize memory allocation, network usage and throughput.

trim-spaces="true" – tells the container to remove useless spaces from the response.

smap="false" - tells the container to no longer generate JSR 045 SMAP files (SMAP files enables better stack traces for debugging).

development="false" (to ignore run-time changes to jsp files)

Note : This can be useful to have on during development, but shouldn't be necessary in production environments, unless we are having trouble debugging something in production.

generate-strings-as-char-arrays="true" – tells the container to use char arrays when it generates strings.

For scriptlets, set a property that will turn on our optimizations, and it will improve the memory allocation for this use case too.
org.apache.jasper.compiler.Parser.OPTIMIZE_SCRIPTLETS=”true”

<system-properties>

<property name=”org.apache.jasper.compiler.Parser.OPTIMIZE_SCRIPTLETS” value=”true”/>

...

</system-properties>

```
<subsystem xmlns="urn:jboss:domain:web:2.2" default-
virtual-server="default-host“ native="false">


 <configuration>
 <jsp-configuration trim-spaces="true" smap="false"
generate-strings-aschar-arrays="true"/>
 </configuration>
```

Web Services

Web service calls, specifically JAX-WS calls, will be handled by the web container thread pool.

Define a specific connector for your web service calls.

> This will allow you to tune this in isolation of other web requests (e.g. JSP, JSF, etc.).

For @OneWay annotated web services there is an internal thread pool, but better to use the calling thread for this instead of configuring another thread pool.

> By using the calling thread you eliminate the hand-off from one thread pool to another.
> We can also eliminate the context switch by using another thread instead of the calling thread.
>
> (We can enable this behavior through a CXF property called USE_ORIGINAL_THREAD)

For all other web services, message persistence is important, as any stream larger than 64k will be written to disk.

We can configure this through a property org.apache.cxf.io.CachedOutputStream.Threshold

So, if we have a large stream, larger than 64k, you can prevent this from being cached in a file, and keep it in memory.

Of course the trade-off is memory used for this, and we may not want something that large to stay in memory. If you have plenty of memory though, it will be faster.

CXF bus strategies for servicing clients that should be considered.

In EAP, the TCCL_BUS (Thread Context Class Loader) bounds the number of buses instantiated to the number of applications deployed that have web services, limiting the memory footprint.

- This has proved to be the best strategy for performance, especially latency, of JAX-WS requests

```
<subsystem xmlns="urn:jboss:domain:webservices:1.2">
 ...
 <endpoint-config name="Standard-Endpoint-Config">

 <property
name="org.apache.cxf.interceptor.OneWayProcessorInt
erceptor.USE_ORIGINAL_THREAD"
value="true"/>

 </endpoint-config>
```

<system-properties>
 <!-- CXF message handling –>
 <property name=”org.apache.cxf.io.CachedOutputStream.Threshold” value=”4096000"/>

 <property name=”org.jboss.ws.cxf.jaxws-client.bus.strategy” value=”TCCL_BUS”/>

</system-properties>

JCA

The JCA container is responsible for the integration of our data sources into the application server and provides services for them.

Three areas we can concentrate:

Database connection pooling,
the Cached Connection Manager, and
Prepared Statement Caching.

The size of the database connection pool is directly related to the concurrent execution of queries across the application.

- To small a pool, add overhead to response times.

- The default timeout for a database connection is 30 seconds. This is a long time to wait

```
<pool>
 <min-pool-size>200</min-pool-size>
 <max-pool-size>250</max-pool-size>
 <prefill>true</prefill>
</pool>
...
<statement>
 <prepared-statement-cache-size>100</prepared-
statement-cache-size>
 <share-prepared-statements>true</share-prepared-
statements>
</statement>
```

connection used with transaction??
complex joins??

JMS

Key configuration parameters:
- journal-type
- ASYNCIO, NIO

The ASYNCIO option specifies using native ASYNC I/O capabilities, plus opens the file using DIRECT I/O, which bypasses the file system buffer cache.

- The NIO option uses the JDK's NIO API's to write to the journal.

- journal-directory

- The placement of the journal files is important, as the default will be relative to the install of the application server, and that may not be the best performing file system to place your persistent messages on.

- Pooled connection factory:

- transaction mode
- Whether to use XA transactions or local transactions.
- min-pool-size, max-pool-size
- The session pool size.
- The sizing of this depends on the number of

concurrent MDB's your application may be executing, and relates to the maxSession on those MDB's, or if you are using the JMS api directly, the number of concurrent messages being processed

Database low-connection & Resource Contention

Problem :

The below thread is unable to get database connection:

"Thread-1" prio=5 tid=0x00a861b8 nid=0xbdc in **Object.wait()**
[0x02d0f000..0x02d0fb68]
at java.lang.Object.wait(Native Method)
- waiting on <0x22aadfc0> (a
org.tw.testyard.thread.ConnectionPool)
at java.lang.Object.wait(Unknown Source)
at
org.tw.testyard.thread.ConnectionPool.getConnection(ConnectionPool.
java:39)
- locked <0x22aadfc0> (a org.tw.testyard.thread.ConnectionPool)

Solution :

We see such things happening often in your application it is time to revisit the connection pool configuration.

In any case the connection pool size should be atleast as big as the number of worker threads in your application.

We may see the similar issues when dealing with resources such as db connections, network connections or file handles.

Resource Contention

Resource contention typically happens when the threads are fighting for the same resources. If a majority of threads in the application are fighting for the same resource this could speak about poor application design.

The most common symptoms of resource contention issues are :

> The throughput / responsiveness of the application has fallen considerably and the application is too slow
> The CPU utilization is low
> All the application resources (db connection pools etc) seem to free and available
> In thread dumps most the thread would appear in the BLOCKED / WAITING FOR MONITOR ENTRY state

Problem 1:

The below thread is unable to get database connection:

"Thread-1" prio=5 tid=0x00a861b8 nid=0xbdc in **Object.wait()**
[0x02d0f000..0x02d0fb68]
at java.lang.Object.wait(Native Method)
- waiting on <0x22aadfc0> (a org.tw.testyard.thread.ConnectionPool)
at java.lang.Object.wait(Unknown Source)
at org.tw.testyard.thread.ConnectionPool.getConnection(ConnectionPool.java:39)
- locked <0x22aadfc0> (a org.tw.testyard.thread.ConnectionPool)

Solution :

We see such things happening often in your application it is time to revisit the connection pool configuration.

In any case the connection pool size should be atleast as big as the number of worker threads in your application.

We may see the similar issues when dealing with resources such as db connections, network connections or file handles.

Problem 2 :

all the threads are waiting to acquire lock on the monitor <0x44008de0>

"**tcpConnection-9011-8009**" daemon prio=1 tid=0x31446a80 nid=0x49f5 **waiting for monitor entry** [0x2edf5000..0x2edf6e30]
at com.opensymphony.oscache.base.algorithm.AbstractConcurrentReadCache.put(AbstractConcurrentReadCache.java:1648)
waiting to lock <0x44008de0>

This lock is obtained by the below thread:

" **tcpConnection-9011-7817**" daemon prio=1 tid=0x24fee130 nid=0x491c **runnable** [0x1513e000..0x15140130]
at java.lang.System.identityHashCode(Native Method)
locked < 0x44008de0 >

Analysis :

operating system had run out of file descriptors and hence thread tcpConnection-9011-7817 couldnt really persist the cached object to the disk and release the monitor.

Solutions :

Increasing the limit on the number of file descriptors (ulimit) and restarting the application.

Which GC???

1. Do I really need a Low Latency

   No? Use Parallel Gc

2. Do I really need a Big Heap?

   Yes? Thing again!

3. Do I really need a Big Heap?

   No? Use small heap & Parallel GC

   Yes? Try CMS 1st

4. Is CMS performing well?

   Yes? Done!

   No? Tune it!

5. Is tuned CMS Performing well?

   Yes? Done!

   No? Tune it more!

6. Is CMS performing well now?

   Yes? Done!

   No? Do you really need such big heap?

7. Yes, I really need a Big Heap a Low Pauses!

   This is the moment where you should start considering to use G1!

Do you have answer for these options?

-XX:MaxGCPauseMillis

-XX:ParallelGCThreads

-XX:ConcGCThreads

-XX:InitiatingHeapOccupancyPercent

-XX:MaxTenuringThreshold

-XX:+DisableExplicitGC

code analysis using JIT

Compilation Thresholds

If the method we are interested in hasn’t been compiled yet, JITWatch will notify the same.

We can generate some more load for our application or lower the thresholds for JIT so it will treat the method as hot enough to compile.

-XX:+CompileThreshold=*N* flag.

The default value of N for the client compiler is 1,500; for the server compiler it is 10,000.

Reading the compiler's mind

The -XX:+LogCompilation flag produces a low-level XML file about compiler and runtime decisions

-XX:+UnlockDiagnosticVMOptions -XX:+LogCompilation -XX:+PrintInlining -XX:+PrintCompilation

Print Assembly:

-XX:+UnlockDiagnosticVMOptions -XX:+PrintAssembly

Exception:

“Could not load hsdis-amd64.dll; library not loadable; PrintAssembly is disabled“.

Solution:

Download hsdis-1.1.1-win32-amd64.zip file from http://fcml-lib.com/download.html
and copy hsdis-amd64.dll file in jre/jdk bin folder.

Ref : x86_64 Assembly to understand generate code

Common flags

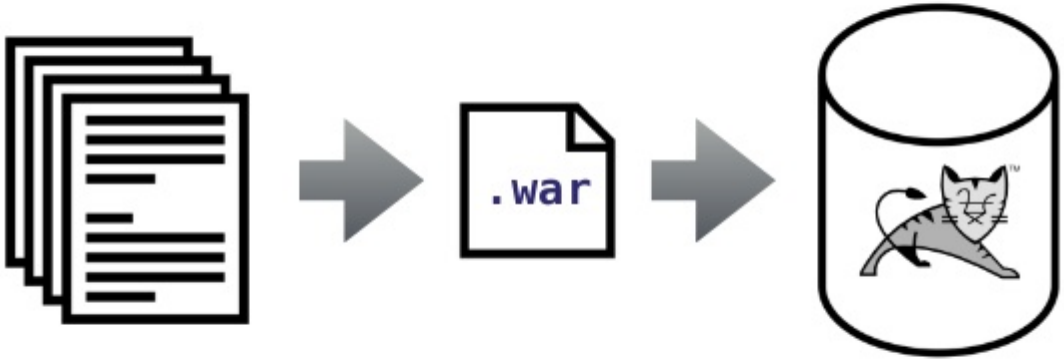
The method attribute flags are common in all these variants:

Symbol	Meaning	Description
%	On stack replacement	*compile type*: * With flag: This CompileTask is an OSR compilation task * Without: This CompileTask is a standard compilation task
s	Synchronized method	* With flag: The method to compile is declared as synchronized * Without: The method to compile is not declared as synchronized
!	Method has exception handlers	* With flag: The method to compile has one or more exception handlers * Without: The method to compile doesn't have any exception handlers

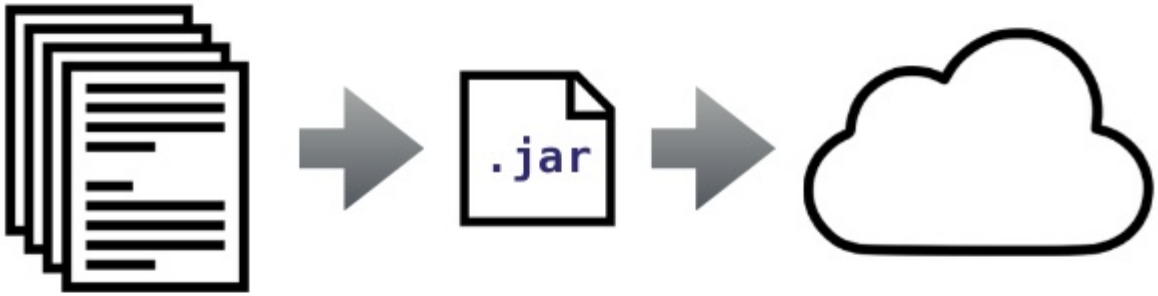
b	Blocking compilation	* With flag: Application thread is blocked by this CompileTask * Without: This CompileTask is executed by a background compiler thread
n	Native wrapper	* With flag: The method to compile is native (not actually compiling the method, but instead generating a native wrapper) * Without: The method to compile is a Java method

JVM Deployment Options

Traditional JVM Deployment



Modern JVM Deployment



Microservices????

Other important areas to focus:

1. Apache MPM modules
2. Caching the responses in webserver
3. Session Replication
4. Dynamic load-balancing algorithms
5. http to https redirection
6. Database locks
7. ORM L1 & L2 Cache
8. HA-Services
9. Sync vs Async communications
10. local vs remote access connectors
11. OS fine-tuning