Monitor Java at System Level

Created by MX Documentation, last modified by EL-HELOU Charlie on 2023 Aug 01

1. Introduction

Everything you need to know on Java Troubleshooting can be found under Java official documentation page on troubleshooting: https://docs.oracle.com/javase/8/docs/technotes/guides/troubleshoot/.

This document contains a summary of the java troubleshooting and describe how to analyze problems occurring on java services using system tools provided by both the Platform and the java package.

To identify if a service is java or native, check the definition file of the service under murex. If the <DefaultBinary> tag is not present in the declaration of the service, then it is a java service.

To locate the service definition, you can check the path from the customerrights.xml; each CODE has a <LauncherDeclaration> tag that indicates the path of the service definition.

2. Troubleshoot Procedure

Different tools can be used to track down java problems, yet note that some of them are intrusive such as the profiles that requires reconfiguration settings, and some are non intrusive such as generating a dump of the heap (freezes the process).

2.1. Generic procedure

- Check the existence of both native and java core files in the application directory.
- Use the kill -3 or kill -SIGQUIT on the java process hanging. The `jstack -l` feature can also be used to generate the thread dump.
- Activate the `-verbose:gc` flag along with the following arguments: `-XX:+PrintGCDetails -XX:+PrintGCTimeStamps`. Use `jstat` tool at runtime to detect statistics on GC. you can analyse the GC with gcparser (sometimes you can use online tools such as http://gceasy.io/)
- Launch the pfiles command on the hanging java process
- use `prstat -s time -Lp` to detect which thread is running (solaris)
- use top -H -p NPID to detect high CPU threads (Linux)
- Launch the following command on hanging java process: {{truss -p -wall -rall <pid>}}
- Monitor the machine status using the `mxloadmon.sh` script. For Aix, request the result of the `nmon` feature, option `-m`.
- Monitor the memory consumption of the process using the `mxpmemloop.sh`
- Request the logs directory
- search the document for the error faced. {{truss -f launchmxj.app -fs 2>&1 | grep open}}
- For CPU related problems, use the following command to identify the most consuming threads `prstat -Lp PID` (linux command: `ps -C java -L -o pid,tid,pcpu,time,size,rss | sort -rk4 | grep <pid>`)
- if problem on CPU due to full memory and the process is not executing a full GC, use `jmap -histo:live` or `jcmd NPID GC.run` to trigger a full GC (use java 64 bits command for 64 bits setups)

The procedure above is generic and will provide enough information to analyze the source of the problem. Check the section below to find which tools can provide additional information for additional analysis of the problem

2.2. Troubleshooting by type

Find in the section below additional information that can help debugging java related problems. Some tools availability depend on version (ex: jcmd for java 1.7)

2.2.1. Hung, Deadlocked, or Looping Process

- Print thread stack for all Java threads:
  - `Ctrl+Pause` (client side)
  - `kill -SIGQUIT pid`
  - `jstack pid` (or `jstack -F pid` if jstack pid does not respond)
  - `jcmd <process id/main class> Thread.print`
- Detect deadlocks
  - Print lock information for a process: `jstack -l pid`
  - Print information on deadlocked threads: Control-PAUSE/kill -3
  - Print list of concurrent locks owned by each thread. Set `-XX:+PrintConcurrentLocks` then `Control-Pause`
  - Request deadlock detection: `JConsole` tool, *Threads* tab
- Get a heap histogram for a process
  - Start Java process with `-XX:+PrintClassHistogram`, then {{kill -SIGQUIT}} / `Ctrl+Pause`
  - `jmap -histo pid` (mandatory to use -d64 if java is 64bits else process will hang on linux, with -F option if pid does not respond. use also -histo:live)
  - `jcmd <pid> GC.class_histogram filename=Myheaphistogram`
- Dump Java heap for a process in binary format to file
  - `jcmd <pid> GC.class_histogram filename=Myheaphistogram`
  - `jmap -dump:format=b,file=heap.dump.hprof $PID` (with -F option if pid does not respond)
- Print heap summary for a process
  - Control-Pause/kill -3
  - jmap -heap pid (mandatory to use -d64 if java is 64bits else process will hang on linux. in case it hangs, use `kill -s SIGCONT <PID>`)
  - starting java 9 and above we use java_path/jhsdb jmap --pid pid --heap
- Print finalization information for a process
  - jmap -finalizerinfo pid
- For CPU related problems, use the following command to identify the most consuming threads
  - prstat -Lp PID
- Attach the command-line debugger to a process
  - {{jdb -connect sun.jvm.hotspot.jdi.SAPIDAttachingConnector:pid=pid}}

2.2.2. Post-mortem Diagnostics, Memory Leaks

- Examine the fatal error log file. Default file name is `hs_err_<pid>.log` in the working-directory.
- Create a heap dump:
  - `jmap -dump:format=b,file=heap.dump.hprof $PID`
  - JConsole tool, MBeans tab
  - Start VM with `-XX:+HeapDumpOnOutOfMemoryError`; if OutOfMemoryError is thrown, VM generates a heap dump.
  - `-XX:+HeapDump` option can be used to observe memory allocation in a running Java application by taking snapshots of the heap over time.
  - `-XX:HeapDumpPath` can be used to specify the path or filename for the heap dump.
  - `-XX:OnError` can be used to specify a script or command to launch when a failure on the process occurs.
    - `-XX:+ShowMessageBoxOnError` command-line option. When a fatal error is encountered, the process prints a message to standard error and waits for a yes or no response from standard input.
    - If the previous flag cannot be used, use the truss utility to suspend a process when the exit system call is excuted. You can afterwards attach the debugger to process: {{truss -t !all -s !all -T exit -p <pid>}}
    - Check with the dump with MemoryAnalyzer
- Browse Java heap dump
  - `jhat heap-dump-file`
- Dump Java heap from core file in binary format to a file
  - `jmap -dump:format=b,file=filename corefile`
- Get a heap histogram for a process
  - Start Java process with `-XX:+PrintClassHistogram`, then Control-Pause
  - `jmap -histo pid` (with -F option if pid does not respond)
- Get a heap histogram from a core file

    - `jmap -histo corefile`
- Print shared object mappings from a core file

- Print finalization information from a core file
    - `jmap -finalizerinfo corefile`
- Print Java configuration information from a core file
    - `jinfo corefile`
- Print thread trace from a core file
    - `jstack corefile`
- Print lock information from a core file
    - `jstack -l corefile`
- Usage of the verbose parameter
    - `-verbose:gc`
    - `-verbose:class`
    - `-verbose:jni`
- Attach the command-line debugger to a core file on the same machine
    - jdb -connect sun.jvm.hotspot.jdi.SACoreAttachingConnector:javaExecutable=path,core=corefile
- Attach the command-line debugger to a core file on a different machine
    - On the machine with the core file: `jsadebugd path corefile` and on the machine with the debugger: {{jdb -connect sun.jvm.hotspot.jdi.SADebugServerAttachingConnector:debugServerName=machine}}
- `libumem` library and `LD_PRELOAD` can be used to debug memory leaks.

2.2.3. Monitoring

- Monitor memory
    - Control-Pause/kill -3 prints generation information.
    - Heap allocation profiles via HPROF: `java -agentlib:hprof=heap=sites`
    - JConsole tool, Memory tab
- Monitor CPU usage, thread and class activity
    - JConsole tool, Overview, VM Summary, Threads and Classes tabs
- Print statistics on the class loader
    - `jstat -class vmID`
- Print statistics on the compiler
    - Compiler behavior: `jstat -compiler vmID`
    - Compilation method statistics: `jstat -printcompilation vmID`
- Print statistics on garbage collection
    - Summary of statistics: `jstat -gcutil vmID`
    - Summary of statistics, with causes: `jstat -gccause vmID`
    - Behavior of the gc heap: `jstat -gc vmID`
    - Capacities of all the generations: `jstat -gccapacity vmID`
    - Behavior of the new generation: `jstat -gcnew vmID`
    - Capacity of the new generation: `jstat -gcnewcapacity vmID`
    - Behavior of the old and permanent generations: `jstat -gcold vmID`
    - Capacity of the old generation: `jstat -gcoldcapacity vmID`
    - Capacity of the permanent generation: `jstat -gcpermcapacity vmID`
- Monitor objects awaiting finalization
    - JConsole tool, VM Summary tab
    - jmap -finalizerinfo pid

2.2.4. Actions on a Remote Debug Server (NOT TESTED)

First, attach the debug daemon `jsadebugd`, then execute the commands below

- Dump Java heap in binary format to a file: `jmap -dump:format=b,file=filename hostID`
- Print shared object mappings: `jmap hostID`
- Print heap summary : `jmap -heap hostID`
- Print finalization information : `jmap -finalizerinfo hostID`
- Print lock information : `jstack -l hostID`
- Print thread trace : `jstack hostID`
- Print Java configuration information: `jinfo hostID`

3. Description of JVM

The Java Virtual Machine (JVM) has the following types of memory: heap, non-heap, and native.

3.1. Heap memory

Heap memory is the runtime data area from which memory for all class instances and arrays are allocated.

Common Out of memory error: java.lang.OutOfMemoryError: Java heap space

Causes:

- Low RAM memory
- Small heap
- Memory leak

Solution:

- More physical memory on the machine.
- If Xmx value is less than 1Gb, double the Xmx directly as a solution. Note that the minimum Xmx for any launcher on production is 256M
- If Xmx is higher than 1 Gb, check if the process is 32 bits or 64 bits. If 32 bits, do not increase the Xmx more than 2.5G (1.5G on Aix) fo

VM needs to allocate all the heap in a single continuous chunk, which may not be possible because of the memory fragmentation.

- Add the following to the launcher's arguments *-jopt:-HeapDumpOnOutOfMemoryError -jopt:-HeapDumpPath="javaheap dump.hprof"*

Fix memory leaks

3.2. Non-heap memory type.

The non-heap memory includes the method area and memory required for the internal processing or optimization for the Java virtual machine. It stores per-class structures such as a runtime constant pool, field and method data, and the code for methods and constructors. The permanent generation is a non-heap memory area that holds all the reflective data of the VM itself such as class and method objects.

Common Out of memory error: {{java.lang.OutOfMemoryError: PermGen space}}

Causes:

- Low memory for permanent generation area
- Classes on permanent generation are prevented to be collected
- Classloader leak

Solution:

- Make permanent generation area larger
- Disable hot deployment or class reloading for classloader leak

3.3. Native memory

Native memory is the virtual memory managed by the operating system. The Java Native Interface (JNI) code or the native library of an application and the JVM implementation allocate memory from the native heap

Common Out of memory error: {{java.lang.OutOfMemoryError: request <size> bytes for <reason>. Out of swap space?}}

Causes:

- Big stack size (-Xss, set the maximum native stack size for any thread)
- Small memory available
- Poor thread management

Solution:

- Reserve more memory to operating system. check if there is sufficient swap/ram space
- Make stack size smaller
- Lower heap size
- Better thread management
- Change the MAXDATA parameter

`java.lang.OutOfMemoryError: unable to create new native thread`

causes: The OS refuse native memory allocation to the java process. might be due to:

- OS has no more memory left
- Process has reached its limitation
- OS has reached "max user process"

Solution:

- check that there is enough free memory
- check that he process has not reached its 4Gb limitation (32bit process).
- Reduce Xmx/Xss to give more space to the native threads
- check the OS max user process limitation using `ulimit -a` and increase it

use the jcmd tool to troubleshoot all Native memory (NMT) issues

3.4. Code Cache

The Java Virtual Machine (JVM) generates native code and stores it in a memory area called the **codecache**.

JVM Code Cache is an area where JVM stores its byte-code compiled into native code. We call each block of the executable native code a nmethod. The nmethod might be a complete or inlined Java method.

The just-in-time (JIT) compiler is the biggest consumer of the code cache area as part of improving the performance of the running process. This is where the JIT compiled methods are kept.

For more details about the CodeCache, please refer to this wiki page: JAVA - Code Cache

3.5. additional info

HotSpot VM also uses a C/C++ heap for storage of HotSpot VM internal objects and data called Arena. This base class and its subclasses provide a rapid C/C++ allocation layer that sits on top of the C/C++ malloc/free memory management routines.
And allocates memory blocks from 3 global ChunkPools. (1k , 10k ...)

minor GC: card table is an array with one byte entry per card (chunk of 512b) in the heap. Every update done on the chunk updates the byte of the card table. During minor GC, only areas with dirty cards are scanned.
end of the minor GC, the two survivor spaces swap roles. The eden is entirely empty; only one survivor space is in use; and the occupancy of the old generation has grown slightly.

`Eden space`: The memory pool where almost all Java objects are allocated.
`Survivor space`: The memory pool containing objects that have survived at least one garbage collection of the eden space.
`Old or tenured space`: The memory pool containing objects that have survived some garbage collection age threshold.

`Permanent generation space`: The memory pool containing all the reflective data of the JVM such as class and method objects. If the monitored JVM supports class data sharing, this space will be divided into read-only and readwrite areas.

```
[PSYoungGen: 99952K->14688K(109312K)]
422212K->341136K(764672K), 0.0631991 secs] [Times: user=0.83 sys=0.00, real=0.06 secs]

GC label indicates this is minor garbage collection
99952K, is the occupancy of the young generation space prior to the garbage collection. The value to the right of the ->, 14688K, is the occup
The value inside the parentheses, (109312K), is the size, not the occupancy, of the young generation space

the ->, 422212K, is the occupancy of the Java heap before the garbage collection. The value to the right of the ->, 341136K, is the occupancy
The value inside the parentheses, (764672K), is the total size of the Java heap.
```

Unused classes are unloaded from the permanent generation space when additional space is required as a result of other classes needing to be loaded. To unload classes from permanent generation, a full garbage collection is required

```
[Full GC[Unloading class sun.reflect.GeneratedConstructorAccessor3]
[Unloading class sun.reflect.GeneratedConstructorAccessor8]
[Unloading class sun.reflect.GeneratedConstructorAccessor11]
[Unloading class sun.reflect.GeneratedConstructorAccessor6]
8566K->5871K(193856K), 0.0989123
```

3.5.1. MAXDATA

The Mx binary was linked with Maxdata = 0xD (3.5 Gb Max Memory) before DEF0080096. Now it is linked with maxdata = 0XA (2.75 GB Max memory).

Note that all maxdata values above 0xA disable the shared memory each binary launched with a MaxData > 0xA will have its own copy of the executable / libraries.

You can find below the important commands / infos regarding AIX with Maxdata & Java.

3.5.1.1. General Knowledge

If you are new to AIX platform, and the first version of Java you have used on AIX is Java 1.4, you may be wondering what I am talking about.
Java 1.4 is more intelligent than its predecessors; it no longer requires you to do calculations and set environment variables. Based on the requested heap size, Java 1.4 launcher sets the appropriate o_maxdata value automatically. The exact behavior of Java 1.4 is a bit different from the one I explained above, and it may change in future releases. You can use svmon to find out for yourself how Java 1.4 and above manipulate o_maxdata.

Note, though, that you can still use the techniques you learnt in this article with Java 1.4. If any `LDR_CNTRL=MAXDATA` values are set in the environment, Java 1.4 will not override it, even if it results in an error during startup. Conversely, if you do want Java 1.4 to manage the heap sizing for you, make sure you remove any `LDR_CNTRL=MAXDATA` entries from the environment settings.

3.5.1.2. Useful Commands

- `ldedit -b maxdata:0xA0000000/dsa mx` Forces a 0xA maxdata to the mx binary (Will be overrided by environment variables)
- `dump -o mx` Checks the maxdata value written into the mx binary

Additional information may be found under the following link:
http://www.ibm.com/developerworks/eserver/articles/aix4java1.html

4. Error messages

4.1. Types of OOME

The `lang.OutOfMemoryError` error is thrown when there is insufficient space to allocate an object in the Java heap or in a particular area of the heap.

4.1.1. Reasons

The main reasons behind this error are:

- Memory leak (can occur without having an OOME crash)
- Insufficient initial heap space allocated
- Finalized objects, not processed by the GC
- Not enough resources on the machine (check first)

4.1.2. Description

`Memory Leaks`: unintentional object retention; VM is being allocated but is not being returned when it is no longer needed.

`Finalizers`: protected method for java class. An object that has a finalizer will not be garbage collected until its finalizer is run

`Deadlocks`: occurs when two or more threads are each waiting for another to release a lock. Causes the application or part of the application to become unresponsive

`Looping Threads`: one or more threads are executing in an infinite loop. Can cause the application to hand and may consume all available resources. To tourbleshoot looping, check points below:

- Focus initially on the threads that are in the RUNNABLE state.
- Check if there are deadlocks
- If no deadlock, check that the running process are blocked with Object wait

4.2. Other errors

`Requested array size exceeds VM limit`

Occurs when the initial heap is low or a bug in application is trying to generate a big array.

Solutions

- Lower the array size
- Make heap larger

`GC overhead limit exceeded`

Occurs when too much time spent on GC (98%) and little heap (2%) to be free.

Solutions

- Similar to java heap space
- Tune GC behavior

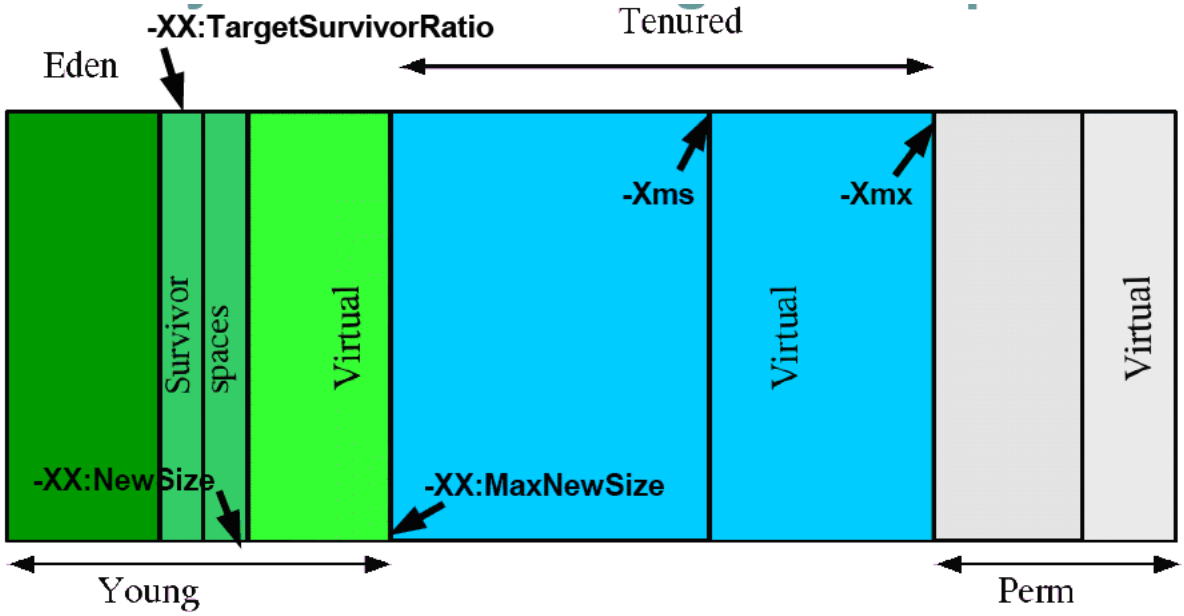
4.3. Crash Instead of OutOfMemoryError

Sometimes an application crashes soon after an allocation from the native heap fails. This occurs with native code that does not check for errors returned by memory allocation functions. For example, the malloc system call returns NULL if there is no memory available. If the return from malloc is not checked, then the application might crash when it attempts to access an invalid memory location

5. Memory management

5.1. Memory pools and configuration Options

The following picture illustrates the different areas of the memory allocated to a java process.



Initial new Size: -XX:NewSize=<total heap size/[3 to 5]>m
Maximum New Size: -XX:MaxNewSize=<total heap size/[3 to 5]>m
Initial Heap Size: -Xms<total heap size>m
Maximum Heap Size: -Xmx<total heap size>m

The Perm space is not calculated in the Xmx of the java process and allocates memory only when it is saturated.

5.2. Memory Sizing

The following table provides the Java recommendation for the sizing:

Table 7-3. Guidelines for Calculating Java Heap Sizing

Java heap	`-Xms and -Xmx`	3x to 4x old generation space occupancy after full garbage collection
Permanent Generation	`-XX:PermSize` `-XX:MaxPermSize`	1.2x to 1.5x permanent generation space occupancy after full garbage collection
Young Generation	`-Xmn`	1x to 1.5x old generation space occupancy after full garbage collection
Old Generation	Implied from overall Java heap size minus the young generation size	2x to 3x old generation space occupancy after full garbage collection

Guidelines to follow when calculating Java heap sizing are summarized in Table 7-3.

5.3. Garbage collector

5.3.1. Description

Java uses an automatic garbage collector to manage memory in the object lifecycle. The programmer determines when objects are created, and the Java runtime is responsible for recovering the memory once objects are no longer in use. Once no references to an object remain, the unreachable object becomes eligible to be freed automatically by the garbage collector.

A garbage collector is responsible for

- Allocating memory
- Ensuring that any referenced objects remain in memory
- Recovering memory used by objects that are no longer reachable from references in executing code.

5.3.2. Design choice

A number of choices must be made when selecting a GC due to the trade-offs that exists between time, space, and frequency.

Example, if a heap size is small, collection will be fast but the heap will fill up more quickly, thus requiring more frequent collections. Conversely, a large heap will take longer to fill up and thus collections will be less frequent, but they may take longer

5.3.2.1. Stop-the-world

all Java executing threads are blocked or stopped from executing in Java code while the garbage collector frees up memory as a result of finding Java objects no longer in use by the application. If an application thread is executing in native code (i.e., JNI), it is allowed to continue, but will block if it attempts to cross the native boundary into Java code.

5.3.2.2. Serial versus Parallel (Mx uses serial)

With serial collection, only one thing happens at a time. For example, even when multiple CPUs are available, only one is utilized to perform the collection. When parallel collection is used, the task of garbage collection is split into parts and those sub-parts are executed simultaneously, on different CPUs. The simultaneous operation enables the collection to be done more quickly, at the expense of some additional complexity and potential fragmentation

5.3.2.3. Concurrent versus Stop-the-world

GC executes simultaneously versus GC freezes the application Trade off for the concurrent GC include shorter pause time versus addition of some overhead to concurrent collectors that affects performance and requires a larger heap size

5.3.2.4. Compacting versus Non-compacting:

Compact the memory, moving all the live objects together and completely reclaiming the remaining memory versuss releases the space utilized by garbage objects in-place

5.3.3. Generational Collection

Young generation collections occur relatively frequently and are efficient and fast because the young generation is usually small and likely to contain a lot of objects that are no longer referenced. When young generation fills up, a young generation collection is performed

Objects that survive some number of young generation collections are eventually promoted, or tenured, to the old generation. When old or permanent generation fills up, what is known as a full collection (sometimes referred to as a major collection) is typically done.

Description of the collection of each generation for each type of GC can be found under management_whitepaper.pdf and can be requested from Operate-MX.

5.3.4. Arguments used for GC

Arguments	Description
-XX:+UseSerialGC	Usage of serail GC
-XX:+UseParallelGC	Usage of Parallet GC
-XX:+UseParallelOldGC	Parallel compacting
-XX:+UseConcMarkSweepGC	Concurrent mark-sweep (CMS)

Arguments	Description
-XX:+PrintGCTimeStamps	Outputs a time stamp at the start of each garbage collection event. Used with -XX:+PrintGC or -XX:+PrintGCDetails to show when each garbage collection begins

6. JDK tools

All tools can be found under: *usr/local/java/jdk1.6.0_06/bin*

Main help page can be found under http://java.sun.com/javase/7/docs/technotes/tools/ . For the tools below, check their man page if need arose for additional information.

6.1. JCMD

The jcmd tool can do anything from jstacks, to heap dump, to record profiling on the fly. It will be a replacement for jmap, jinfo, jstack... in the future
https://docs.oracle.com/javase/8/docs/technotes/guides/troubleshoot/tooldescr006.html#BABEJDGE

it can prints all information relative to native threads https://docs.oracle.com/javase/8/docs/technotes/guides/troubleshoot/tooldescr007.html#BABGFCDA

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

-XX:NativeMemoryTracking=off

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

jcmd <pid> VM.native_memory summary scale= KB

Use the following VM diagnostic command line option to obtain last memory usage data at VM exit when Native Memory Tracking is enabled. The level of detail is based on tracking level.
-XX:+UnlockDiagnosticVMOptions -XX:+PrintNMTStatistics

6.2. Jinfo

Prints Java configuration information for a given Java process or core file or a remote debug server

jinfo $PID

```
.
sun.io.unicode.encoding = UnicodeBig
sun.cpu.endian = big
org.apache.commons.logging.Log = murex.shared.log.commons.CommonsLogger
sun.cpu.isalist =

VM Flags:

-XX:MaxPermSize=128M -Xms32M -Xmx512M -Dsun.rmi.dgc.client.gcInterval=3600000 -Dsun.rmi.dgc.server.gcInterval=3600000 -XX:+UseSerialGC -Xbootc
```

```
=jinfo -flag +HeapDumpOnOutMemoryError $PID=

/usr/local/java/jdk1.7.0_51/bin/jinfo -flag ParallelGCThreads 15836
-XX:ParallelGCThreads=8
```

Dynamically set, unset, and change the values of certain JavaVM flags for a specified Java process

To know the list of flags/arguments and the default value that is given to the java process, use the following command, it will print everything

```
/usr/local/java/jdk1.7.0_79/bin/java  -XX:+AggressiveOpts -XX:+UnlockDiagnosticVMOptions -XX:+UnlockExperimentalVMOptions -XX:+PrintFlagsFinal
```

the following can be added on the startup of any java process `-XX:+PrintFlagsFinal`

List of interesting flags can be found under oracle documentation

Some interesting flags:
Error rendering macro 'code': Invalid value specified for parameter 'com.atlassian.confluence.ext.code.render.InvalidValueException'

```
CheckMemoryInitialization       Checks memory initialization
CITime  collect timing information for compilation
GCALotAtAllSafepoints   Enforce ScavengeALot/GCALot at all potential safepoints
GCTimeLimit     Limit of proportion of time spent in GC before an OutOfMemory" error is thrown (used with GCHeapFreeLimit)
GCTimeRatio     Adaptive size policy application time to GC time ratio
OnOutOfMemoryError      Run user-defined commands on first java.lang.OutOfMemoryError
HeapDumpOnOutOfMemoryError      Dump heap to file when java.lang.OutOfMemoryError is thrown
HeapDumpPath    When HeapDumpOnOutOfMemoryError is on, the path (filename or" directory) of the dump file (defaults to java_pid.hprof" in the working di
LogCompilation  Log compilation activity in detail to hotspot.log or LogFile
LogFile If LogVMOutput is on, save VM output to this file [hotspot.log]
LogVMOutput     Save VM output to hotspot.log, or to LogFile
MaxFDLimit      Bump the number of file descriptors to max in solaris.
MemProfiling    Write memory usage profiling to log file
```

```
VerifyAfterGC   Verify memory system after GC
VerifyBeforeExit        Verify system before exiting
VerifyBeforeGC  Verify memory system before GC
```

6.3. Jmap

Allows obtaining a heap histogram and a heap dump (picture if in memory objects) at runtime.

`jmap -heap $PID` is used to display a snapshot of the heap.

Important note: it is mandatory to use -d64 if java is 64bits else process will hang on linux. in case it hangs, use `kill -s SIGCONT <PID>`

```
using thread-local object allocation.
Mark Sweep Compact GC   //Type of garbage collector

Heap Configuration:
   MinHeapFreeRatio = 40
   MaxHeapFreeRatio = 70
   MaxHeapSize      = 536870912 (512.0MB)     //Xmx parameter
   NewSize          = 2228224 (2.125MB)
   MaxNewSize       = 4294901760 (4095.9375MB)
   OldSize          = 4194304 (4.0MB)
   NewRatio         = 2     //Ratio between the new and old generation
   SurvivorRatio    = 8     //Ratio between the new and survivor space
   PermSize         = 16777216 (16.0MB)
   MaxPermSize      = 134217728 (128.0MB)

Heap Usage:
New Generation (Eden + 1 Survivor Space): // Survivor space are the From and To. Eden always added to the From since the To should be empty
   capacity = 10158080 (9.6875MB)
   used     = 4080880 (3.8918304443359375MB)
   free     = 6077200 (5.7956695556640625MB)
   40.173733618951616% used
Eden Space:
   capacity = 9043968 (8.625MB)
   used     = 4080880 (3.8918304443359375MB)
   free     = 4963088 (4.7331695556640625MB)
   45.12267181838768% used
From Space:
   capacity = 1114112 (1.0625MB)
   used     = 0 (0.0MB)
   free     = 1114112 (1.0625MB)
   0.0% used
To Space:
   capacity = 1114112 (1.0625MB)
   used     = 0 (0.0MB)
   free     = 1114112 (1.0625MB)
   0.0% used
tenured generation:     //Old generation. Space increase permanently when value is higher than the free ration
   capacity = 22413312 (21.375MB)
   used     = 1802160 (1.7186737060546875MB)
   free     = 20611152 (19.656326293945312MB)
   8.040578741776315% used
Perm Generation:
   capacity = 16777216 (16.0MB)
   used     = 8956912 (8.541976928710938MB)
   free     = 7820304 (7.4580230712890625MB)
   53.38735580444336% used
```

`jmap -heap -d64 $PID` is used for the mxml launcher or other launchers defined in 64 bit.

`jmap -permstat $PID` is used to display the list of objects in the permGen space. The dead objects are no longer in use and should be removed when the GC is launched.

```
class_loader    classes bytes   parent_loader   alive?  type

<bootstrap>     1249    3031248   null          live    <internal>
0xde7e1e68      1       1456      null          dead    sun/reflect/DelegatingClassLoader@0xf3c46b60
0xde6b28e8      103     333072    null          dead    sun/misc/Launcher$ExtClassLoader@0xf3d26db8
0xde7e1d18      1       1456      null          dead    sun/reflect/DelegatingClassLoader@0xf3c46b60
0xde7e09b8      1       1448    0xde6a03d0      dead    sun/reflect/DelegatingClassLoader@0xf3c46b60
0xde7e1ea0      1       1456    0xde6a03d0      dead    sun/reflect/DelegatingClassLoader@0xf3c46b60
0xde7dfa58      1       1448    0xde6a03d0      dead    sun/reflect/DelegatingClassLoader@0xf3c46b60
0xde7e12f0      1       1456    0xde6a03d0      dead    sun/reflect/DelegatingClassLoader@0xf3c46b60
```

```
0xde78c490      0       0       0xde6a03d0      dead    java/util/ResourceBundle$RBClassLoader@0xf404b7f8
0xde7e0fd8      1       920     0xde6a03d0      dead    sun/reflect/DelegatingClassLoader@0xf3c46b60
0xde7e1fb8      1       1448    0xde6a03d0      dead    sun/reflect/DelegatingClassLoader@0xf3c46b60
0xde6a03d0      756     1629680 0xde6b28e8      dead    sun/misc/Launcher$AppClassLoader@0xf3d68798
0xde7637f8      1       904       null          dead    sun/reflect/DelegatingClassLoader@0xf3c46b60
0xde7dcda0      1       1448    0xde6a03d0      dead    sun/reflect/DelegatingClassLoader@0xf3c46b60
0xde7e1df8      1       1456      null          dead    sun/reflect/DelegatingClassLoader@0xf3c46b60
0xde6a0328      45      97584   0xde6a03d0      dead    murex/rmi/loader/FileServerClassLoader@0xf3ef6a08
0xde7e1ed8      1       1456      null          dead    sun/reflect/DelegatingClassLoader@0xf3c46b60
0xde7df100      1       1456      null          dead    sun/reflect/DelegatingClassLoader@0xf3c46b60
0xde7e1c70      1       1448    0xde6a03d0      dead    sun/reflect/DelegatingClassLoader@0xf3c46b60

total = 39      2187    5138800     N/A         alive=1, dead=38            N/A
```

`jmap -histo:live $PID` prints the histogram of the process with the live objects only (`note that this will trigger a full GC`). Remove the :live to check all the histogram.

```
num     #instances         #bytes  class name
----------------------------------------------
   1:         19979        2239360  <constMethodKlass>
   2:         19979        1601848  <methodKlass>
   3:         31806        1342128  <symbolKlass>
   4:          3873        1043048  [B
   5:         10961        1002272  [C
   6:          1909         984344  <constantPoolKlass>
   7:          1909         762184  <instanceKlassKlass>
   8:          1705         614320  <constantPoolCacheKlass>
   9:          2368         587184  [I
  10:         10675         256200  java.lang.String
  11:          2108         202368  java.lang.Class
  12:          2808         176712  [S
  13:          3063         136968  [[I
  14:           363         104352  <methodDataKlass>
  15:          1907          94800  [Ljava.lang.Object;
  16:          2581          61944  java.util.Hashtable$Entry
  17:           594          57024  java.io.ObjectStreamClass
  18:           695          55600  java.lang.reflect.Method
  19:           170          54400  <objArrayKlassKlass>
  20:           413          47976  [Ljava.util.HashMap$Entry;
  21:          1822          43728  java.util.HashMap$Entry
```

`jmap -finalizerinfo $PID`
Print information on the objects that are waiting to be finalized.

```
Attaching to process ID 19960, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 10.0-b22
Number of objects pending for finalization: 0
```

`jmap -dump: format=b,file=heap.dump.hprof $PID`

Generate core called heap.dump.prof for analysis purpose.

6.4. Jstack

Prints the stack traces of all java threads for a given process or core dump. Each thread consists of a header and a stack trace. A point of entry is to check for threads having BLOCKED status.

`jstack $PID`

```
"Thread-713" daemon prio=3 tid=0x0011c000 nid=0x59c in Object.wait() [0xe59ff000..0xe59ff870]
   java.lang.Thread.State: WAITING (on object monitor)
        at java.lang.Object.wait(Native Method)
        - waiting on <0xe7db97d8> (a murex.apps.middleware.client.shared.thread.worker.Worker)
        at java.lang.Object.wait(Object.java:485)
        at murex.apps.middleware.client.shared.thread.worker.Worker.run(Worker.java:85)
        - locked <0xe7db97d8> (a murex.apps.middleware.client.shared.thread.worker.Worker)

"Thread-712" daemon prio=3 tid=0x00117400 nid=0x59b in Object.wait() [0xe55ff000..0xe55ff8f0]
   java.lang.Thread.State: WAITING (on object monitor)
```

```
        at java.lang.Object.wait(Native Method)
        - waiting on <0xe7db9578> (a murex.apps.middleware.client.shared.thread.worker.Worker)
```

```
        - locked <0xe7db9578> (a murex.apps.middleware.client.shared.thread.worker.Worker)

"Attach Listener" daemon prio=3 tid=0x0011c400 nid=0x39b waiting on condition [0x00000000..0xe5dffe00]
   java.lang.Thread.State: RUNNABLE

"Thread-6" daemon prio=3 tid=0x00579000 nid=0x19 in Object.wait() [0xe5aff000..0xe5aff8f0]
   java.lang.Thread.State: WAITING (on object monitor)
        at java.lang.Object.wait(Native Method)
        - waiting on <0xed26aa78> (a murex.apps.middleware.client.shared.thread.worker.Worker)
        at java.lang.Object.wait(Object.java:485)
        at murex.apps.middleware.client.shared.thread.worker.Worker.run(Worker.java:85)
        - locked <0xed26aa78> (a murex.apps.middleware.client.shared.thread.worker.Worker)
```

`jstack -F -m -l $PID | c++filt`

With the `-m`, the jstack print both java and native frames. As for the `-l` feature, it is used to check for deadlocks situation. Note that these features can be eached used by itself.

```
JVM version is 10.0-b22
Deadlock Detection:
No deadlocks found.
----------------- t@1 -----------------
0xff2c6a18      __lwp_wait + 0x4
0xff2c2348      _thrp_join + 0x34
0xff2c24b4      thr_join + 0x10
0x00018a04      ContinueInNewThread + 0x30
0x00012480      main + 0xeb0
0x000111a0      _start + 0x108
----------------- t@2 -----------------
0xff2c6acc      ___lwp_cond_wait + 0x4
0xfee5b34c      bool Monitor::wait(bool,long,bool) + 0x484
0xfef1bd80      bool Threads::destroy_vm() + 0xac
0xfecad5e0      jni_DestroyJavaVM + 0x198
----------------- t@3 -----------------
0xff2c6acc      ___lwp_cond_wait + 0x4
0xff2b1304      _lwp_cond_timedwait + 0x1c
0xfee5b1cc      bool Monitor::wait(bool,long,bool) + 0x304
0xfef6a854      void VMThread::loop() + 0x1c4
0xfea4fc00      void VMThread::run() + 0x98
0xfee6d0b0      java_start + 0x168
0xff2c57f8      _lwp_start
----------------- t@4 -----------------
0xff2c6acc      ___lwp_cond_wait + 0x4
0xfee746a0      void os::PlatformEvent::park() + 0x100
0xfef06194      void ObjectMonitor::wait(long long,bool,Thread*) + 0x3d4
0xfe9ca914      void ObjectSynchronizer::wait(Handle,long long,Thread*) + 0x180
0xfe9ca524      JVM_MonitorWait + 0x2b8
0xfc00e1e8      * java.lang.Object.wait(long) bci:189679432 (Interpreted frame)
0xfc00e194      * java.lang.Object.wait(long) bci:0 (Interpreted frame)
0xfc005a30      * java.lang.Object.wait() bci:2 line:485 (Interpreted frame)
0xfc005a30      * java.lang.ref.Reference$ReferenceHandler.run() bci:46 line:116 (Interpreted frame)
0xfc00021c      <StubRoutines>
0xfe93344c      void JavaCalls::call_helper(JavaValue*,methodHandle*,JavaCallArguments*,Thread*) + 0x1e4
0xfec7cad8      void JavaCalls::call_virtual(JavaValue*,KlassHandle,symbolHandle,symbolHandle,JavaCallArguments*,Thread*) + 0xec
0xfe9d3ce0      void JavaCalls::call_virtual(JavaValue*,Handle,KlassHandle,symbolHandle,symbolHandle,Thread*) + 0x74
0xfe9e734c      void thread_entry(JavaThread*,Thread*) + 0x110
0xfef175dc      void JavaThread::thread_main_inner() + 0x48
0xfee6d0b0      java_start + 0x168
0xff2c57f8      _lwp_start
 Locked ownable synchronizers:
    -<0xf0bc2928>, (a java/util/concurrent/locks/ReentrantLock$NonfairSync)
----------------- t@5 -----------------
0xff2c6acc      ___lwp_cond_wait + 0x4
0xfee746a0      void os::PlatformEvent::park() + 0x100
0xfef06194      void ObjectMonitor::wait(long long,bool,Thread*) + 0x3d4
0xfe9ca914      void ObjectSynchronizer::wait(Handle,long long,Thread*) + 0x180
0xfe9ca524      JVM_MonitorWait + 0x2b8
0xfc00e1e8      * java.lang.Object.wait(long) bci:205735544 (Interpreted frame)
0xfc00e194      * java.lang.Object.wait(long) bci:0 (Interpreted frame)
```

```
0xfc005a30      * java.lang.ref.ReferenceQueue.remove(long) bci:44 line:116 (Interpreted frame)
0xfc0058c0      * java.lang.ref.ReferenceQueue.remove() bci:2 line:132 (Interpreted frame)
```

- None

6.5. Jstat

Used to diagnose performance issues and particularly issues related to heap sizing and garbage collection. Check additional parameters to be used with this command.

jstat -gcutil $PID <interval> <count>

```
S0     S1     E      O      P     YGC     YGCT    FGC    FGCT     GCT
0.00   0.00  49.99   8.31  53.40     11    1.646    19    2.031    3.677
0.00   0.00  49.99   8.31  53.40     11    1.646    19    2.031    3.677
0.00   0.00  49.99   8.31  53.40     11    1.646    19    2.031    3.677
```

The display shows the number of GC launched (part and full) and the gain in space of in each generation. Additional parameters allows zooming on different generation, example of zooming on the old generation:
{{/usr/local/java/jdk1.6.0_06/bin/jstat -gcoldcapacity -t $PID <interval> <count>}}

```
Timestamp          OGCMN       OGCMX        OGC          OC       YGC   FGC    FGCT     GCT
        67706.9     21888.0    349568.0     21888.0     21888.0    11    19    2.031    3.677
        67707.1     21888.0    349568.0     21888.0     21888.0    11    19    2.031    3.677
        67707.4     21888.0    349568.0     21888.0     21888.0    11    19    2.031    3.677
```

`jstat -class` displays multiple columns of statistics on the behavior of the class loader

```
Loaded  Bytes  Unloaded  Bytes     Time
  1937  2039.3       37    32.6       0.41
```

`jstat -compiler` displays multiple statistics tracing the behavior of the HotSpot Just-in-Time compiler

```
Compiled Failed Invalid   Time   FailedType FailedMethod
     181      0       0     2.10          0
```

6.6. Jhat utility

Stand for Java Heap Analysis Tool, the utility is used to debug unintentional object retention (memory leak) in core files. It provides a way to browse an object dump, view all reachable objects inthe heap, and understand which references are keeping an object alive.

When launched, the utility reserves a port on the Unix server in order to connect and browse the core file.

Jhat <core file>

```
-- Unix side
Reading from heap.dump.out...
Dump file created Thu Feb 26 17:41:59 MET 2009
Snapshot read, resolving...
Resolving 48234 objects...
Chasing references, expect 9 dots.........
Eliminating duplicate references.........
Snapshot resolved.
Started HTTP server on port 7000
Server is ready.

-- Explorer side
Package <Arrays>
class [Lmurex.apps.middleware.client.core.basicserver.BasicServerID; [0xb3ffab60]
class [Lmurex.apps.middleware.client.core.basicserver.BasicServerRemote; [0xb3ff6ff8]
class [Lmurex.apps.middleware.client.core.basicserver.MonitorRemote; [0xb3ff6bc8]
class [Lmurex.apps.middleware.client.core.basicserver.session.SessionID; [0xb3ff4538]
class [Lmurex.apps.middleware.client.core.launcher.LauncherServerID; [0xb4001628]
class [Lmurex.apps.middleware.client.core.launcher.LauncherServerRemote; [0xb4001468]

Other Queries
All classes including platform
Show all members of the rootset
Show instance counts for all classes (including platform)
Show instance counts for all classes (excluding platform)
```

```
Show heap histogram
Show finalizer summary
```

In addition to browsing the core and the objects, the jhat allows to check the heap histogram, the count of all classes and the access to the OQL which can be used to check values taken by classes.

6.7. jdb Utility

The JavaDebug Interface (JDI) is a high-level Java API that provides information useful for debuggers and similar systems that need access to the running state of a (usually remote) virtual machine.

```
$ jdb -connect sun.jvm.hotspot.jdi.SAPIDAttachingConnector:pid= $PID
```

6.8. Verbose Arguments

6.8.1. -verbose:gc

The flag enables analysis and performance tuning for java processes from a garbage collection point of view and report on each GC event. Best used with the following arguments:

- -XX:+PrintGCTimeStamps
- -XX:+PrintGCDetails

```
1.906: [GC [PSYoungGen: 19954K->1529K(23040K)] 26385K->16998K(47616K), 0.0837581 secs] [Times: user=0.11 sys=0.02, real=0.08 secs]
1.989: [Full GC [PSYoungGen: 1529K->0K(23040K)] [PSOldGen: 15469K->11041K(36864K)] 16998K->11041K(59904K) [PSPermGen: 3603K->3603K(16384K
Times: user=0.13 sys=0.00, real=0.13 secs]
2.404: [GC [PSYoungGen: 21496K->1526K(36928K)] 32538K->15353K(73792K), 0.0362621 secs] [Times: user=0.05 sys=0.01, real=0.04 secs]
jar/file.version saved.
bin/file.version saved.
4.289: [GC [PSYoungGen: 36918K->1532K(39424K)] 50745K->18652K(76288K), 0.0411807 secs] [Times: user=0.07 sys=0.01, real=0.04 secs]
```

Additional information found under:
http://java.sun.com/developer/technicalArticles/Programming/GCPortal/

6.8.2. -verbose:class

Display information about each class loaded. It enables logging of class loading and unloading.

```
[Loaded murex.apps.middleware.client.core.server.connection.engine.call.CallCache from file:/beast/apps/qa18720_TPK0000779_1313564/jar/middlew
[Loaded sun.reflect.GeneratedConstructorAccessor6 from __JVM_DefineClass__]
[Loaded sun.reflect.GeneratedMethodAccessor3 from __JVM_DefineClass__]
[Loaded murex.apps.middleware.client.core.launcher.CommandInput from file:/beast/apps/qa18720_TPK0000779_1313564/jar/middleware-client.jar]
[Loaded murex.apps.middleware.client.core.launcher.CommandOutput from file:/beast/apps/qa18720_TPK0000779_1313564/jar/middleware-client.jar]
[Loaded murex.apps.middleware.client.core.launcher.ServiceParameters from file:/beast/apps/qa18720_TPK0000779_1313564/jar/middleware-client.ja
[Loaded murex.apps.middleware.client.core.launcher.FileInfo from file:/beast/apps/qa18720_TPK0000779_1313564/jar/middleware-client.jar]
[Loaded sun.reflect.GeneratedSerializationConstructorAccessor18 from __JVM_DefineClass__]
[Loaded sun.reflect.GeneratedSerializationConstructorAccessor19 from __JVM_DefineClass__]
[Loaded java.io.StreamCorruptedException from /nfs_tools/SunOS/5.8/java/jdk1.6.0_06/jre/lib/rt.jar]
[Loaded sun.reflect.GeneratedSerializationConstructorAccessor20 from __JVM_DefineClass__]
[Loaded murex.shared.xml.XmlHeader from file:/beast/apps/qa18720_TPK0000779_1313564/jar/common-client.jar]
[Loaded sun.reflect.GeneratedSerializationConstructorAccessor21 from __JVM_DefineClass__]
[Loaded murex.shared.xml.parser.Mx2Translator from
[Loaded java.sql.Wrapper from /nfs_tools/SunOS/5.8/java/jdk1.6.0_06/jre/lib/rt.jar]
[Loaded java.sql.Statement from /nfs_tools/SunOS/5.8/java/jdk1.6.0_06/jre/lib/rt.jar]
[Loaded java.sql.PreparedStatement from /nfs_tools/SunOS/5.8/java/jdk1.6.0_06/jre/lib/rt.jar]
[Loaded java.sql.ResultSet from /nfs_tools/SunOS/5.8/java/jdk1.6.0_06/jre/lib/rt.jar]
2009-03-04 08:07:00,503 [main] console : INFO - Launcher [[ID:1187480974][ConnectionsAddress:[UserName:autoengine][HostName:beast][IPAddress:1
teName:site1][ClientRPCTransportConfig:[Protocol:RMI][Site:site1][RMIServerRPCRemote:RMIServerRPCTransport_Stub[UnicastRef2 [liveRef: [endpoin
:57717,murex.apps.middleware.client.core.server.transport.rpc.tcp.socket.DefaultRMISocketFactory@b1cc87,murex.apps.middleware.client.core.serv
.tcp.socket.DefaultRMISocketFactory@b1cc87](local),objID:[-51f6c477:11fd04e7f0f:-7ffd, 7576008055319130570]]]]]][PlatformName:MX][Installation
ERNATE]][[CreationTime:Wed Mar 04 08:06:59 MET 2009][Type:MX-XML Server][Description:LauncherHome Server][Source:null]]] started.
```

6.8.3. -verbose:jni

Report information about use of native methods and other Java Native Interface activity. Specifically, when a JNI native method is resolved, the Java VM prints a trace message to the application console (standard output). It also prints a trace message when a native method is registered using the JNI RegisterNative()() function. The -verbose:jni option may be useful when trying to diagnose issues with applications that use native libraries.

```
[Dynamic-linking native method murex.apps.shared.log.Logger.getLogLevel ... JNI]
[Dynamic-linking native method sun.security.pkcs11.wrapper.PKCS11.C_CreateObject ... JNI]
```

```
[Dynamic-linking native method java.lang.Runtime.totalMemory ... JNI]
[Dynamic-linking native method java.lang.Runtime.freeMemory ... JNI]
[Dynamic-linking native method java.util.zip.ZipFile.getNextEntry ... JNI]
[Dynamic-linking native method java.lang.SecurityManager.getClassContext ... JNI]
[Dynamic-linking native method java.lang.ProcessEnvironment.environ ... JNI]
[Dynamic-linking native method java.lang.UNIXProcess.initIDs ... JNI]
[Dynamic-linking native method java.lang.UNIXProcess.forkAndExec ... JNI]
[Dynamic-linking native method java.lang.UNIXProcess.waitForProcessExit ... JNI]
```

7. Kill -3

7.1. Usage

`Kill -3` prints the thread stack (into the related java process log file) of all java threads for a given process and.
To launch it, use the following commands:

```
kill -3 $PID
```

An example of the output you get is:

```
2009-06-01 15:15:24 Full thread dump Java HotSpot(TM) Server VM (10.0-b22 mixed mode):

"RMI TCP Connection(48)-172.21.26.85" daemon prio=3 tid=0x0046c000 nid=0x5b runnable [0xd0d7f000..0xd0d7f970]
   java.lang.Thread.State: RUNNABLE
        at java.net.SocketInputStream.socketRead0(Native Method)
        at java.net.SocketInputStream.read(SocketInputStream.java:129)
        at java.io.BufferedInputStream.fill(BufferedInputStream.java:218)
        at java.io.BufferedInputStream.read(BufferedInputStream.java:237)
        - locked <0xd419e3f8> (a java.io.BufferedInputStream)
        at java.io.FilterInputStream.read(FilterInputStream.java:66)
        at sun.rmi.transport.tcp.TCPTransport.handleMessages(TCPTransport.java:517)
        at sun.rmi.transport.tcp.TCPTransport$ConnectionHandler.run0(TCPTransport.java:790)
        at sun.rmi.transport.tcp.TCPTransport$ConnectionHandler.run(TCPTransport.java:649)
        at java.util.concurrent.ThreadPoolExecutor$Worker.runTask(ThreadPoolExecutor.java:885)
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:907)
        at java.lang.Thread.run(Thread.java:619)
[...]

"Thread-6" daemon prio=3 tid=0x0046b400 nid=0x19 in Object.wait() [0xd0e7f000..0xd0e7fa70]
   java.lang.Thread.State: WAITING (on object monitor)
        at java.lang.Object.wait(Native Method)
        - waiting on <0xd3c64ce8> (a murex.apps.middleware.client.shared.thread.worker.Worker)
        at java.lang.Object.wait(Object.java:485)
        at murex.apps.middleware.client.shared.thread.worker.Worker.run(Worker.java:85)
        - locked <0xd3c64ce8> (a murex.apps.middleware.client.shared.thread.worker.Worker)

"DestroyJavaVM" prio=3 tid=0x00033400 nid=0x2 waiting on condition [0x00000000..0xfe4ffb58]
   java.lang.Thread.State: RUNNABLE

[...]

"Thread-3" daemon prio=3 tid=0x00c38800 nid=0x15 in Object.wait() [0xd107f000..0xd107f870]
   java.lang.Thread.State: TIMED_WAITING (on object monitor)
        at java.lang.Object.wait(Native Method)
        - waiting on <0xdf342bb0> (a murex.apps.middleware.client.shared.thread.timer.TaskRunner)
        at murex.apps.middleware.client.shared.thread.timer.TaskRunner.startWork(TaskRunner.java:121)
        - locked <0xdf342bb0> (a murex.apps.middleware.client.shared.thread.timer.TaskRunner)
        at murex.apps.middleware.client.shared.thread.worker.Worker.run(Worker.java:97)

"RMI RenewClean-[172.21.26.85:51878,murex.apps.middleware.client.core.server.transport.rpc.tcp.socket.DefaultRMISocketFactory@da2cef]" daemon
8f800 nid=0x13 in Object.wait() [0xd127f000..0xd127f970]
   java.lang.Thread.State: TIMED_WAITING (on object monitor)
        at java.lang.Object.wait(Native Method)
        - waiting on <0xd46709b0> (a java.lang.ref.ReferenceQueue$Lock)
        at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:116)
        - locked <0xd46709b0> (a java.lang.ref.ReferenceQueue$Lock)
        at sun.rmi.transport.DGCClient$EndpointEntry$RenewCleanThread.run(DGCClient.java:516)
        at java.lang.Thread.run(Thread.java:619)
[...]
```

```
"RMI Scheduler(0)" daemon prio=3 tid=0x001f3c00 nid=0x10 waiting on condition [0xd157f000..0xd157fbf0]
        - parking to wait for  <0xdf390a08> (a java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject)
        at java.util.concurrent.locks.LockSupport.parkNanos(LockSupport.java:198)
        at java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.awaitNanos(AbstractQueuedSynchronizer.java:1963)
        at java.util.concurrent.DelayQueue.take(DelayQueue.java:164)
        at java.util.concurrent.ScheduledThreadPoolExecutor$DelayedWorkQueue.take(ScheduledThreadPoolExecutor.java:582)
        at java.util.concurrent.ScheduledThreadPoolExecutor$DelayedWorkQueue.take(ScheduledThreadPoolExecutor.java:575)
        at java.util.concurrent.ThreadPoolExecutor.getTask(ThreadPoolExecutor.java:946)
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:906)
        at java.lang.Thread.run(Thread.java:619)

"GC Daemon" daemon prio=3 tid=0x00c1f800 nid=0xf in Object.wait() [0xd167f000..0xd167fb70]
   java.lang.Thread.State: TIMED_WAITING (on object monitor)
        at java.lang.Object.wait(Native Method)
        - waiting on <0xdf391358> (a sun.misc.GC$LatencyLock)
        at sun.misc.GC$Daemon.run(GC.java:100)
        - locked <0xdf391358> (a sun.misc.GC$LatencyLock)

"RMI Reaper" prio=3 tid=0x00c3ac00 nid=0xe in Object.wait() [0xd177f000..0xd177f8f0]
   java.lang.Thread.State: WAITING (on object monitor)
        at java.lang.Object.wait(Native Method)
        - waiting on <0xdf390328> (a java.lang.ref.ReferenceQueue$Lock)
        at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:116)
        - locked <0xdf390328> (a java.lang.ref.ReferenceQueue$Lock)
        at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:132)
        at sun.rmi.transport.ObjectTable$Reaper.run(ObjectTable.java:333)
        at java.lang.Thread.run(Thread.java:619)

"RMI TCP Accept-0" daemon prio=3 tid=0x00c4a800 nid=0xd runnable [0xd187f000..0xd187f870]
   java.lang.Thread.State: RUNNABLE
        at java.net.PlainSocketImpl.socketAccept(Native Method)
        at java.net.PlainSocketImpl.accept(PlainSocketImpl.java:384)
        - locked <0xdf3904c0> (a java.net.SocksSocketImpl)
        at java.net.ServerSocket.implAccept(ServerSocket.java:453)
        at murex.apps.middleware.client.core.server.transport.rpc.tcp.socket.DefaultServerSocket.accept(DefaultServerSocket.java:17)
        at sun.rmi.transport.tcp.TCPTransport$AcceptLoop.executeAcceptLoop(TCPTransport.java:369)
        at sun.rmi.transport.tcp.TCPTransport$AcceptLoop.run(TCPTransport.java:341)
        at java.lang.Thread.run(Thread.java:619)

"murex.apps.middleware.client.shared.thread.worker.WorkerManager@203c31[ID:0][Daemon:true][MaxWorker:0][MaxInactivityTime:300000][Valid:true][
WorkingWorkers:[]]" daemon prio=3 tid=0x00c10c00 nid=0xc in Object.wait() [0xd197f000..0xd197f9f0]
   java.lang.Thread.State: TIMED_WAITING (on object monitor)
        at java.lang.Object.wait(Native Method)
        - waiting on <0xdf343180> (a murex.apps.middleware.client.shared.thread.worker.WorkerManager)
        at murex.apps.middleware.client.shared.thread.worker.WorkerManager.run(WorkerManager.java:123)

[...]

"murex.apps.middleware.client.shared.thread.worker.WorkerManager@203c31[ID:0][Daemon:true][MaxWorker:0][MaxInactivityTime:300000][Valid:true][
WorkingWorkers:[]]" daemon prio=3 tid=0x00c10c00 nid=0xc in Object.wait() [0xd197f000..0xd197f9f0]
   java.lang.Thread.State: TIMED_WAITING (on object monitor)
        at java.lang.Object.wait(Native Method)
        - waiting on <0xdf343180> (a murex.apps.middleware.client.shared.thread.worker.WorkerManager)
        at murex.apps.middleware.client.shared.thread.worker.WorkerManager.run(WorkerManager.java:123)
        - locked <0xdf343180> (a murex.apps.middleware.client.shared.thread.worker.WorkerManager)
        at java.lang.Thread.run(Thread.java:619)

"Low Memory Detector" daemon prio=3 tid=0x000d7800 nid=0x9 runnable [0x00000000..0x00000000]
   java.lang.Thread.State: RUNNABLE

"CompilerThread1" daemon prio=3 tid=0x000d5400 nid=0x8 waiting on condition [0x00000000..0xd367ef18]
   java.lang.Thread.State: RUNNABLE

"CompilerThread0" daemon prio=3 tid=0x000d3800 nid=0x7 waiting on condition [0x00000000..0xd377ee98]
   java.lang.Thread.State: RUNNABLE

"Signal Dispatcher" daemon prio=3 tid=0x000d2400 nid=0x6 waiting on condition [0x00000000..0x00000000]
   java.lang.Thread.State: RUNNABLE

"Finalizer" daemon prio=3 tid=0x000bd800 nid=0x5 in Object.wait() [0xd397f000..0xd397f870]
```

```
   java.lang.Thread.State: WAITING (on object monitor)
        at java.lang.Object.wait(Native Method)
        - locked <0xde6a0298> (a java.lang.ref.ReferenceQueue$Lock)
        at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:132)
        at java.lang.ref.Finalizer$FinalizerThread.run(Finalizer.java:159)

"Reference Handler" daemon prio=3 tid=0x000bc800 nid=0x4 in Object.wait() [0xd3a7f000..0xd3a7f9f0]
   java.lang.Thread.State: WAITING (on object monitor)
        at java.lang.Object.wait(Native Method)
        - waiting on <0xde6a0320> (a java.lang.ref.Reference$Lock)
        at java.lang.Object.wait(Object.java:485)
        at java.lang.ref.Reference$ReferenceHandler.run(Reference.java:116)
        - locked <0xde6a0320> (a java.lang.ref.Reference$Lock)

"VM Thread" prio=3 tid=0x000b9400 nid=0x3 runnable

"VM Periodic Task Thread" prio=3 tid=0x00103400 nid=0xa waiting on condition

JNI global references: 1042

Heap
 def new generation   total 10688K, used 6300K [0xd3c00000, 0xd4790000, 0xde6a0000)
  eden space 9536K,  60% used [0xd3c00000, 0xd41a9fb8, 0xd4550000)
  from space 1152K,  43% used [0xd4670000, 0xd46ed208, 0xd4790000)
  to   space 1152K,   0% used [0xd4550000, 0xd4550000, 0xd4670000)
 tenured generation   total 23664K, used 14195K [0xde6a0000, 0xdfdbc000, 0xf3c00000)
   the space 23664K,  59% used [0xde6a0000, 0xdf47cfc0, 0xdf47d000, 0xdfdbc000)
 compacting perm gen  total 16384K, used 8781K [0xf3c00000, 0xf4c00000, 0xfbc00000)
   the space 16384K,  53% used [0xf3c00000, 0xf4493420, 0xf4493600, 0xf4c00000)
No shared spaces configured.
```

7.2. Architecture of thread

Also known as java core, the thread dump is generated by the `kill -3` command, used for viewing the thread activity inside the JVM at a given time. The header line contains the following information about the thread:

- Thread name: indication if the thread is a daemon thread
- Thread priority
- Thread ID (tid), which is the address of a thread structure in memory ID of the native thread (nid)
- Thread state, which indicates what the thread was doing at the time of the thread dump
- Address range, which gives an estimate of the valid stack region for the thread

Below is a list of possible thread states that can be printed and their meaning:

- `NEW`: The thread has not yet started.
- `RUNNABLE`: The thread is executing in the Java virtual machine.
- `BLOCKED`: The thread is blocked waiting for a monitor lock.
- `WAITING`: The thread is waiting indefinitely for another thread to perform a particular action.
- `TIMED_WAITING`: The thread is waiting for another thread to perform an action for up to a specified waiting time.
- `TERMINATED`: The thread has exited.

8. Fatal error

In case of Fatal error on the java process, an error file is created and named hs_err_<pid>.log.

Find below an explanation of the architecture and different sections of this error file.

8.1. Header format

Contains a brief description of the problem.

SIGSEGV (0xb) at pc=0x417789d7, pid=21139, tid=1024

			+- thread id
		+---------- process id	
	+-------------------------- program counter (instruction pointer)		
+---------------------------------- signal number			
C	Native C frame		

The next line contains the VM version, an indication whether the application was run in mixed or interpreted mode and an indication whether class file sharing was enabled.

The following information is the function frame that caused the crash

C	Native C frame

v	VMgenerated stub frame
J	Other frame types, including compiled Java frames

The problematic frame indicates the primary class where the crash occurred.

8.2. Thread format

If multiple threads crash at the same time, only one thread is printed.

8.2.1. Thread information

The first part of the thread section shows the thread that provoked the fatal error:

Current thread (0x0805ac88): JavaThread "main" _thread_in_native, id=21139

			+ID
		+---------- state	
	+--------------------------- name		
+-- type			
_thread_uninitialized	Thread is not created. This occurs only in the case of memory corruption		

The list of thread state can be found below

_thread_uninitialized	Thread is not created. This occurs only in the case of memory corruption
_thread_new	Thread has been created but it has not yet started.
_thread_in_native	Thread is running native code. The error is probably a bug in native code
_thread_in_vm	Thread is runningVMcode.
_thread_in_Java	Thread is running either interpreted or compiled Java code.
_thread_blocked	Thread is blocked.
..._trans	If any of the above states is followed by the string _trans, that means the thread is changing to a different state

Possible thread types: JavaThread, VMThread, CompilerThread, JVMPIDaemonthread, GCTaskThread, WatcherThread, ConcurrentMarkSweepThread.

8.2.2. Signal information:

Describes the unexpected signal that caused the VM to terminate

```
siginfo: ExceptionCode=0xc0000005, reading address 0xd8ffecf1
```

8.2.3. Register Context & Machine Instructions:

The register values might be useful when combined with instructions. Instructions (opcodes) can be decoded with a disassembler to produce the instructions around the location of the crash.

8.2.4. Thread Stack

Includes the addresses of the base and the top of the stack, the current stack pointer, and the amount of unused stack available to the thread

Thread stack example:

```
Stack: [0x00040000,0x00080000), sp=0x0007f9f8, free space=254k
Native frames: (J=compiled Java code, j=interpreted, Vv=VM code, C=native code)
V [jvm.dll+0x83d77]
C [App.dll+0x1047]
j Test.foo()V+0
j Test.main([Ljava/lang/String;)V+0
v ~StubRoutines::call_stub
V [jvm.dll+0x80f13]
V [jvm.dll+0xd3842]
V [jvm.dll+0x80de4]
C [java.exe+0x14c0]
C [java.exe+0x64cd]
```

```
C [kernel32.dll+0x214c7]
```

```
j Test.main([Ljava/lang/String;)V+0
v ~StubRoutines::call_stub
```

The first thread stack is Native frames, which prints the native thread showing all function calls. The information in the thread stack for native frames provides important information about the cause of the crash. By analyzing the libraries in the list from the top down, you can generally determine which library might have caused the problem and report it to the appropriate organization responsible for that library

The second thread stack is Java frames, which prints the Java frames including the inlined methods, skipping the native frames. Depending on the crash it might not be possible to print the native thread stack but it might be possible to print the Java frames

8.3. Process section

The process section is printed after the thread section. It contains information about the whole process, including thread list and memory usage of the process.

8.3.1. Thread List

The thread list includes the threads that theVMis aware of. This includes all Java threads and some VM internal threads, but does not include any native threads created by the user application that have not attached to the VM. The output format follows.

```
=>0x0805ac88 JavaThread "main" [_thread_in_native, id=21139]
|       |               |       |       |               +----- ID
|       |               |       |       +------------------- state (JavaThread only)
|       |               |       +--------------------------------- name
|       |               +------------------------------------------ type
|       +------------------------------------------------------ pointer
+-------------------------------------------------------------- "=>" current thread
```

VMState: Indicates the overall state of the virtual machine.

8.3.2. Mutexes and Monitors mutexes

The mutexes are VM internal locks. For each lock the log contains the name of the lock, its owner, and the addresses of a VM internal mutex structure and its OS lock. In general this information is useful only to those who are very familiar with the HotSpot VM

8.3.3. Heap Summary

Snapshot of the heap. Output depends on the garbage collection (GC) configuration.

8.3.4. MemoryMap and Arguments

Indicates the libraries that are actually being used, the permission, their location in memory, as well as the location of heap, stack, and guard pages. The format of the memory map is operating-system-specific.

The arguments list both the VM arguments and the environment variables used by the process.

Last section include the Signal handlers and the system information

8.4. C++filt

You can do a c++filt to demangle the stacks present in the hs_err and search for them on the internet for more information.

8.5. core file size on Linux

The hs_err file has what looks like a pmap output on the process, so getting the size of each memory segment and summing them all up gives us the size of the process when it crashed

```
awk --non-decimal-data ' BEGIN \{FS="[- ]"\} \{ if ( $0 ~ "[0-9a-f]+-[0-9a-f]+ +[r-][w-][x-][ps]") \{sum +=(("0x"$2) - ("0x"$1))\} \} END \{ p
```

8.6. empty hs_err

In case you face empty hs_err<PID>.log, try to add the following lines in the mxg2000_settings.sh, restart the services and re-produce. A core file should be generated

```
LD_PRELOAD=libumem.so.1
UMEM_DEBUG=default
export LD_PRELOAD
export UMEM_DEBUG
```

Another approach will be using the truss functionality or increasing the MXJ_JVM of the mx process.

An empty hs_err was generated on the Adapt tables of an Oracle environment when the /RDBMS_STATISTICS was set to 4 and the last query showed a creation of a sequence. setting it to 1 and re-launchign the adapt tables solved the issue.

9. Platform tools

This section presents a listing of some common Solaris platform-specific tools that can be used to monitor java processes.

Tool	Description	
pfiles	Print information on process file descriptors. Starting with Solaris 10 OS,the tool prints the filename also.	
pflags	Print the /proc tracing flags, the pending and held signals, and other /proc status information for each lwp in each process.	
pldd	List the dynamic libraries linked into each process, including shared objects.	
pmap	Print memory layout of a process, including heap, data, text sections. Starting with Solaris 10 OS, stack segments are clearly identified with the text stack along with the thread ID.	
prstat	Report statistics for active Solaris OS processes. (Similar to top.).	
psig	List the signal handlers of a process.	
truss	Trace entry and exit events for system calls, user-mode functions, and signals; optionally stop the process at one of these events. This tool also prints the arguments of system calls and user functions.	

9.1. pfiles

pfiles is used to check what files a specific process is accessing at the moment and to list file descriptor details.

pfiles can be used to check for big size jars that are accessed when a process fails(ex: fs or xmls) with an OutOfMemoryError.

9.1.1. Usage

```
pfiles \{PID\}
```

A grep command can be added to the pfiles command to restrict the output on the AF_INET address family sockets.
This would be useful to check on zombie processes.
The corresponding command line to use for this is the following:

```
pfile \{PID\} | grep AF_INET
```

9.1.2. Output

Given a specific process ID (ex: 28446), the pfiles command will output the following:

```
bash$ pfiles 8176
8176:   java -server -Xms32M -Xmx256M -showversion -XX:+UseSerialGC -Xbootclas
  Current rlimit: 1024 file descriptors
   0: S_IFCHR mode:0666 dev:316,0 ino:6815752 uid:0 gid:3 rdev:13,2
      O_RDONLY|O_LARGEFILE
      /devices/pseudo/mm@0:null
   1: S_IFREG mode:0664 dev:118,110 ino:1479183 uid:5012 gid:5000 size:76453
      O_WRONLY|O_LARGEFILE
      /beast2/apps/qa19128_TPK0000779_1463720/logs/beast.mandatory.site1.murex.mxres.common.launchermandatory.mxres.log
   2: S_IFREG mode:0664 dev:118,110 ino:1479183 uid:5012 gid:5000 size:76453
      O_WRONLY|O_LARGEFILE
      /beast2/apps/qa19128_TPK0000779_1463720/logs/beast.mandatory.site1.murex.mxres.common.launchermandatory.mxres.log
```

If the output is restricted to the AF_INET address family sockets using the grep command explained above:

```
bash$ pfiles 23283 | grep AF_INET
        sockname: AF_INET 172.21.17.162  port: 19140
        sockname: AF_INET 0.0.0.0  port: 55424
```

9.1.3. Output analysis

- The first line shows the current limit on filehandles.
- Then the output will show three lines per file:
  - The leftmost number is the file handle number and after the number is permissions on the file, what device, inode, owner,group and size.
  - The second line are the options used to open the files
  - and the third line is the name of the file that is open.

When the output is restricted to the AF_INET address family sockets, it will show on each line the socket address with the corresponding port numbe
guilty zombie process may be residing.

9.2. pflags

9.2.1. Usage

```
pflags \{ PID | core \}
```

9.2.2. Output

```
bash$ pflags 23810
23810:  java -Xms32M -Xmx64M -showversion -XX:+UseSerialGC -Xbootclasspath/p:j
    data model = _ILP32  flags = ORPHAN|MSACCT|MSFORK
 /1:    flags = ASLEEP  lwp_wait(0x2,0xffbfe4dc)
    sigmask = 0x00000004,0x00000000
...
```

9.2.3. Output analysis

The output includes the mode32-bit or 64-bitin which the process is running and the current state for each thread within the process. In addition, the top-level function on each thread's stack is displayed.

9.3. pldd

pldd list the dynamic libraries linked into each process or a core, including shared objects explicitly attached.Given a pid, pldd prints a list of shared libraries loaded, including those loaded explicitly.
pldd works by attaching to the process to read its memory.

9.3.1. Usage

```
pldd \{pid | core\}
```

9.3.2. Output

Given a specific process ID (ex: 9820), the pldd command will output the following:

```
bash$ pldd 9820
9820:   java -cp mxjboot.jar -Djava.security.policy=java.policy -Djava.rmi.ser
/lib/libthread.so.1
/lib/libdl.so.1
/lib/libc.so.1
/platform/sun4u-us3/lib/libc_psr.so.1
/nfs_tools/SunOS/5.8/java/j2sdk1.4.2_09/jre/lib/sparc/client/libjvm.so
/usr/lib/libCrun.so.1
/lib/libsocket.so.1
/lib/libnsl.so.1
```

9.3.3. Output Analysis

- The first line shows the current process ID along with its corresponding executed command.
- The remaining lines shows the libraries linked to this process, one library per line.

9.4. pmap

The pmap utility prints information about the address space of a process.

Here is a list of available options supported by pmap:

Option	Description
-a	Prints anonymous and swap reservations for shared mappings.
-A <address range>	Specify the subrange of address space to display as one of the following:
	• <start addr>: The single address limits the output to the segment (or the page if the -L option is present) containing that address. If the specified address corresponds to the starting address of a segment, the output will always include the whole segment even when the -L option is given.
	• <start addr>,: An address followed by comma without the end address limits the output to all segments (or pages if the -L option is

Option	Description
	present) starting from the one containing the specified address.
	address.
	• ,<end addr>: An address started with comma without the start address limits the output to all segments (or pages if the -L option is present) starting from the first one present until the segment (or page if the -L option is present) containing the specified address.
-F	Force. Grabs the target process even if another process has control.
-l	Shows unresolved dynamic linker map names.
-r	Prints the process's reserved addresses.
-s	Prints HAT page size information
-S	Displays swap reservation information per mapping.
-x	Displays additional information per mapping.

9.4.1. Process mapping

By default pmap displays all of the mapping in the virtual address order they are mapped into the given process.

9.4.2. Usage

```
pmap \{pid\}
```

9.4.3. Output

Given a process ID (ex: 17071), executing the pmap command without any of the optional arguments will output the following:

```
bash$ pmap 17071
17071:  java -Xms32M -Xmx64M -showversion -cp murex/code/kernel/jar/fileserver
00010000      48K r-x--  /nfs_tools/SunOS/5.8/java/jdk1.6.0_06/bin/java
0002A000       8K rwx--  /nfs_tools/SunOS/5.8/java/jdk1.6.0_06/bin/java
0002C000    3920K rwx--    [ heap ]
00400000    4096K rwx--    [ heap ]
F2BF6000      40K rwx-R    [ anon ]
F2CF4000      48K rwx-R    [ anon ]
F2DF6000      40K rwx-R    [ stack tid=290 ]
F2E16000     608K r--s-  dev:118,110 ino:1876509
...
 total     81160K
```

9.4.4. Output analysis

- The first line shows the current process ID along with its corresponding executed command (This is true for all pmap outputs).
- Then the first column represents the starting virtual address of each mapping. Virtual addresses are displayed in ascending order.
- The second column shows the virtual size in kilobytes of each mapping.
- The third column shows permission flags.The virtual memory permissions are shown for each mapping.
  - Valid permissions are:

Flag	Description
r	The mapping may be read by the process.
w	The mapping may be written by the process.
x	Instructions that reside within the mapping may be executed by the process.

- Flags showing additional information for each mapping may be also displayed:

Additional flag	Description
s	The mapping is shared such that changes made in the observed address space are committed to the mapped file, and are visible from all other processes sharing the mapping.
R	Swap space is not reserved for this mapping.Mappings created with MAP_NORESERVE and System V ISM shared memory mappings do not reserve swap space.

- The fourth column shows a descriptive name for each mapping. The following major types of names are displayed for mappings.

Mapping name type	Description
A mapped file name	For mappings between a process and a file, the pmap command attempts to resol[illegible] name for each mapping. If the file name cannot be resolved, pmap displays

Mapping name type	Description
	number of the device containing the file, and the file system inode number of the file. If the common name for the mapping is unknown, [anon] is also displayed as the mapping name.
[heap]	The mapping is the process heap.
[stack]	The mapping is the main stack
[stack tid=n]	The mapping is the stack of thread n
[altstack tid=n]	The mapping is used as the alternate signal stack for thread n
System V shared memory:Mappings created using System V shared memory system calls are reported with the names shown below:	
shmid=n	The mapping is a System V shared memory mapping. The shared memory identifier that the mapping was created with is reported.
ism shmid=n	The mapping is an "Intimate Shared Memory" variant of System V shared memory
dism shmid=n	The mapping is a pageable variant of ISM.
Other	Mappings of other objects, including devices such as frame buffers. No mapping name is shown for other mapped objects.

- The last line shows the total of memory size in kilobytes.

9.4.5. Process anon/locked mapping details

The -x option displays additional information per mapping. The size of each mapping, the amount of resident physical memory (RSS), the amount of anonymous memory, and the amount of memory locked is shown with this option. This does not include anonymous memory taken by kernel address space due to this process.

9.4.5.1. Usage

```
pmap -x  \{pid\}
```

9.4.5.2. Output

When used with the -x argument option, pmap will output the following:

```
bash$ pmap -x 17071
17071:  java -Xms32M -Xmx64M -showversion -cp murex/code/kernel/jar/fileserver
 Address  Kbytes     RSS    Anon  Locked Mode   Mapped File
00010000      48      48       -       - r-x--  java
0002A000       8       8       8       - rwx--  java
0002C000    3920    3408    3408       - rwx--    [ heap ]
00400000    4096    4096    4096       - rwx--    [ heap ]
F2BF6000      40      40      40       - rwx-R    [ anon ]
F2CF4000      48      48      48       - rwx-R    [ anon ]
F2DF6000      40      40      40       - rwx-R    [ stack tid=290 ]
F2E16000     608     608       -       - r--s-  dev:118,110 ino:1876509
...
-------- ------- ------- ------- -------
total Kb   81160   46848   28592       -
```

9.4.5.3. Output analysis

- The first line shows the current process ID along with its corresponding executed command (This is true for all pmap outputs).
- Then the first column represents the starting virtual address of each mapping. Virtual addresses are displayed in ascending order.
- The second column shows the virtual size in kilobytes of each mapping.
- The third column shows the amount of Resident Physical Memory (RSS):
  this is the amount of physical memory resident for each mapping, including that which is shared with other address spaces.
- The fourth column shows the amount of Anonymous Memory ANON:this is the amount of dedicated memory consumed by the process.
- The fifth column shows the number of pages locked within the mapping if available.
- The sixth column shows permission flags (describded in previous section).
- The last column shows a descriptive name for each mapping (described in previous section).
- The last line shows the total of memory size in kilobytes for each of the columns that involve memory sizing (Kbytes, RSS, Anon and Locked).

9.5. prstat

The prstat utility iteratively examines all active processes on the system and reports statistics based on the selected output mode and sort order.

9.6. Get a restricted report on java processes:

```
prstat -s size -n 400 1 1 | grep java
```

The -s size option sorts the output according to the size of process image.

The -n 400 1 1 option restricts the number of output lines.

The grep java command restricts the output to java processes.

9.6.1.2. Output

```
bash$ prstat -s size -n 400 1 1 | grep java
  9885 autoengi 1230M  209M sleep   29   10  13:00:42 0.1% java/70
  5391 noaccess  190M   91M sleep   59    0   1:07:16 0.0% java/31
  9820 autoengi  180M   46M sleep   29   10   0:13:06 0.0% java/8
...
```

9.7. Get the micro statistics on a process:

9.7.1.1. Usage

```
prstat -m -p {PID}
```

The -m option reports microstate process accounting information.

The -p option reports only the process(es) whose ID(s) is(are) given.

9.7.1.2. Output

```
bash$ prstat -m -p 23810
   PID USERNAME USR SYS TRP TFL DFL LCK SLP LAT VCX ICX SCL SIG PROCESS/NLWP
 23810 autoengi 0.0 0.0 0.0 0.0 0.0  86  14 0.0 146   0  94   0 java/21
```

9.8. Get the Resource Statistics for each thread within a process:

9.8.1.1. Usage

```
prstat -Lp \{NPID\}
```

The -L option reports statistics for each light-weight process (LWP).
The -p option reports only the process(es) whose ID(s) is(are) given.

on Linux, two methods are available:

- Use the following command: `ps -C java -L -o pid,tid,pcpu,time,size,rss | sort -rk4 | grep <pid>`. Note that tid corresponds to the thread id. Its hex representation corresponds to the nid in the jstack.
- Use the `top -n <pid>` command, followed by `Shift + H` to list the process threads, then `Shift + T` to sort by cpu time. Note that the PID column would correspond to the thread id. Its hex representation corresponds to the nid in the jstack.

9.8.1.2. Output

```
bash$ prstat -Lp 23810
   PID USERNAME  SIZE   RSS STATE  PRI NICE      TIME  CPU PROCESS/LWPID
 23810 autoengi   89M   73M sleep   59    0   0:00:02 0.0% java/10
 23810 autoengi   89M   73M sleep   59    0   0:00:00 0.0% java/25
 23810 autoengi   89M   73M sleep   59    0   0:00:00 0.0% java/22
 23810 autoengi   89M   73M sleep   59    0   0:00:00 0.0% java/21
 23810 autoengi   89M   73M sleep   59    0   0:00:00 0.0% java/19
 23810 autoengi   89M   73M sleep   59    0   0:00:00 0.0% java/17
 23810 autoengi   89M   73M sleep   59    0   0:00:00 0.0% java/16
 23810 autoengi   89M   73M sleep   59    0   0:00:00 0.0% java/15
 23810 autoengi   89M   73M sleep   59    0   0:00:00 0.0% java/14
...
```

The LWPID corresponds to the nid in hexadecimal inside the thread. hence, value 3512 corresponds to db8 that can be checked inside the jstack or kill -3.

```
"RMI TCP Connection(4499)-172.21.25.222" daemon prio=3 tid=0x09073000 nid=0xdb8 runnable [0xad46b000]
```

the `-s time` parameter is also useful for sorting the threads by the most consuming cpu wise:

```
prstat -s time -L -p \{PID\}
```

9.8.2. Outputs Analysis

The following list defines the column headings and the meanings of a prstat report.

column heading	meaning
PID	The process ID of the process
USERNAME	The real user (login) name or real user ID
SIZE	The total virtual memory size of the process, including all mapped files and devices, in kilobytes(K), megabytes(M), or gigabytes(G)
RSS	The resident set size of the process (RSS), in kilobytes (K), megabytes (M), or gigabytes (G).
STATE	The state of the process:
	• cpuN: Process is running on CPU N.
	• sleep: Sleeping: process is waiting for an event to complete.
	• run: Runnable: process in on run queue.
	• zombie: Zombie state: process terminated and parent not waiting.
	• stop: Process is stopped.
PRI	The priority of the process. Larger numbers mean higher priority
NICE	Nice value used in priority computation. Only processes in certain scheduling classes have a nice value. Default nice value is 20. To check the default nice value do a 'ps -elf' and look at the NI field.
TIME	The cumulative execution time for the process.
CPU	The percentage of recent CPU time used by the process. If executing in a non-global zone and the pools facility is active, the percentage will be that of the processors in the processor set in use by the pool to which the zone is bound.
PROCESS	The name of the process (name of executed file).
NLWP	The number of lwps in the process.

The following columns are displayed when the -v or -m option is specified with the prstat command:

column heading	meaning
USR	The percentage of time the process has spent in user mode
SYS	The percentage of time the process has spent in system mode
TRP	The percentage of time the process has spent in processing system traps
TFL	The percentage of time the process has spent processing text page faults.
DFL	The percentage of time the process has spentprocessing data page faults.
LCK	The percentage of time the process has spentwaiting for user locks.
SLP	The percentage of time the process has spent sleeping.
LAT	The percentage of time the process has spent waiting for CPU.
VCX	The number of voluntary context switches.
ICX	The number of involuntary context switches.
SCL	The number of system calls.
SIG	The number of signals received.

9.9. psig

psig is used to list the signal actions and handlers of each process. It will print a list showing what the process' response will be to each of the different available Solaris signals. A signal is a message sent to a process to interrupt it and cause a response. If the process has been designed to respond to signals of the type sent, it does so.

9.9.1. Usage

9.9.2. Output

Given a specific process ID (ex:4632), the psig command will output the following:

```
bash$ psig 5632
5632:   java -Xms32M -Xmx64M -showversion -XX:+UseSerialGC -Xbootclasspath/p:j
HUP     ignored
INT     ignored
QUIT    caught  sigacthandler   RESTART HUP,INT,QUIT,ILL,TRAP,ABRT,EMT,FPE,BUS,SEGV,SYS,PIPE,ALRM,TERM,USR1,USR2,CLD,PWR,WINCH,URG,POLL,TSTP,C
ILL     caught  sigacthandler   RESTART,SIGINFO HUP,INT,QUIT,ILL,TRAP,ABRT,EMT,FPE,BUS,SEGV,SYS,PIPE,ALRM,TERM,USR1,USR2,CLD,PWR,WINCH,URG,POL
TRAP    default
ABRT    default
...
```

9.9.3. Output analysis

- The first column shows the names of the different available signals. The table below provides a complete list of signals, along with a description and default action. Every signal has a unique signal name, an abbreviation that begins with SIG. For all possible signals, the system defines a default action to be taken when a signal occurs.
  There are 4 possible default actions:
- Exit: forces the process to exit
- Core: forces the process to exit, and creates a core file
- Stop: Stops the process
- Ignore: Ignores the signal; no action taken

Name	Description	Default action
SIGHUP	Hangup	Exit
SIGINT	Interrupt	Exit
SIGQUIT	Core	Quit
SIGILL	Illegal Instruction	Core
SIGTRAP	Trace or breakpoint trap	Core
SIGABRT	Abort	Core
SIGEMT	Emulation trap	Core
SIGFPE	Arithmetic exception	Core
SIGKILL	Kill	Exit
SIGBUS	Bus error – actually a misaligned address error	Core
SIGSEGV	Segmentation fault – an address reference boundary error	Core
SIGSYS	Bad system call	Core
SIGPIPE	Broken pipe	Exit
SIGALRM	Alarm clock	Exit
SIGTERM	Terminated	Exit
SIGUSR1	User defined signal 1	Exit
SIGUSR2	User defined signal 2	Exit
SIGCHLD	Child process status changed	Ignore
SIGPWR	Power fail or restart	Ignore
SIGWINCH	Window size change	Ignore
SIGURG	Urgent socket condition	Ignore
SIGPOLL	Pollable event	Exit
SIGSTOP	Stop (cannot be caught or ignored)	Stop
SIGTSTP	Stop (job control, e.g., ^z)	Stop
SIGCONT	Continued	Ignore
SIGTTIN	Stopped – tty input	Stop
SIGTTOU	Stopped – tty output	Stop

Name	Description	Default action
SIGXCPU	CPU time limit exceeded	Core
SIGXFSZ	File size limit exceede	Core
SIGWAITING	Concurrency signal used by threads library	Ignore
SIGLWP	Inter-LWP signal used by threads library	Ignore
SIGFREEZE	Checkpoint suspend	Ignore
SIGTHAW	Checkpoint resume	Ignore
SIGCANCEL	Cancellation signal used by threads library	Ignore
SIGLOST	Resource lost	Ignore
SIGRTMIN	Highest priority realtime signal	Exit
SIGRTMAX	Lowest priority realtime signal	Exit

- The second column describes the process response to the corresponding signal. The action to be taken by the process when a signal is received can be changed from the default one with the exception of SIGKILL and SIGSTOP that can't be changed.
  - default: means that the default action will be taken
  - ignored: means that the signal will be ignored even if its default action was not Ignore
  - caught: means that the signal will be caught but instead of taking the default action a signal handling routine will be invoked. In this case, a third and fourth column will also appear in the output.
- The third and fourth columns will show the signal handling routine that will be invoked when the signal is caught and not the default action will be taken.

9.10. truss

Truss is used to trace the system/library calls (not user calls) and signals made/received by a new or existing process.executes a specified command, or attaches to listed process IDs, and produces a trace of the system calls, received signals, and machine faults a process incurs.

Here is a list of argument options supported by truss:

Option	Description
-a	Displays the parameter strings which are passed in each exec system call.
-c	Counts traced system calls, faults, and signals rather than displaying trace results line by line. A summary report is produced after the traced command terminates or when truss is interrupted. If the -f flag is also used, the counts include all traced Syscalls, Faults, and Signals for child processes.
-d	A timestamp will be included with each line of output. Time displayed is in seconds relative to the beginning of the trace. The first line of the trace output will show the base time from which the individual time stamps are measured. By default timestamps are not displayed.
-D	Delta time is displayed on each line of output. The delta time represents the elapsed time for the LWP that incurred the event since the last reported event incurred by that thread. By default delta times are not displayed.
-e	Displays the environment strings which are passed in each exec system call.
-f	Follows all children created by the fork system call and includes their signals, faults, and system calls in the trace output. Normally, only the first-level command or process is traced. When the -f flag is specified, the process id is included with each line of trace output to show which process executed the system call or received the signal.
-i	Certain system calls on terminal devices or pipes, such as open and kread, can sleep for indefinite periods and are interruptible. Normally, truss reports such sleeping system calls if they remain asleep for more than one second. The system call is then reported a second time when it completes. The -i flag causes such system calls to be reported only once, upon completion.
-l	Display the id (thread id) of the responsible LWP process along with truss output. By default LWP id is not displayed in the output.
-m !Fault	Traces the machine faults in the process. Machine faults to trace must be separated from each other by a comma. Faults may be specified by name or number (see the sys/procfs.h header file). If the list begins with the "!" symbol, the specified faults are excluded from being traced and are not displayed with the trace output. The default is -mall -m!fltpage.
-o Outfile	Designates the file to be used for the trace output. By default, the output goes to standard error.
-p	Interprets the parameters to truss as a list of process ids for an existing process rather than as a command to be executed. truss takes control of each process and begins tracing it, provided that the user id and group id of the process match those of the user or that the user is a privileged user.
-r ! FileDescriptor	Displays the full contents of the I/O buffer for each read on any of the specified file descriptors. The output is formatted 32 bytes per line and shows each byte either as an ASCII character (preceded by one blank) or as a two-character C language escape sequence for control characters, such as horizontal tab (\t) and newline (\n). If ASCII interpretation is not possible, the byte is shown in two-character hexadecimal representation. The first 12 bytes of the I/O buffer for each traced read are shown, even in the absence of the -r flag. The default is -r!all.
-s ! Signal	Permits listing Signals to trace or exclude. Those signals specified in a list (separated by a comma) are traced. The trace output reports the receipt of each specified signal even if the signal is being ignored, but not blocked, by the process. Blocked signals are not received until the process releases them. Signals may be specified by name or number (see sys/signal.h). If the list begins with the "!" symbol, the listed signals are excluded from being displayed with the trace output. The default is -s all.
-t ! Syscall	Includes or excludes system calls from the trace process. System calls to be traced must be specified in a list and separated by

Option	Description
	commas. If the list begins with an "!" symbol, the specified system calls are excluded from the trace output. The default is -tall.
...:...FunctionName ...]	The FunctionName is a comma separated list of function names. In both cases the names can include name matching metacharacters *, ?, [] with the same meanings as interpreted by the shell but as applied to the library/function name spaces, and not to files.
	A leading ! on either list specifies an exclusion list of names of libraries or functions not to be traced. Excluding a library excludes all functions in that library. Any function list following a library exclusion list is ignored. Multiple -u options may be specified and they are honored left-to-right. By default no library/function calls are traced.
-w ! FileDescriptor	Displays the contents of the I/O buffer for each write on any of the listed file descriptors (see -r). The default is -w!all.
-x ! Syscall	Displays data from the specified parameters of traced sytem calls in raw format, usually hexadecimal, rather than symbolically. The default is -x!all.

9.10.1. Usage

```
truss [ -f] [ -c] [ -a] [ -l ] [ -d ] [ -D ] [ -e] [ -i]
 [ \{ -t | -x\} [!] Syscall [...] ] [ -s [!] Signal [...] ]
 [ \{ -m \}[!] Fault  [...]] [ \{ -r | -w\} [!] FileDescriptor [...] ]
 [ \{ -u \} [!]LibraryName [...]:: [!]FunctionName [ ... ] ] [ -o Outfile] \{Command| -p pid [. . .]\}
```

9.10.2. Output

In its simplest form, executing the truss command without any argument option for a given pocess ID or command (ex: the date command), would output the following:

```
bash$ truss date
execve("/usr/bin/date", 0xFFBFF5C4, 0xFFBFF5CC)  argc = 1
resolvepath("/usr/lib/ld.so.1", "/lib/ld.so.1", 1023) = 12
resolvepath("/usr/bin/date", "/usr/bin/date", 1023) = 13
stat("/usr/bin/date", 0xFFBFF3A0)               = 0
open("/var/ld/ld.config", O_RDONLY)             Err#2 ENOENT
stat("/nettools/subversion/lib/libc.so.1", 0xFFBFEE58) Err#2 ENOENT
stat("/opt/sybase/oc12.5.1-EBF12807/OCS-12_5/lib/libc.so.1", 0xFFBFEE58) Err#2 ENOENT
stat("/usr/local/java/jdk1.6.0_06/jre/lib/sparc/libc.so.1", 0xFFBFEE58) Err#2 ENOENT
stat("/nettools/sunstudio/sunstudio11/SUNWspro/lib/libc.so.1", 0xFFBFEE58) Err#2 ENOENT
stat("/usr/local/lib/libc.so.1", 0xFFBFEE58)    Err#2 ENOENT
stat("/usr/lib/libc.so.1", 0xFFBFEE58)          = 0
resolvepath("/usr/lib/libc.so.1", "/lib/libc.so.1", 1023) = 14
open("/usr/lib/libc.so.1", O_RDONLY)            = 3
mmap(0x00010000, 8192, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_ALIGN, 3, 0) = 0xFF3A0000
mmap(0x00010000, 1015808, PROT_NONE, MAP_PRIVATE|MAP_NORESERVE|MAP_ANON|MAP_ALIGN, -1, 0) = 0xFF280000
mmap(0xFF280000, 909301, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_TEXT, 3, 0) = 0xFF280000
mmap(0xFF36E000, 32017, PROT_READ|PROT_WRITE|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_INITDATA, 3, 909312) = 0xFF36E000
mmap(0xFF376000, 5984, PROT_READ|PROT_WRITE|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_ANON, -1, 0) = 0xFF376000
munmap(0xFF35E000, 65536)                       = 0
memcntl(0xFF280000, 144128, MC_ADVISE, MADV_WILLNEED, 0, 0) = 0
close(3)                                        = 0
mmap(0x00000000, 8192, PROT_READ|PROT_WRITE|PROT_EXEC, MAP_PRIVATE|MAP_ANON, -1, 0) = 0xFF390000
munmap(0xFF3A0000, 8192)                        = 0
mmap(0x00010000, 24576, PROT_READ|PROT_WRITE|PROT_EXEC, MAP_PRIVATE|MAP_ANON|MAP_ALIGN, -1, 0) = 0xFF3A0000
getcontext(0xFFBFF090)
getrlimit(RLIMIT_STACK, 0xFFBFF070)             = 0
getpid()                                        = 13054 [ 13053 ]
setustack(0xFF3A2088)
brk(0x00022D30)                                 = 0
brk(0x00024D30)                                 = 0
stat("/platform/SUNW,Sun-Fire-V490/lib/libc_psr.so.1", 0xFFBFED20) = 0
resolvepath("/platform/SUNW,Sun-Fire-V490/lib/libc_psr.so.1", "/platform/sun4u-us3/lib/libc_psr.so.1", 1023) = 37
open("/platform/SUNW,Sun-Fire-V490/lib/libc_psr.so.1", O_RDONLY) = 3
mmap(0x00010000, 8192, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_ALIGN, 3, 0) = 0xFF380000
close(3)                                        = 0
time()                                          = 1240919737
brk(0x00024D30)                                 = 0
brk(0x00026D30)                                 = 0
open("/usr/share/lib/zoneinfo/MET", O_RDONLY)   = 3
fstat64(3, 0xFFBFF360)                          = 0
read(3, " T Z i f\0\0\0\0\0\0\0\0"..., 755)     = 755
close(3)                                        = 0
ioctl(1, TCGETA, 0xFFBFF304)                    = 0
fstat64(1, 0xFFBFF220)                          = 0
```

```
Tue Apr 28 13:55:37 MEST 2009
write(1, " T u e   A p r   2 8   1"..., 30)      = 30
```

9.10.3. Output analysis

Each line of the trace output reports either the Fault or Signal name, or the Syscall name with parameters and return values. The subroutines defined in system libraries are not necessarily the exact system calls made to the kernel. The truss command does not report these subroutines, but rather, the underlying system calls they make. When possible, system call parameters are displayed symbolically using definitions from relevant system header files. By default, undefined system calls are displayed with their name, all eight possible argments and the return value in hexadecimal format. So the output will capture everything the given command or process is doing.

When an error is detected in the truss output (i.e. Err#2 ENOENT), the meaning of the error abbreviation can be checked by performing a grep command in the directory *usr/include/sys* which contains the header files that describe the error symbols. For example, use the following command to understand the meaning of the raised Err#2 ENOENT error:

```
bash$ grep ENOENT /usr/include/sys/*
/usr/include/sys/errno.h:#define        ENOENT  2       /* No such file or directory            */
```

Since the error was raised on a open command:
open("/var/ld/ld.config", O_RDONLY), this explains that an attempt to open a non-existent file raised this error.

Another useful tactic, especially on a very big truss log, is to perform a grep on open.This way, the files that a process is trying to open up will be seen.

When the process dies, it should report at the end of the truss output what the failure code is. Something like the following:

```
Incurred fault #6, FLTBOUNDS  %pc = 0xEF2617FC
    siginfo: SIGSEGV SEGV_MAPERR addr=0x00000000
  Received signal #11, SIGSEGV [caught]
    siginfo: SIGSEGV SEGV_MAPERR addr=0x00000000
```

You may be able to determine from the truss output alone what's wrong (it could show that the process was trying to access a certain file, or directory, but did not have access, for example). This would probably be listed a few lines above the error/fault section of the truss output.

When the -o flag is used with truss, or if standard error is redirected to a non-terminal file, truss ignores the hangup, interrupt, and signals processes. This facilitates the tracing of interactive programs which catch interrupt and quit signals from the
terminal. If the trace output remains directed to the terminal, or if existing processes are traced (using the -p flag), then truss responds to hangup, interrupt, and quit signals by releasing all traced processes and exiting. This enables the user to terminate excessive trace output and to release previously existing processes. Released processes continue to function normally.

For those options which take a list argument, the name all can be used as a shorthand to specify all possible members of the list. If the list begins with a !, the meaning of the option is negated (for example, exclude rather than trace). Multiple occurrences of the same option may be specified. For the same name in a list, subsequent options (those to the right) override previous ones (those to the left).

Every machine fault, with the exception of a page fault, results in posting a signal to the process which incurred the fault. A report of a received signal immediately follows each report of a machine fault, unless that signal is being blocked by the process.

10. JDB Debugger

The Java Debugger, jdb, is a simple command-line debugger for Java classes. It is a demonstration of the Java Platform Debugger Architecture that provides inspection and debugging of a local or remote Java Virtual Machine. That is, jdb helps in finding and fixing bugs in Java language programs.

10.1. Usage

JDB can be called from the command line as follows:

```
jdb [ options ] [ class ] [ arguments ]
```

options	Command-line options, as described below in the following subsection.
class	Name of the class to begin debugging.
arguments	Arguments passed to the main() method of class

10.1.1. Starting a jdb session

There are many ways to start a jdb session. The most frequently used way is to have jdb launch a new Java Virtual Machine (VM) with the main class of the application to be debugged. This is done by substituting the command jdb for java in the command line. For example, if your application's main class is MyClass, you use the following command to debug it under JDB:

```
% jdb MyClass
```

When started this way, jdb invokes a second Java VM with any specified parameters, loads the specified class, and stops the VM before executing that class's first instruction.

Another way to use jdb is by attaching it to a Java VM that is already running. A VM that is to be debugged with jdb must be started with the following options:

option	purpose

For example, the following command will run the MyClass application, and allow jdb to connect to it at a later time.

```
% java -Xdebug -Xrunjdwp:transport=dt_socket,address=8000,server=y,suspend=n MyClass
```

You can then attach jdb to the VM with the following commmand:

```
% jdb -attach 8000
```

Note that "MyClass" is not specified in the jdb command line in this case because jdb is connecting to an existing VM instead of launching a new one.

10.1.2. Basic jdb commands

The following is a list of the basic jdb commands. The Java debugger supports other commands which you can list using jdb's help command.

command	description
help, or ?	The most important jdb command, help displays the list of recognized commands with a brief description.
run	After starting jdb, and setting any necessary breakpoints, you can use this command to start the execution the debugged application. This command is available only when jdb launches the debugged application (as opposed to attaching to an existing VM).
cont	Continues execution of the debugged application after a breakpoint, exception, or step.
print	Displays Java objects and primitive values. For variables or fields of primitive types, the actual value is printed. For objects, a short description is printed. See the dump command below for getting more information about an object.
dump	For primitive values, this command is identical to print. For objects, it prints the current value of each field defined in the object. Static and instance fields are included. The dump command supports the same set of expressions as the print command.
threads	List the threads that are currently running. For each thread, its name and current status are printed, as well as an index that can be used for other commands, for example:" 4. (java.lang.Thread)0x1 main running" In this example, the thread index is 4, the thread is an instance of java.lang.Thread, the thread name is "main", and it is currently running,
thread	Select a thread to be the current thread. Many jdb commands are based on the setting of the current thread. The thread is specified with the thread index described in the threads command above.
where	where with no arguments dumps the stack of the current thread. where all dumps the stack of all threads in the current thread group. where threadindex dumps the stack of the specified thread. If the current thread is suspended (either through an event such as a breakpoint or through the suspend command), local variables and fields can be displayed with the print and dump commands. The up and down commands select which stack frame is current.
list	After setting a breakpoint on a specific line, list is used to show the corresponding line of the code on which the breakpoint was set and its surrounding lines.
pop	Pop the stack throught the current frame
step	Execute current line

10.1.3. Breakpoints

Breakpoints can be set in jdb at line numbers or at the first instruction of a method, for example:

- stop at MyClass:22 (sets a breakpoint at the first instruction for line 22 of the source file containing MyClass)
- stop in java.lang.String.length (sets a breakpoint at the beginning of the method java.lang.String.length)
- stop in MyClass.<init> (<init> identifies the MyClass constructor)
- stop in MyClass.<clinit> (<clinit> identifies the static initialization code for MyClass)

If a method is overloaded, you must also specify its argument types so that the proper method can be selected for a breakpoint. For example, "MyClass.myMethod(int,java.lang.String)", or "MyClass.myMethod()".

The clear command removes breakpoints using a syntax as in "clear MyClass:45". Using the clear or command with no argument displays a list of all breakpoints currently set. The cont command continues execution.

10.1.4. Stepping

The step commands advances execution to the next line whether it is in the current stack frame or a called method. The next command advances execution to the next line in the current stack frame.

10.1.5. Exceptions

When an exception occurs for which there isn't a catch statement anywhere in the throwing thread's call stack, the VM normally prints an exception trace and exits. When running under jdb, however, control returns to jdb at the offending throw. You can then use jdb to diagnose the cause of the exception.

Use the catch command to cause the debugged application to stop at other thrown exceptions, for example: "catch java.io.FileNotFoundException" or "catch mypackage.BigTroubleException. Any exception which is an instance of the specified class (or of a subclass) will stop the application at the point where it is thrown. Then use the pop command to pop the stack of execution.

The ignore command negates the effect of a previous catch command.

NOTE: The ignore command does not cause the debugged VM to ignore specific exceptions, only the debugger.

10.1.6. Command Line options

sourcepath <dir1:dir2:...>	Uses the given path in searching for source files in the specified path. If this option is not specified, the default path of "." is used.
-attach <address>	Attaches the debugger to previously running VM using the default connection mechanism.
-launch	Launches the debugged application immediately upon startup of jdb. This option removes the need for using the run command. The debuged application is launched and then stopped just before the initial application class is loaded. At that point you can set any necessary breakpoints and use the cont to continue execution.
-Joption	Pass option to the Java virtual machine, where option is one of the options described on the reference page for the java application launcher. For example, -J-Xms48m sets the startup memory to 48 megabytes.

11. Analysis

11.1. pinpoint a meaningful JAVA exception

The starting point of any JAVA failure analysis using Jdb is to find a meaningful JAVA exception/error that is thrown (In the logs folder for instance). For example, the following JAVA exception was thrown in the xmls log folder when trying to start the CACHE service from a monit session:

```
Caused by: java.lang.NullPointerException
        at java.util.Hashtable.put(Hashtable.java:399)
        at murex.apps.datalayer.server.cache.XmlCacheHome.putCacheUser(XmlCacheHome.java:232)
        at murex.apps.datalayer.server.cache.XmlCacheHome.initCacheUsers(XmlCacheHome.java:253)
        at murex.apps.datalayer.server.cache.XmlCacheHome.<init>(XmlCacheHome.java:79)
        at murex.apps.datalayer.server.cache.XmlCacheHome.create(XmlCacheHome.java:87)
        at murex.apps.datalayer.server.cache.home.MxXmlCacheHome.init(MxXmlCacheHome.java:40)
        at murex.apps.datalayer.server.cache.home.MxXmlCacheHome.xmlProcessStart(MxXmlCacheHome.java:66)
        at murex.apps.middleware.server.mx.AbstractMxHome.startService(AbstractMxHome.java:143)
        at murex.apps.middleware.server.core.service.ServiceServer.start(ServiceServer.java:78)
```

11.2. Start the jdb session

The next step to follow in the debugging process using JDB is to activate the jdb session as described in the previous section above.
As an illustration, the running example will show how to use jdb by attaching it to a JAVA VM that is already running. This can be done by starting the corresponding service (which represents the VM to be debugged with jdb) with the additional options -Xdebug and -Xrunjdwp:transport=dt_socket,server=y,suspend=n explained above. For instance, in our running example, since the problem is related to the CACHE service, the service to be launched with the aditional options is the mdcs service as follows:

```
launchmxj.app -mdcs -jopt:-Xdebug -jopt:-Xrunjdwp:transport=dt_socket,server=y,suspend=n,address=5466
```

11.3. Attach the debugger to the running JAVA VM

The debugger can now be attached to the running VM on the address number specified in the -Xrunjdwp option as follows:

```
jdb -attach 5466
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
Initializing jdb ...
```

11.4. Set the breakpoint

After attaching the running VM, the breakpoint can be set using one of the commands explained previously above. For instance, in our running example, we will be using the stop in command to set the breakpoint on the JAVA class `murex.apps.datalayer.server.cache.XmlCacheHome.putCacheUser` which was depicted in the initial excception.

```
> stop in murex.apps.datalayer.server.cache.XmlCacheHome.putCacheUser
Set breakpoint murex.apps.datalayer.server.cache.XmlCacheHome.putCacheUser
>
Breakpoint hit: "thread=Thread-11", murex.apps.datalayer.server.cache.XmlCacheHome.putCacheUser(), line=231 bci=0
```

11.5. Dump the stack of the thread

Use the variety of the where commands explained in the previous section above in order to dump the stack of the tread. For instance, in our runnning example, we will use the where command with no arguments to dump the stack of the current thread.

```
Thread-11[ 1 ] where
```

```
  [ 1 ] murex.apps.datalayer.server.cache.XmlCacheHome.putCacheUser (XmlCacheHome.java:231)
  [ 2 ] murex.apps.datalayer.server.cache.XmlCacheHome.initCacheUsers (XmlCacheHome.java:253)
  [ 5 ] murex.apps.datalayer.server.cache.home.MxXmlCacheHome.init (MxXmlCacheHome.java:40)
  [ 6 ] murex.apps.datalayer.server.cache.home.MxXmlCacheHome.xmlProcessStart (MxXmlCacheHome.java:66)
  [ 7 ] murex.apps.middleware.server.mx.AbstractMxHome.startService (AbstractMxHome.java:143)
  [ 8 ] murex.apps.middleware.server.core.service.ServiceServer.start (ServiceServer.java:78)
  [ 9 ] murex.apps.middleware.server.core.launcher.Service.start (Service.java:252)
  [ 10 ] murex.apps.middleware.server.core.launcher.LauncherHome.createService (LauncherHome.java:155)
  [ 11 ] sun.reflect.NativeMethodAccessorImpl.invoke0 (native method)
  [ 12 ] sun.reflect.NativeMethodAccessorImpl.invoke (NativeMethodAccessorImpl.java:39)
  [ 13 ] sun.reflect.DelegatingMethodAccessorImpl.invoke (DelegatingMethodAccessorImpl.java:25)
  [ 14 ] java.lang.reflect.Method.invoke (Method.java:597)
  [ 15 ] murex.apps.middleware.client.core.server.connection.engine.dispatcher.ReflectionDispatcher.call (ReflectionDispatcher.java:38)
  [ 16 ] murex.apps.middleware.client.core.server.connection.engine.dispatcher.ObjectContainer$ObjectDetail.call (ObjectContainer.java:153)
  [ 17 ] murex.apps.middleware.client.core.server.connection.engine.dispatcher.ObjectContainer.call (ObjectContainer.java:53)
  [ 18 ] murex.apps.middleware.client.core.server.connection.engine.AbstractServerConnection.startWork (AbstractServerConnection.java:252)
  [ 19 ] murex.apps.middleware.client.shared.thread.worker.Worker.run (Worker.java:97)
```

11.6. Display variables and fields

When the current thread is suspended either through a breakpoint event or through the suspend command described previously, local variable and fields can be displayed with the print and dump commands. For instance, in our running example, by checking the corresponding code of the function `murex.apps.datalayer.server.cache.XmlCacheHome.putCacheUser`, we depicted that the passed argument in the variable `cacheUser`.

```
private void putCacheUser(CacheUser cacheUser)\{
        cacheUsers.put(cacheUser.getUserIDDouble(),cacheUser);
        cacheUsersByName.put(cacheUser.getUserName(),cacheUser);
    \}
```

So the print command can be used to display the values of this variable.

```
Thread-11[ 1 ] print cacheUser
 cacheUser = "UserID[6.0,FIXING_PAGE,page de fixing,Public,false]"
```

11.7. Start or continue the execution of the debugged application

Use the run or cont command explained in the previous section above in order to start or continue the execution of the debugged application.
For instance:

```
Thread-11[ 1 ] run
>
Breakpoint hit: "thread=Thread-11", murex.apps.datalayer.server.cache.XmlCacheHome.putCacheUser(), line=231 bci=0
```

The display and execution commands can be interleaved until the crash occurs and the guilty variable is detected. For instance:

```
Thread-11[ 1 ] print cacheUser
 cacheUser = "UserID[4.0,INTERNAL.37,AS541,Private,true]"
Thread-11[ 1 ] run
>
Breakpoint hit: "thread=Thread-11", murex.apps.datalayer.server.cache.XmlCacheHome.putCacheUser(), line=231 bci=0

Thread-11[ 1 ] print cacheUser
 cacheUser = "UserID[15.0,INTERNAL.FIXING,null,Private,false]"
Thread-11[ 1 ] run
>
Breakpoint hit: "thread=Thread-11", murex.apps.datalayer.server.cache.XmlCacheHome.putCacheUser(), line=231 bci=0

Thread-11[ 1 ] print cacheUser
 cacheUser = "UserID[38.0,null,null,Private,false]"
Thread-11[ 1 ] run
> Input stream closed.
```

In our running example, one of the fields of the variable cacheUser has a null value and is the cause of the crash.

12. Profiling

The tool mainly used for profiling java services can be used in troubleshooting purposes. Yet the installation on servers is not obvious due to the fact that all launchers should have a dedicated port assigned to them and be launched with the jmx parameters.
Additional information can be found under:

http://java.sun.com:80/javase/6/docs/technotes/guides/visualvm/index.html
http://www.exist-db.org/jmx.html

12.2. Jconsole

Java monitoring and management console, used to monitor the JVM. The tool is mainly used to graph the parameters used by the java process; Heap/non-heap memory, CPU usage, threads, classes

The tool can also be used to dynamically change several parameters in the running system.

The jconsole requires the reflexionX and can launched from the following location: /usr/local/java/jdk1.6.0_06/bin/jconsole

Information on how to use the tool can be found under the following link:
http://java.sun.com/developer/technicalArticles/J2SE/jconsole.html

12.3. Memory Analyzer

MAT or Memory analyzer tool is an interesting tool to analyze java heap and helps you find memory leaks and reduce memory consumption.

The documentation below provides a summary about the tool and the basic knowledge needed to start using it; we recommend to refer to the official documentation for advanced troubleshooting.

12.3.1. Documentation

- All documentation available under this link and take the basic tutorial to get more familiar with the tool.
- Refer to this blog to find memory leaks in one click.

12.3.2. Generate the heap

With java we can easily get a full memory dump from a live Java process. Also it is possible to instruct the JVM to create automatically a heap dump in case that it runs out of memory.

- To get the full memory dump from a live Java process we can execute the following command: `JAVAPATH/bin/jmap -dump:live,format=b,file=<fileName>.bin PID_OF_SERVICE`

  Full GC Trigger
  Note that using dump:live option will trigger a full gc priority to dumping the heap - To retrieve the full heap remove the live option

- To get the full memory dump once the JVM is out of memory, we can add the following argument : `-XX:+HeapDumpOnOutOfMemoryError`

12.3.3. MAT Download

Once you have your heap dump file (fileName.bin), you need to install the standalone MAT for your machine. All MAT versions can be downloaded from this link.

Start using the windows version first (under citrix we have one under D:\) then switch to Linux if needed (make sure to have enough free memory i.e dump size + 10g at least)

The downloaded file is a zip file, all you need is to unzip it. You will have MAT folder with MemoryAnalyzer executable file under it.

12.3.4. MAT Configuration

MAT will be launched by default with -Xmx1024m. In case the memory dump file is big, and MAT is going OutOfMemory, you need to increase the Xmx. It could be done with 2 ways:

1. Modifying the Xmx under the file MemoryAnalyzer.ini.
2. Adding the following arguments to the launching command: -vmargs –Xmx4g

The Xmx should be usually grater than the size of the memory dump file.

MAT will be started using the default java on the machine. In case the default java on the machine is not the correct one and MAT should be started with different java version, you can add the below argument:

- -vm <path to the correct libjvm version>

12.3.5. Known errors

12.3.5.1. On windows

Error: Eclipse returns error message "Java was started but returned exit code = 1"

Solution: Set the java path under MemoryAnalyzer.ini

MemoryAnalyzer.ini

```
-startup
plugins/org.eclipse.equinox.launcher_1.5.0.v20180512-1130.jar
```

```
--launcher.library
plugins/org.eclipse.equinox.launcher.win32.win32.x86_64_1.1.700.v20180518-1200
```

```
D:\jdk1.8.0_91-x64\jre\bin\server\jvm.dll
-Xmx2048m
-vmargs
```

12.3.5.2. On Linux

Error 1:

```
bash$ MemoryAnalyzer
Java HotSpot(TM) Server VM warning: You have loaded library /data/apps/riad/mat/plugins/org.eclipse.equinox.launcher.gtk.linux.x86_64_1.1.700.v20180518-
The VM will try to fix the stack guard now.
It's highly recommended that you fix the library with 'execstack -c <libfile>', or link it with '-z noexecstack'.
Gtk-Message: GtkDialog mapped without a transient parent. This is discouraged.
```

Solution: Make sure to add the jvm library path of the 64bits java version in MemoryAnalyzer.ini

MemoryAnalyzier.ini

```
--launcher.library
plugins/org.eclipse.equinox.launcher.gtk.linux.x86_64_1.1.300.v20150602-1417
-vm
/usr/local/java/jdk1.8.0_131-x64/bin/java
-startup
plugins/org.eclipse.equinox.launcher_1.3.100.v20150511-1540.jar
-vmargs
-Xms2g
-Xmx8g
```

Error 2:

When launching the MAT tool from Linux, you get the following WARNINGS followed by the log file under which you have an error:

MemoryAnalyzier.ini

```
bash$ MemoryAnalyzer -vm /usr/local/java/jdk1.8.0_111-x64/bin/java
***WARNING: Gtk+ version too old (micro mismatch)
***WARNING: SWT requires GTK 2.24.0
***WARNING: Detected: 2.18.9
***WARNING: SWT requires Cairo 1.9.4 or newer
***WARNING: Detected: 1.8.8
MemoryAnalyzer:
An error has occurred. See the log file
/hp431srv1/apps/HeapDump/mat/workspace/.metadata/.log.
```

If you check the log file, you have the following error stack:

MemoryAnalyzier.ini

```
!ENTRY org.eclipse.osgi 4 0 2019-05-07 14:50:41.529
!MESSAGE Application error
!STACK 1
org.eclipse.swt.SWTError: No more handles
        at org.eclipse.swt.SWT.error(SWT.java:4578)
        at org.eclipse.swt.SWT.error(SWT.java:4467)
        at org.eclipse.swt.SWT.error(SWT.java:4438)
        at org.eclipse.swt.graphics.Image.init(Image.java:1394)
        at org.eclipse.swt.graphics.Image.<init>(Image.java:230)
        at org.eclipse.ui.internal.WorkbenchImages.declareImages(WorkbenchImages.java:316)
        at org.eclipse.ui.internal.WorkbenchImages.initializeImageRegistry(WorkbenchImages.java:521)
        at org.eclipse.ui.internal.WorkbenchImages.getImageRegistry(WorkbenchImages.java:494)
        at org.eclipse.ui.internal.Workbench$16.runWithException(Workbench.java:1642)
        at org.eclipse.ui.internal.StartupThreading$StartupRunnable.run(StartupThreading.java:32)
        at org.eclipse.swt.widgets.Synchronizer.syncExec(Synchronizer.java:233)
        at org.eclipse.ui.internal.UISynchronizer.syncExec(UISynchronizer.java:144)
```

```
        at org.eclipse.swt.widgets.Display.syncExec(Display.java:5831)
        at org.eclipse.ui.internal.StartupThreading.runWithoutExceptions(StartupThreading.java:95)
```

Therefore, the MAT tool fails to start because the gtk version (in the WARNING messages) are old, hence, not supported by this tool.

Solution: Try launching the tool from another Linux machine that has a new gtk. GTK version can be retrieved using the following command:

```
pkg-config --modversion <Package Name>
```

```
pkg-config --modversion gtk+-2.0
```

12.3.6. Getting Started

To start MAT from Linux you need to export the display. It can be exported using reflection X. Make sure to start reflection X from Citrix, then run the following command from UNIX side:

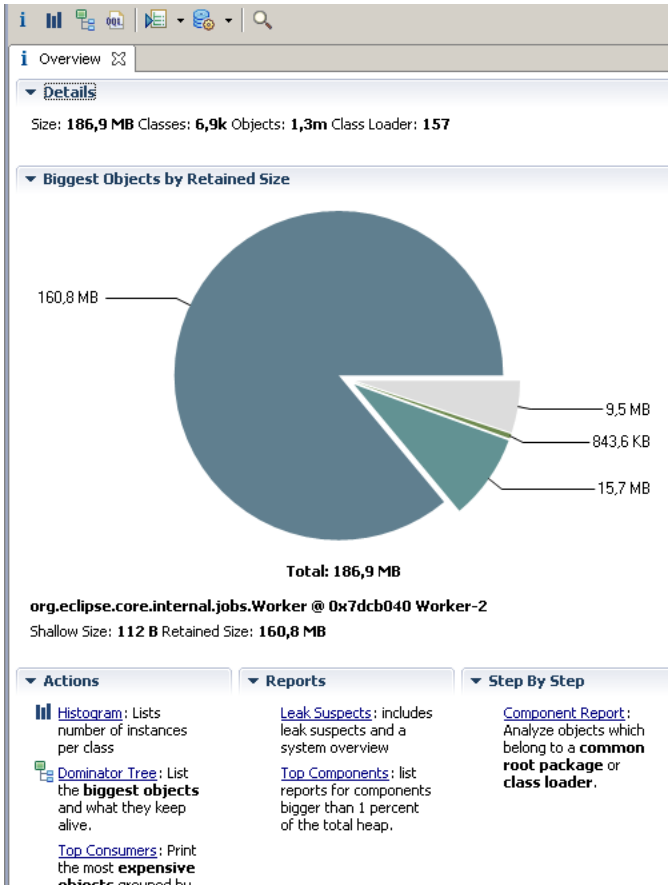
```
Export DISPLAY=<citrix ip address>:<display number>
```

Then you can start MemoryAnalzer from under MAT folder.

```
MemoryAnalyzer -vm <path to the correct libjvm version>/bin/java
```

`Click on File -> Open Heap Dump` and open the .bin file.

In the overview page, you'll find the size of the dump and the number of classes, objects and class loaders. Right below, the pie chart gives a global view on the biggest objects in the dump.



You have interesting things to go through like:

Histogram: lists the number of instances per class, the shallow size and the retained size. The histogram can also be filtered using a regular expression. For example, we can show only classes that match the pattern `murex.*`

Dominator tree:It displays the biggest objects in the heap dump; The dominator tree is a powerful tool to investigate which objects keep which other objects alive.

Leak Suspects: The Memory Analyzer can inspect the heap dump for leak suspects, e.g. objects or set of objects which are suspiciously big.

> Shallow vs. Retained Heap
> Shallow heap of an object is its size in the heap and retained size of the same object is the amount of heap memory that will be freed when the object is garbage collected.
>
> We recommend to understand the difference between Shallow and Retained heap to better understand how the tool operates.

12.3.7. Analyzing the leak

- Where the problem is
- The name of the class keeping the memory
- The component to which this class belongs
- How much memory is kept
- And where exactly the memory is accumulated.

Example:

OufOfMemory on PrintSrv launcher with Xmx 4G.

> One instance of **"murex.ui.model.export.print.GimExportEngine"** loaded by **"murex.rmi.loader.FileServerClassLoader @ 0x6eb96f5c8"** occupies **2,994,722,224 (98.27%)** bytes. The memory is accumulated in one instance of **"java.lang.Object[]"** loaded by **"<system class loader>"**.
>
> **Keywords**
> murex.ui.model.export.print.GimExportEngine
> murex.rmi.loader.FileServerClassLoader @ 0x6eb96f5c8
> java.lang.Object[]

In the above leak suspect report, we can see that one instance of class **GimExportEngine** occupies **98.27%** of heap size and having the memory accumulated under **java.lang.object[].**

Besides an overview of the leak suspects, the report contains detailed information about each of the suspects. You can display it by following the "**details**" link.

Two questions usually arise when a leak suspect is found:

- Why are the accumulated objects in memory? or Who is keeping them alive?
- Why is the suspect so big? What is its content?

First, you will find in the details the **shortest path** to the accumulation point;

Here you can see all the classes and fields through which the reference chain goes, and if you are familiar with the coding they should give you a good understanding how the objects are held.

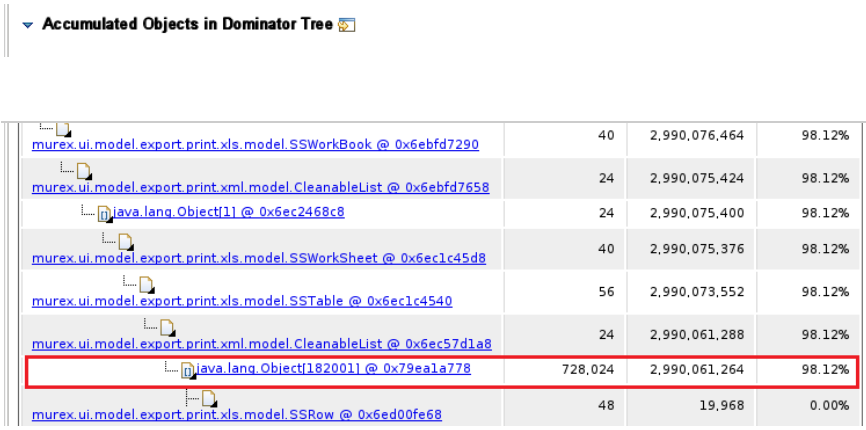
The object **GimExportEngine** is accumulating the memory.

Shortest Paths To the Accumulation Point

Class Name	Shallow Heap	Retained Heap
java.lang.Object[182001] @ 0x79ea1a778	728,024	2,990,061,264
elementData murex.ui.model.export.print.xml.model.CleanableList @ 0x6ec57d1a8	24	2,990,061,288
rows murex.ui.model.export.print.xls.model.SSTable @ 0x6ec1c4540	56	2,990,073,552
table murex.ui.model.export.print.xls.model.SSWorkSheet @ 0x6ec1c45d8	40	2,990,075,376
[0] java.lang.Object[1] @ 0x6ec2468c8	24	2,990,075,400
elementData murex.ui.model.export.print.xml.model.CleanableList @ 0x6ebfd7658	24	2,990,075,424
workSheets murex.ui.model.export.print.xls.model.SSWorkBook @ 0x6ebfd7290	40	2,990,076,464
root murex.ui.model.export.print.GimExportEngine @ 0x6ebfd7178	88	2,994,722,224
exportEngine murex.ui.print.PrintEngineDelegate @ 0x6ebfd7140	56	80
<Java Local> murex.shared.worker.Worker @ 0x74108e810 Thread-20 - Worker-3 Thread	**152**	**36,306,896**
printEngineDelegate murex.apps.datalayer.server.print.xml.PrintEngine @ 0x6ebfd7108 »	56	80
∑ Total: 2 entries		

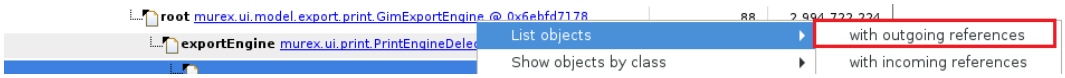
Then (to answer the question why is the suspect so big) the report contains some information about the **content** which was accumulated;

98.12% of the memory of the heap is consumed by java.lang.object items.



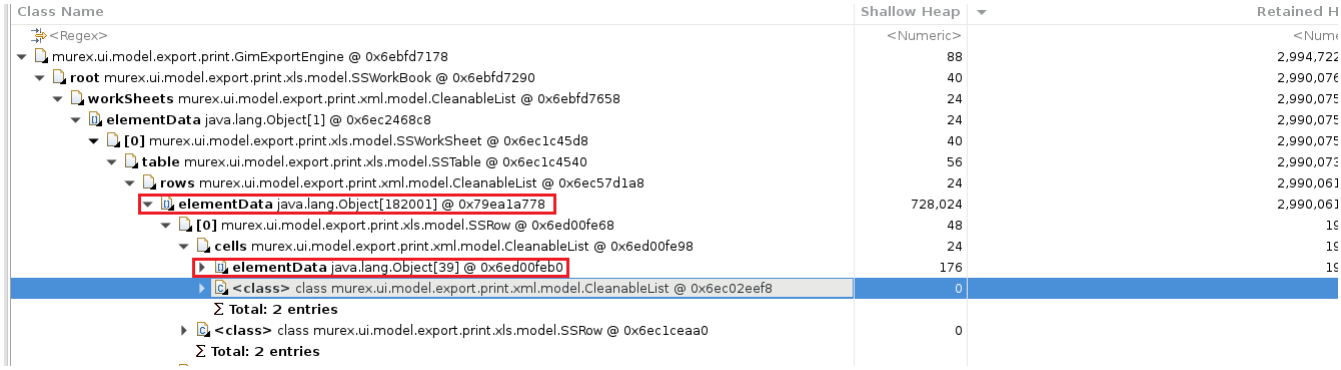
In order to better understand this accumulation, we need to list the outgoing references on this object **GimExportEngine;**

Therefore, we click on the link → select **List Objects** → with outgoing references; In this way we will be listing all the objects dependents from this object **GimExportEngine.**



Under the next page, we ill see that the memory was consumed by an excel worksheet; And the most consuming part is the rows (which is the data in this sheet) with 258289 rows.

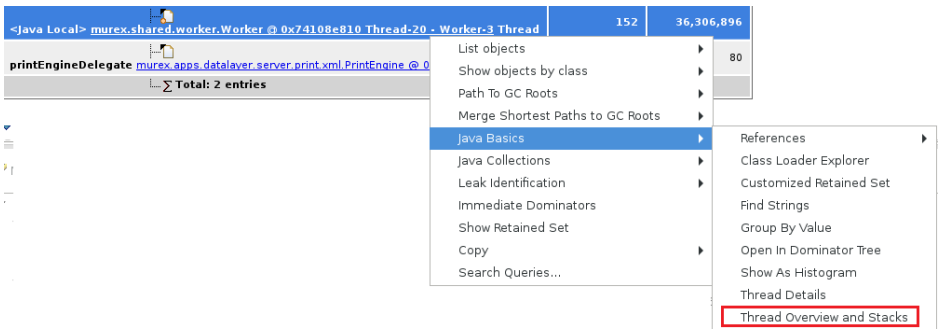
The objects holding the memory in side the **rows** are **cells** defined by **java.lang.object**.



In this way we were to identify that the memory is consumed by one worksheet holding **182001** rows. You can drill down in the tree to get more information about the worksheet name, and the data inside every cell.

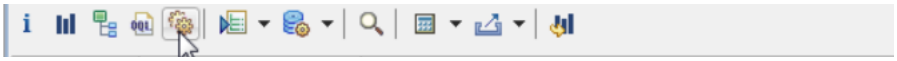
12.3.8. Analyzing threads

In order to get the stack (in case available) for a specific class object code, you click on the object in the accumulation tree → Java Basics → Thread Overview and Stacks:



In the stack trace, we can see where the OutOfMemory happened:

Or Alternatively to show all the threads dumps, use the Query Browser → Thread Overview and Stacks query:.



12.3.9. Querying Heap Objects (OQL)

Memory Analyzer allows to query the heap dump with custom SQL-like queries. OQL represents classes as tables, objects as rows, and fields as columns.

More information available under this link → Memory Analyzer → Tasks → **Querying Heap Objects.**

12.3.10. Analyze Unreachable Objects

> **IMPORTANT**
> In case the heap dump file loaded in MAT is not showing the correct memory size (usually equals to the size of the Xmx of the process), it might be due to unreachable objects available in the JVM.

By default unreachable objects are removed from the heap dump while parsing and will not appear in class histogram, dominator tree, etc.

Yet it is possible to open a histogram of unreachable objects. You can do it:

- From the link on the Overview page
- From the Query Browser via Java Basics → Unreachable Objects Histogram

This histogram has no object graph behind it(unreachable objects are removed during the parsing of the heap dump, only class names are stored). Thus it is not possible to see e.g. a list of references for a particular unreachable object.

But there is a possibility to keep unreachable objects while parsing. For this you need to either:

- Parse the heap dump from the command line providing the argument -keep_unreachable_objects, i.e. ParseHeapDump.bat -keep_unreachable_objects <heap dump>

or

- Set the preference using 'Window' > 'Preferences' > 'Memory Analyzer' > 'Keep Unreachable Objects', then parse the dump. Memory Analyzer version 1.1 and later has this preference page option to select keep_unreachable_objects.

12.4. TDA

Straightforward Thread dump analyzer that will display statistic on the threads running and help identify the highest consuming threads. Use the help section for additional information

12.5. Other tools

Jprofiler: intuitive GUI that helps finding performance bottlenecks, pin down memory leaks and resolve threading issues for java services.

Visualgc utility: same as jstat tool but graphical.

Sun Studio Performance Analyzer: analyse code for performance and deadlock situations. http://profiler.netbeans.org.

Dtrace: DTrace is a dynamic troubleshooting and analysis tool first introduced in the Solaris 10 and OpenSolaris operating systems to provide observability across the entire software stack. DTrace helps understand a software system by enabling to dynamically modify the operating system kernel and user processes to record additional data that you specify at locations of interest, called probes.

13. Other

13.1. JIT

JIT compilers generate dynamic optimizations; in other words, it makes optimization decisions while the Java application is running and generates high performing native machine instructions targeted for the underlying system architecture

`Illustration:`

According to most researches, 80% of execution time is spent in executing 20% of code. That would be great if there was a way to determine those 20% of code and to optimize them. That's exactly what JIT does - during runtime it gathers statistics, finds the "hot" code compiles it from JVM interpreted bytecode (that is stored in .class files) to a native code that is executed directly by Operating System and heavily optimizes it. Smallest compilation unit is single method. Compilation and statistics gathering is done in parallel to program execution by special threads. During statistics gathering the compiler makes hypotheses about code function and as the time passes tries to prove or to disprove them. If the hypothesis is dis-proven the code is deoptimized and recompiled again.

Most of the benchmarks show that JITed code runs 10 to 20 times faster than interpreted code.

`Client JIT Compiler:`

targets applications desiring rapid startup time and quick compilation so as to not introduce jitter in responsiveness such as client GUI applications.

`Server JIT Compiler:`

targets peak performance and high throughput for Java applications, so its design tends to focus on using the most powerful optimizations it can. This often means that compiles can require much more space or time than an equivalent compile by the Client JIT compiler. It tends to aggressively inline as well, which often leads to large methods, and larger methods take longer to compile

`JIT compiler threads`: These threads perform runtime compilation of bytecode to machine code.

13.2. JNI

Java Native Interface is a native programming interface. It allows Java code that runs inside a Java Virtual Machine to inter-operate with applications written in other programming languages, such as C, C++, and assembly language

13.3. VM runtime

standard command line options begin with a -X prefix
Developer command line options begin with a -XX prefix. + or - before the name of the options indicates a true or false value

- Add the following to the launcher arguments `-jopt:-HeapDumpOnOutOfMemoryError -jopt:-HeapDumpPath="javaheap dump.hprof"`

Which will generate a heap dump that can be analyzed in MAT (Eclipse Memory Analyzer) check section 12.3

- Fix memory leaks

No labels