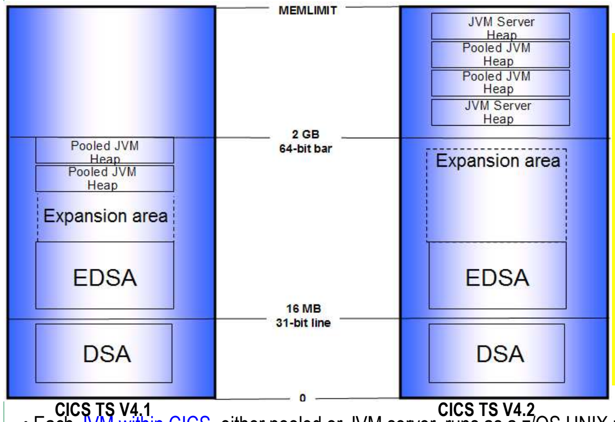
Cheat Sheet #86 zTidBits z/OS CICS' 64-bit Java

• CICS TS V4.2 uses the IBM® 64-bit SDK for z/OS, Java Technology Edition, Version 6.0.1. in *both* pooled JVMs and JVM server configurations.
NOTE: There is no support for using a 31-bit JVM with CICS TS V4.2.
• Java 6.0.1 includes enhancements to JIT optimization and exploitation of z196 hardware through the use of new hardware instructions and out-of-order pipeline execution.
NOTE: Java 6.0.1 is not distributed with CICS TS V4.2 but is available from http://www.ibm.com/systems/z/os/zos/tools/java/ at no charge.
• Using a 64-bit JVM provides significant virtual storage constraint relief by moving most of the memory used by both pooled JVMs and JVM server above the 64-bit bar as shown in the illustration.
• This allows a single CICS TS V4.2 region to support many more JVMs than earlier versions of CICS and each JVM can have much larger heaps. With CICS TS V4.2 it is quite possible to host traditional and Java workloads in a single CICS region in CICS TS V4.2 without encountering 'short-on-storage' conditions.

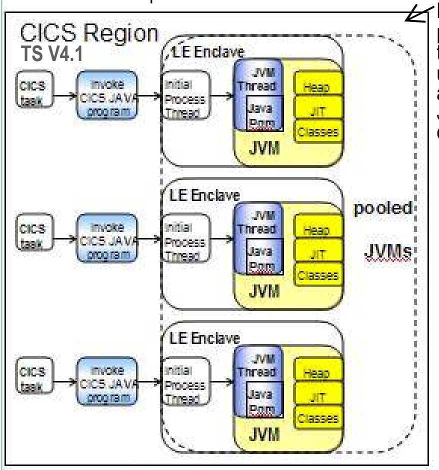


A potential disadvantage to using a 64-bit JVM is the associated increase in the size of Java objects needed to accommodate 64-bit object references. It is most likely that Java applications using 31-bit memory with CICS TS V4.1 and Java 6.0 will consume a larger amount of 64-bit memory if migrated to CICS TS V4.2 and Java 6.0.1.However this increase can be moderated by using the 'compressed references' feature of the IBM JVM that reduces the size of 64-bit Java objects.

IBM intends a future release to discon-tinue support for pooled JVMs and encourages customers to migrate to a JVM server configuration.

LE manages the Unix program and memory environment for z/OS. It provides a set of program stacks and memory heaps to support the execution of these programs the JVM runs as such a program within an LE enclave.

• Each JVM within CICS, either pooled or JVM server, runs as a z/OS UNIX process.
• An LE enclave provides the UNIX execution environment for such processes, managing the UNIX programs and memory.

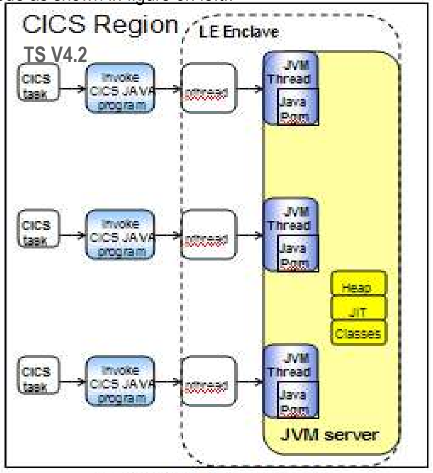


LE enclave structure for pooled JVMs

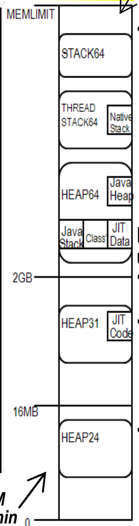
Prior to CICS TS V4.2, each CICS Java program ran in a reusable pooled JVM that could only support single-threaded use. Although pooled JVMs share some attributes, each has its own Java heap, Java class libraries and JIT-compiled code as shown in figure on left..

The CICS-supplied LE options load module DFHAXRO for JVM server sets an initial 64-bit heap of 100MB and initial 31-bit heap of 4MB. Both areas allow incremental expansion to increase these initial allocations.

CICS TS V4.2 introduced the JVM server configuration for general use. A JVM server allows multiple concurrent CICS tasks to run in a single multi-threaded JVM. These multiple requests "share" the same Java heap, class libraries and JIT-compiled code. JVM server provides the potential benefit of requiring less memory to run a workload, since heap, JITted code and Java classes are shared by concurrent users as shown in figure on right.



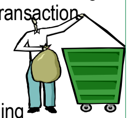
LE enclave structure for JVM server



64-bit JVM areas within an LE enclave

Choosing between pooled JVMs and JVM server: IBM found that CICS TS V4.2 using multiple pooled JVMs gives slightly better performance in terms of processor cost per transaction than a single multi-threaded JVM server. The JVM server does introduce a small amount of processor use to provide an OSGi framework and to manage multiple threads. But, a complex workload using many Java classes makes better use of a JVM server environment due to sharing functions.

•There are a number of GC policies supported by IBM Java 6.0.1. These GC policies can be specified in the jvmprofile used by both pooled JVM and JVM server in both CICS TS V4.1 and V4.2 but the default policies and CICS-supplied settings differ between CICS TS V4.1 and V4.2.
• CICS TS V4.1 supports Java 6.0 which has a default gcpolicy of *optthruput*. This treats the Java heap as a single space. This policy is optimized to deliver high throughput but it can produce occasional long pauses. These long pauses are usually avoided by CICS TS V4.1, which forces garbage collection by starting a CJGC transaction in each pooled JVM when it detects that the Java heap reaches a heap occupancy exceeding GC_HEAP_THRESHOLD.



• CICS TS V4.2 supports Java 6.0.1 which has a default policy of *gencon*. This treats the Java heap as separate generational areas. This policy is optimised to deliver lower pause times while maintaining high throughput. The support of concurrent threads by a JVM server makes the forced GC mechanism used for a pooled JVM undesirable since it may lead to a forced GC when other CICS tasks are still running Java applications
• GC in a JVM server is managed by the JVM whenever a GC event occurs (such as an allocation failure).
NOTE: You will not see any CJGC transactions running in a JVM server.
• The IBM Java Diagnostics Guide contains a comprehensive explanation of how the JVM manages memory and garbage collection. It is available at http://www.ibm.com/developerworks/java/jdk/diagnosis/ .

Which GC policy to use: The default GC policy for both Java 6.0.1 and CICS TS V4.2 is *gencon* but you are at liberty to use any of the GC policies and tuning parameters that are detailed in the Java Diagnostics Guide.
• The default GC policy for the previous versions of Java and CICS was *optthruput*.
- The significant difference between *optthruput* and *gencon* is that *optthruput* treats the Java heap as a single space whereas *gencon* splits the heap into an area to allocate new objects (a Nursery) and an area to keep long-lived objects (a Tenure space).
NOTE: Garbage collection of a single space heap is normally a less frequent but longer lasting process than garbage collecting a smaller nursery area. *gencon* is better suited to Java workloads that create many small short-lived objects or large long-lived objects, typical behaviour of a transactional workload.

Who pays to run a JVM in CICS: The CPU cost of starting a pooled JVM is paid by the user transaction that runs the first Java application in a JVM which triggers its creation.
•The cost of starting a JVM server is paid by a CICS-supplied CJSR transaction which is invoked when the JVM server resource is installed.
• Each pooled JVM or JVM server runs in its own LE enclave with its own 31-bit and 64-bit memory areas.
- The 64-bit JVM used by both pooled JVM and JVM server in CICS TS V4.2 still has a small requirement for some 31-bit memory but most of the storage used by the JVM, such as the Java heap, is located in 64-bit memory.
- The 31-bit JVM used in by CICS TS V4.1 needs to allocate memory from the same area of 31-bit memory used to allocate CICS EDSAs.
- This competition for the limited 31-bit memory space in a CICS region restricts the number of JVMs that a CICS TS V4.1 region can support. This limit is effectively removed in CICS TS V4.2.
• A 64-bit JVM used by CICS TS V4.2 will use more memory than its 31-bit counterpart in CICS TS V4.1, but this memory comes from the significantly less constrained 64-bit area.
- The amount of 64-bit memory available to CICS is restricted by the MEMLIMIT associated with the CICS address space.
- The 64-bit area is managed as memory objects and its use is reported by RMF.
> To be available for processing any memory (24-, 31- or 64-bit) needs to be allocated to real memory.
> Real memory is the physical memory belonging to a computer system and Real memory use is likely to become the restricting factor affecting 64-bit memory use since excessive demand for real memory will cause the z/OS system to page, degrading overall system performance.

NOTE: Storage Manager statistics in CICS TS V4.2 provide a measure of how many real memory frames have been used to support 64-bit memory objects by a CICS region.
• The main benefit to using JVM server with Java workload is that delivers similar throughput at similar CPU cost but uses significantly less memory than running multiple JVMPOOL JVMs.
• JVM server uses the shared class support provided by Java 6.0.1 directly.
- Unlike pooled JVMs there is no management interface (using SPI or CEMT commands for example) provided between CICS and the shared class cache.
NOTE: Class data sharing between JVMs is described in the IBM User Guide for Java 6 on z/OS (available at http://publib.boulder.ibm.com/infocenter/javasdk/v6r0/index.jsp).
• The potential benefits of using a shared class cache are:
☞ faster JVM start
☞ faster application start
☞ reduced memory consumption if using multiple JVMs

NOTE: CICS Java Server uses the Open Services Gateway initiative (OSGi) framework which is a module system and service platform for the Java programming language that implements a complete and dynamic component model.

Java Just-In-Time Compilation: For both pooled JVM and JVM server the JIT runs on a non-CICS TCB so any CPU consumed won't be charged to a CICS transaction.
- It will be included in the total CPU usage of the CICS address space.