IBM® Developer Kit and Runtime Environment,
Java™ 2 Technology Edition, Version 1.4.1, Service Refresh 1

IBM JVM
Garbage Collection and Storage Allocation techniques

Note

Before using this information and the product it supports, read the information in Appendix A.Notices

First Edition (November 2003)

This edition applies to all the platforms that are included in the IBM Developer Kit and Runtime Environment, Java 2 Technology Edition, Version 1.4.1, Service Refresh 1 and to all subsequent releases and modifications until otherwise indicated in new editions.

1 Introduction

This document describes the functions of the Storage (ST) component from release 1.2.2 to 1.4.1, Service Refresh 1.

The ST component allocates areas of storage in the heap. These areas of storage define objects, arrays, and classes. When an area of storage has been allocated, an object continues to be *live* while a reference (pointer) to it exists somewhere in the active state of the JVM; thus the object is *reachable*. When an object ceases to be referenced from the active state, it becomes *garbage* and can be reclaimed for reuse. When this reclamation occurs, the Garbage Collector must process a possible finalizer and also ensure that any monitor that is associated with the object is returned to the pool of available monitors (sometimes called the monitor cache). Not all objects are treated equally by the ST component. Some (ClassClass and Thread) are allocated into special regions of the heap (pinned clusters); others (Reference and its derivatives) are treated specially during tracing of the heap. More details on these special cases are given in section 4.4 “Reference Objects”.

1.1 Object allocation

Object allocation is driven by calls to one of the allocation interfaces; for example, stCacheAlloc, stAllocObject, stAllocArray, stAllocClass. These interfaces all allocate a given amount of storage from the heap, but have different parameters and semantics. The stCacheAlloc routine is specifically designed to deliver optimal allocation performance for small objects. Objects are allocated directly from a thread local allocation buffer that the thread has previously allocated from the heap. A new object is allocated from the end of this cache without the need to grab the heap lock; therefore, it is very efficient. Objects that are allocated through the stAllocObject and stAllocArray interfaces are, if small enough (currently 512 bytes), also allocated from the cache.

1.2 Reachable Objects

The active state of the JVM is made up of the set of stacks that represents the threads, the static’s that are inside Java classes, and the set of local and global JNI references. All functions that are invoked inside the JVM itself cause a frame on the C stack. This information is used to find the *roots*. These roots are then used to find references to other objects. This process is repeated until all reachable objects are found.

1.3 Garbage Collection

When the JVM cannot allocate an object from the current heap because of lack of space, the first task is to collect all the garbage that is in the heap. This process starts when any thread calls stGC either as a result of allocation failure, or by a specific call to System.gc(). The first step is to get all the locks that the garbage collection process needs. This step ensures that other threads are not suspended while they are holding critical locks. All the other threads are then suspended through an execution manager (XM) interface, which guarantees to make the suspended state of the thread accessible to the calling thread. This state is the top and bottom of the stack and the contents of the registers at the suspension point. It represents the state that is required to trace for object references. Garbage collection can then begin. It occurs in three phases:

- Mark
- Sweep
- Compaction (optional)

1.3.1 Mark Phase

In the mark phase, all the objects that are referenced from the thread stacks, static's, interned strings, and JNI references are identified. This action creates the root set of objects that the JVM references. Each of those objects might, in turn, reference others. Therefore, the second part of the process is to scan each object for other references that it makes. These two processes together generate a vector that defines live objects.

Each bit in the vector (allocbits) corresponds to an 8-byte section of the heap. The appropriate bit is set when an object is allocated. When the Garbage Collector traces the stacks, it first compares the pointer against the low and high limits of the heap. It then ensures that the pointer is pointing to an object that is on an 8-byte boundary (GRAIN) and that the appropriate allocbit is set to indicate that the pointer is actually pointing at an object. The Garbage Collector now sets a bit in the markbits vector to indicate that the object has been referenced.

Finally, the Garbage Collector scans the fields of the object to search for other object references that the object makes. This scan of the objects is done accurately because the method pointer that is stored in its first word enables the Garbage Collector to know the class of the object. The Garbage Collector therefore has access to a vector of offsets that the classloader builds at class linking time (before the creation of the first instance). The offsets vector gives the offset of fields that are in the object that contains object references.

1.3.2 Sweep Phase

After the mark phase, the markbits vector contains a bit for every reachable object that is in the heap. The markbits vector must be a subset of the allocbits vector. The task of the sweep phase is to identify the intersection of these vectors; that is, objects that have been allocated but are no longer referenced.

The original technique for this sweep phase was to start a scan at the bottom of the heap, and visit each object in turn. The length of each object was held in the word that immediately preceded it on the heap. At each object, the appropriate allocbit and markbit was tested to locate the garbage.

Now, the *bitsweep* technique avoids the need to scan the objects that are in the heap and therefore avoids the associated overhead cost for paging. In the bitsweep technique, the markbits vector is examined directly to look for long sequences of zeros (not marked), which probably identify free space. When such a long sequence is found, the length of the object that is at the start of the sequence is examined to determine the amount of free space that is to be released.

1.3.3 Compaction Phase

After the garbage has been removed from the heap, the Garbage Collector can compact the resulting set of objects to remove the spaces that are between them. Because compaction can take a long time, it is avoided if possible. Compaction, therefore, is a rare event. Compaction avoidance is explained in more detail later in section “4.3.1 Compaction Avoidance”.

The process of compaction is complicated because handles are no longer in the JVM. If any object is moved, the Garbage Collector must change all the references that exist to it. If one of those references was from a stack, and therefore the Garbage Collector is not sure that it was an object reference (it might have been a float, for example), the Garbage Collector cannot move the object. Such objects that are temporarily fixed in position are referred to as *dosed* in the code and have the dosed bit set in the header word to indicate this fact. Similarly, objects can be *pinned* during some JNI operations. Pinning has the same effect, but is permanent until the object is explicitly unpinned by JNI. Objects that remain mobile are compacted in two phases by taking advantage of the fact that the mptr is known to have the low three bits set to zero. One of these bits can therefore be used to denote the fact that it has been swapped. Note that this swapped bit is applied in two places: the link field (where it is known as OLINK_IsSwapped), and also the mptr (where it is known as GC_FirstSwapped). In both cases, the least significant bit (x01) is being set.

At the end of the compaction phase, the threads are restarted through an XM interface.

2 Data Areas

2.1 An Object

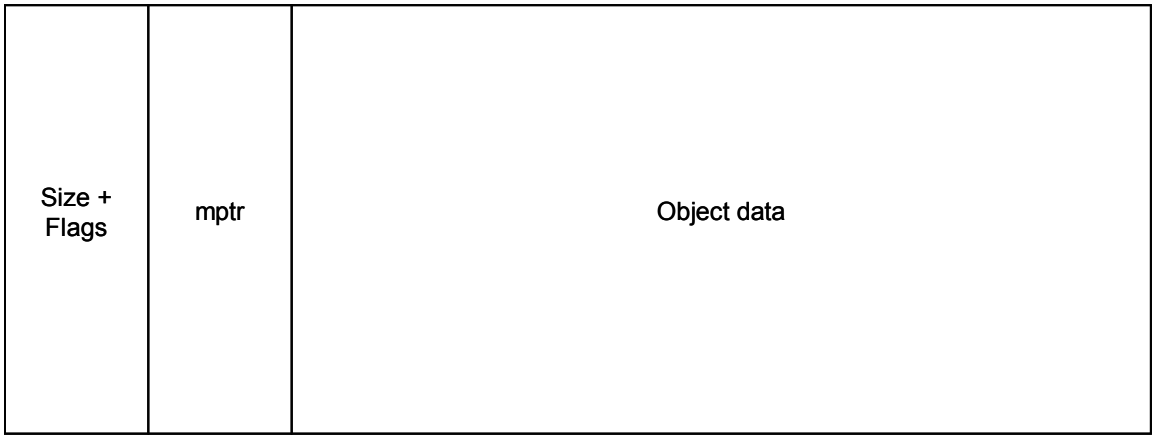

Figure 1. An object

Figure 1 shows the layout of an object on the heap.

- **size + flags**
  The size + flags slot is four bytes on 32-bit architecture and eight bytes on 64-bit architecture. The main purpose of this slot is to contain the length of the object. Because all objects start on an 8-byte boundary, and the size is divisible by 8, the bottom three bits are not used for the size, so the Garbage Collector uses them for some flags to indicate different states of the object. Also, because the size of objects is limited, the top two bits can be used for flags. (Note that the ***mptr*** slot, not the size + flags slot, is grained on an 8-byte boundary.).

  The flags that are in the size + flags slot are as follows:

  - Bit 1 has several purposes. It is the swapped bit, and is used during compaction (see section "4.3 Compaction Phase" for more information). Bit 1 is also the multipinned bit. It is used to indicate that this object has been pinned multiple times. During a garbage collection cycle, the multipinned bit is removed and restored to allow the other uses of this multipurpose bit.

  - Bit 2 is the dosed bit. The dosed bit is set on if the object is referenced from the stack or registers. "Referenced" means that the object cannot be moved in this garbage collection cycle because the Garbage Collector cannot fix up the reference because it might not be a real reference but an integer that happens to have the same value that an object on the heap has.

- Bit 3 is the pinned bit. Pinned objects cannot be moved, usually because they are referenced from outside the heap. Examples of this are Thread and ClassClass objects.

- Bit 31 in 32-bit architecture, or bit 63 in 64-bit architecture, is the flat locked contention (flc) bit and is used by the locking (LK) component.

- Bit 32 in 32-bit architecture, or bit 64 in 64-bit architecture, is the hashed bit and is used to denote an object that has returned its hashed value. This is required because the hash value is the address of the object and the Garbage Collector needs to maintain this if it moves the object.

- **mptr**
  The mptr slot is four bytes on 32-bit architecture and eight bytes on 64-bit architecture. The mptr slot is grained on an 8-byte boundary, not the size + flags. The mptr has one of two functions:

  1. If this is not an array, the mptr points to the method table, from where the Garbage Collector can get to the class block. In this way, the Garbage Collector can tell of what class an object is an instantiation. The method table and class block are allocated by the class loader (CL) component and are not in the heap.

  2. If this is an array, the mptr contains a count of how many array entries are in this object.

- **locknflags**
  The locknflags slot is four bytes on 32-bit architecture and eight bytes on 64-bit architecture, although only the lower four bytes are used. Its main use is to contain data for the LK component when locking. It also contains these flags:
  - Bit 2 is the array flag. If this bit is set on, the object is an array and the mptr field contains a count of how many elements are in the array.
  - Bit 3 is the hashed and moved bit. If this bit is set on, it indicates that this object has been moved after it was hashed, and that the hash value can be found in the last slot of the object.

- **Object data**
  This is where the object data starts, the layout of which is object dependent.

  The size + flags, mptr, and locknflags are sometimes known collectively as the header.

2.2 The Heap

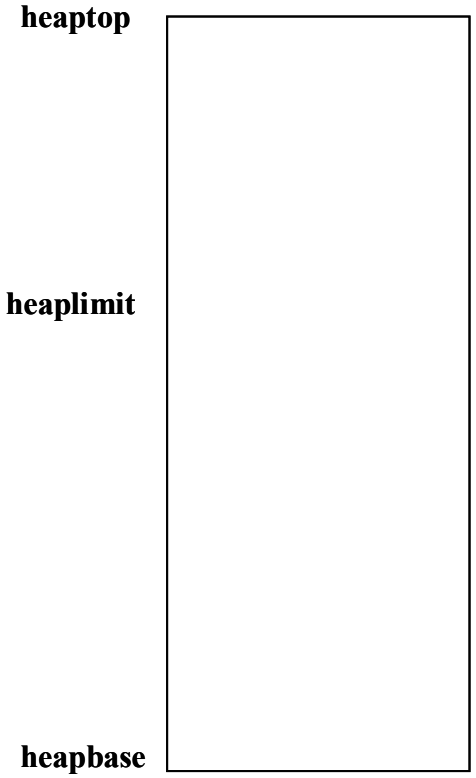

Figure 2. The heap

Figure 2 shows the layout of the heap. The heap is a contiguous area of storage that is obtained from the operating system at JVM initialization. heapbase is the address of the start of the heap and heaptop is the address of the end of the heap. heaplimit is the address of the top of the currently-used part of the heap. heaplimit can expand and shrink (see section "4.5 Heap Expansion" and section "4.6 Heap Shrinkage"). The -Xmx option controls the size from heapbase to heaptop. The -Xms option controls the initial size from heapbase to heaplimit. If these options are not specified they default to the following:

- **Xmx**
  - Windows: Half the real storage with a minimum of 16 MB and a maximum of 2 GB-1.
  - OS/390 and AIX: 64 MB.
  - Linux: Half the real storage with a minimum of 16 MB and a maximum of 512 MB-1.

- **Xms**
  - Windows, AIX, and Linux: 4 MB.
  - OS/390: 1 MB

2.2.1 Setting the heap size

For most applications, the default settings work well. The heap expands until it reaches a steady state, then remains in this state, which should give heap occupancy (that is, the amount of live data on the heap at any given time) of 70%. At this level, the frequency and pause time of garbage collection should be at an acceptable level.

For some applications, the default settings might not give the best results. Here are some problems that might occur, and some suggested actions that you can take. Use verbosegc to help monitor the heap.

- **The frequency of garbage collections is too high until the heap reaches a steady state:**

Use verbosegc to determine the size of the heap at a steady state, then set -Xms to this value.

- **The heap is fully expanded and the occupancy level is greater than 70%:**
  Increase the -Xmx value so that the heap is not more than 70% occupied. However, for best performance, ensure that the heap never pages. The maximum heap size should if possible be able to be contained in physical memory.
- **At 70% occupancy, the frequency of GCs is too great:**
  Change the setting of -Xminf. The default is 0.3, which tries to maintain 30% free space by expanding the heap. A setting of 0.4, for example, increases this free space target to 40%, thereby reducing the frequency of garbage collections.
- **Pause times are too long:**
  Try using -Xgcpolicy:optavgpause. It reduces the pause times and makes them more consistent as the heap occupancy rises. The cost is a drop in throughput which varies with applications, and will be approximately 5%.

Here are some tips that work well:

- Ensure that the heap never pages (that is, the maximum heap size must be able to be contained in physical memory).
- Avoid finalizers. You can never be guarantee when a finalizer will run. Often they cause problems. A verbosegc trace shows whether finalizers are being called. If you do use finalizers, try to follow these key points:
  - Avoid allocating objects in the finalizer method.
  - Do not use finalizers as a way to free native resources.
  - Avoid calling long or blocking routines from within a finalizer.
- Avoid compaction. A verbosegc trace shows whether compaction is occurring. Compaction is usually caused by requests for large memory allocations. Analyze requests for large memory allocations and avoid them if possible. If, for example, the memory allocations are large arrays, try to split them into smaller pieces.

2.3 Alloc bits and mark bits

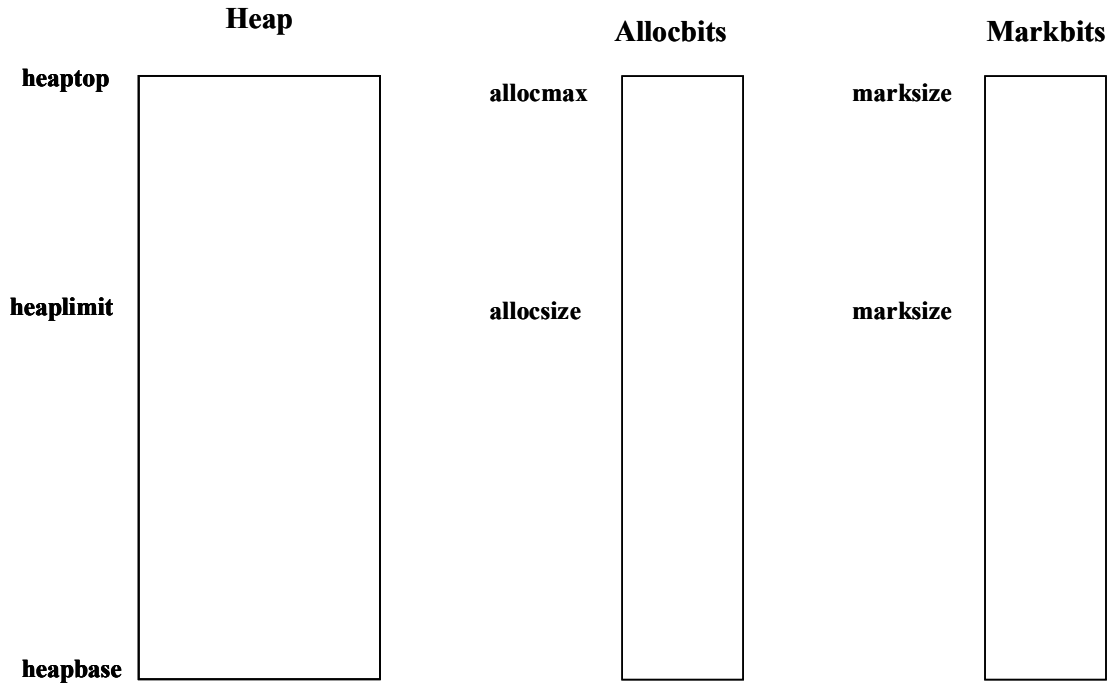

Figure 3. The heap with allocbits and markbits

Figure 3 shows the heap in relation to the allocbits and markbits bit vectors. These two bit vectors indicate the state of objects that are on the heap. Because all objects that are on the heap start on an 8-byte boundary, both vectors have one bit to represent eight bytes of the heap. Therefore, each of these vectors is $\frac{1}{64}$ of the heap size.

When objects are allocated in the heap, a bit is set on in allocbits to indicate the start of the object. This bit indicates where allocated objects are, but not whether the object is alive. During the mark phase, a bit is set on in markbits to indicate the start of a live object. Figure 4 shows two objects on the heap. The allocbit is set on for both objects.

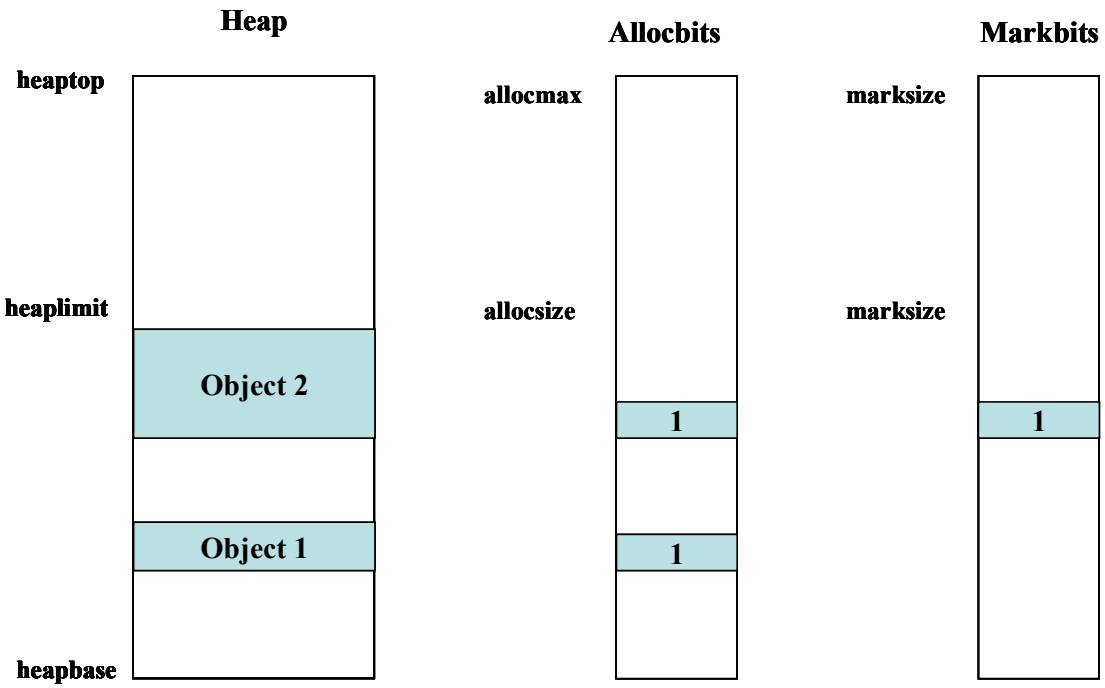

Figure 4. Some objects in the heap

During the mark phase, Object 2 was found to be referenced, but Object 1 was not. Therefore, a markbit is set on for Object 2. Object 1 will be collected during the sweep phase.

2.4 The System Heap

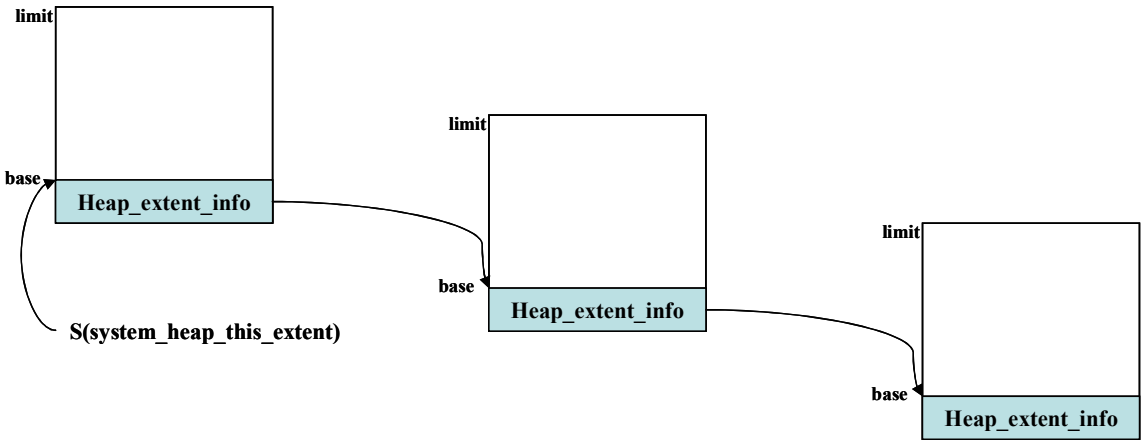

Figure 5. The system heap

The system heap contains only objects that have a life-expectancy of the life of the JVM. The objects that are in this heap are the class objects for system and shareable middleware and application classes. The Garbage Collector never collects the system heap because all objects that are in the heap are either reachable for the lifetime of the

JVM, or, in the case of shareable application classes, have been selected to be reused during the lifetime of the JVM. Figure 5 shows the layout of the system heap. The system heap is a chain of noncontiguous areas of storage. The initial size of the system heap is 128 KB in 32-bit architecture, and 8 MB in 64-bit architecture. If the system heap fills, it obtains another extent and chains the extents together.

2.5 The Free List

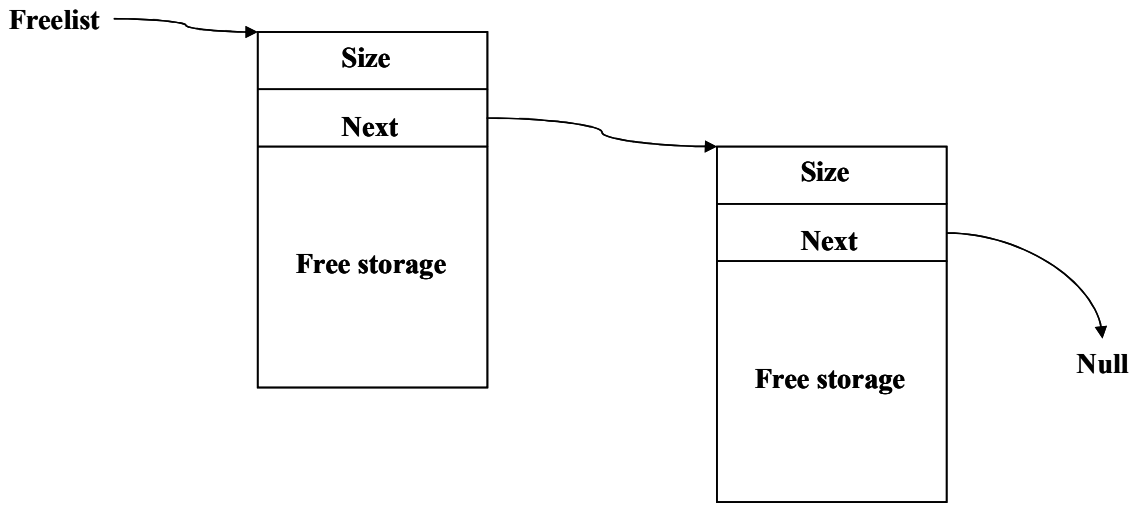

Figure 6. The Free List

Figure 6 shows the free list chain. The head of the list is in global storage and points to the first free chunk that is on the heap. Each chunk of free storage has a size field and a pointer that points to the next free chunk. The free chunks are in address sequence. The last free chunk has a NULL pointer.

3 Allocation

3.1 Heap lock allocation

Heap lock allocation occurs when the allocation request is greater than 512 bytes or the allocation cannot be contained in the existing cache (see section “3.2 Cache Allocation”). Heap lock allocation requires a lock, and is avoided if possible by using the cache instead.

```
If size < 512 or enough space in cache
        try cacheAlloc
        return if OK
HEAP_LOCK
Do
        If there is a big enough chunk on freelist
                takeit
                goto Got it
        else
                manageAllocFailure
                if any error
                        goto Get out
End Do
Got it:
Initialise object
Get out:
HEAP_UNLOCK
```

Figure 7. Heap Lock allocation

Figure 7 Heap Lock allocation shows some pseudo code for heap lock allocation. The Garbage Collector first checks the size of the allocation request. If the size is less than 512 bytes, or can be contained in the existing cache, the Garbage Collector tries to allocate by using cache allocation. If the Garbage Collector does not use cache allocation, or cache allocation failed to find free space, the HEAP_LOCK occurs. The Garbage Collector now searches the freelist for a chunk of free storage that is big enough to satisfy the allocation request. If it finds one, the Garbage Collector takes it and initializes the object, returning any remaining free storage to the freelist. Note that if the remaining free storage is less than 512 bytes plus the header size (12 bytes on 32-bit architecture, and 24 bytes on 64-bit architecture) it is not put onto the freelist. These small areas of storage are known as ‘dark matter’. They are recovered when the objects next to them become free or when the heap is compacted. If the Garbage Collector cannot find a big enough chunk of free storage an allocation failure occurs, and a garbage collection is performed. If the

Garbage Collection created enough free storage, it searches the freelist again and picks up the free chunk. If the Garbage Collection does not find enough free storage, it returns an out of memory condition. The HEAP_LOCK is released either after the object has been allocated, or if not enough free space is found.

3.1.1 Hints

In some conditions, for example, in large heaps where the freelist has many small free spaces, or in an application that is allocating many larger allocations, the heap lock allocation scheme has a problem. The problem is that because the scheme always starts at the beginning of the list, it has to search through most of the long list to find a freespace that is big enough to satisfy an allocation. The quick freelist hint algorithm was introduced to solve this problem.

For all heap lock allocation attempts that walk the freelist, the following data is collected:

- A search count of how many chunks on the freelist were examined before a freespace was found that was large enough to contain the desired allocation of size n.

- The size of the largest freespace chunk found in the freelist before the freespace that was used to satisfy the allocation request is also recorded. That is, the largest chunk that was not large enough to satisfy the request.

When a freespace that can satisfy the allocation is found, if the search count is larger than 20, it is desirable to create a new active hint that points into the freelist.

Active hints can now be used to start searching the freelist, at a point other than the beginning, depending on the size of the allocation request. Hints are dynamically updated when chunks are allocated from the freelist.

3.2 Cache Allocation

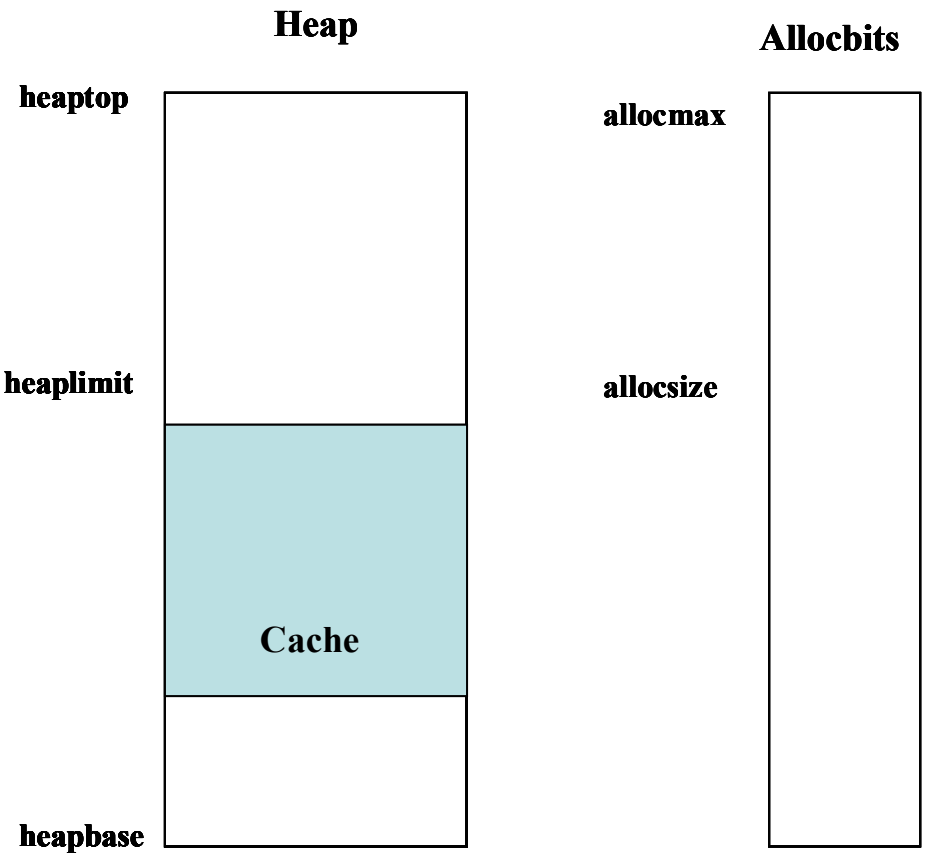

Figure 8. A cache block on the heap

Cache allocation is specifically designed to deliver the best possible allocation performance for small objects. Objects are allocated directly from a thread local allocation buffer that the thread has previously allocated from the heap. A new object is allocated from the end of this cache without the need to grab the heap lock; therefore, cache allocation is very efficient. The criterion for using cache allocation is:

- Use cache allocation if the size of the object is less than 512 bytes, or if the object can be contained in the current cache block.

Figure 8 shows a cache block on the heap. The cache block is sometimes called a thread local heap (TLH). When the Garbage Collector allocates a TLH for a thread, it goes through heap-locked allocation and reserves a part of the heap that will be used exclusively by a single thread. All cache allocation can then be made into the TLH without the need for any locks. Note that the allocbit is not set on for the TLH. The Garbage Collector sets allocbits bits for a TLH when the TLH is full or when a garbage collection cycle occurs. To increase the efficiency of allocating a TLH, the TLH allocator always takes the next free chunk on the free list up to a maximum size of 40 KB.

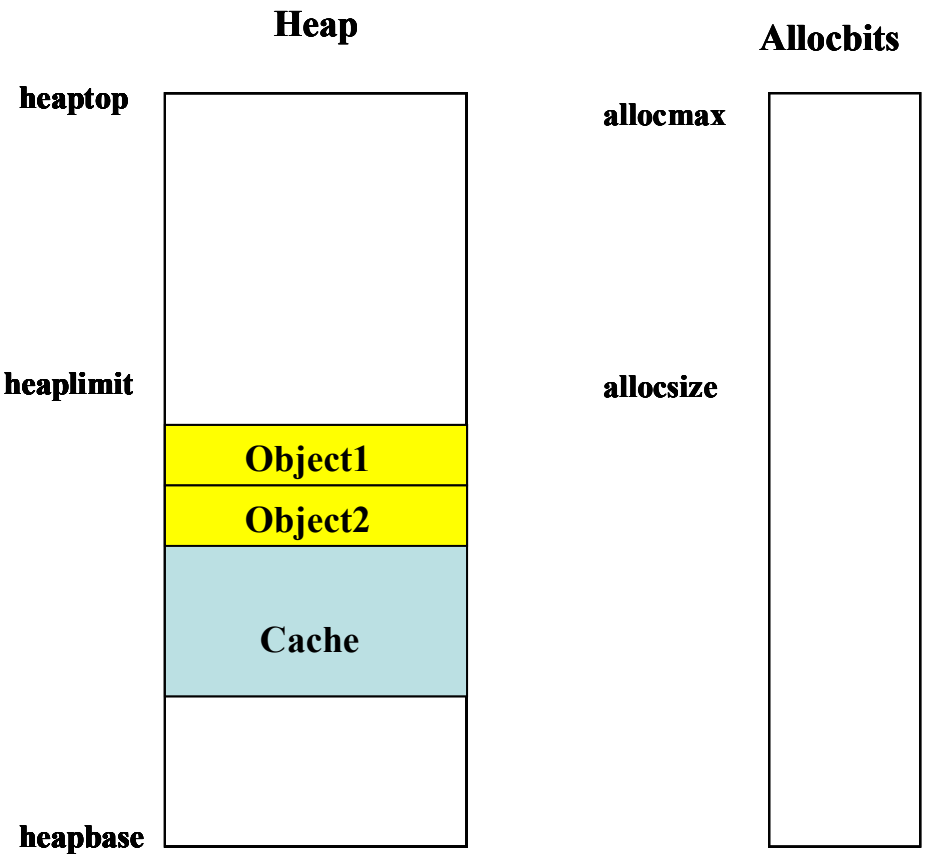

Figure 9. Objects allocated within cache block

Figure 9 shows some objects that have been allocated in the TLH. Here, objects are allocated from the back of the TLH. Objects can be allocated from the back of the TLH more efficiently than they can from the front. Figure 9 also shows that no allocbits have been set. They will be set for all objects in the TLH when the cache is full or when a garbage collection occurs.

4 Garbage Collection

Garbage collection is performed when an allocation failure occurs in heap lock allocation, or if a specific call to System.gc() occurs. The thread that has the allocation failure or the System.gc() call takes control and performs the garbage collection. It first gets all the locks that are required for a garbage collection, and then suspends all the other threads. Garbage collection then goes through the three phases: mark, sweep, and, optionally, compaction. The IBM Garbage Collector is known as a stop-the-world (STW) operation, because all application threads are stopped while the garbage is collected.

4.1 Mark Phase

In this phase, all the live objects are marked. Because unreachable objects cannot be identified singly, all the reachable objects must be identified. Therefore, everything else must be garbage. The process of marking all reachable objects is also known as tracing.

The active state of the JVM is made up of the saved registers for each thread, the set of stacks that represent the threads, the static's that are in Java classes, and the set of local and global JNI references. All functions that are invoked in the JVM itself cause a frame on the C stack. This frame might contain instances of objects as a result of either an assignment to a local variable, or a parameter that is sent from the caller. All these references are treated equally by the tracing routines. The Garbage Collector views the stack of a thread as a set of 4-byte fields (8 bytes in 64-bit architecture) and scans them from the top to the bottom of each of the stacks. The Garbage Collector assumes that the stacks are 4-byte aligned (8-byte aligned in 64-bit architecture). Each slot is examined to see whether it points at an object that is in the heap. Note that this does not make it necessarily a pointer to an object, because it might be only an accidental combination of bits in a float or integer. So, when the Garbage Collector performs the scan of a thread stack, it handles conservatively anything that it finds. Anything that points at an object is assumed to be an object, but the object in question must not be moved during garbage collection. A slot is thought to be a pointer to an object if it meets these three requirements:

1. It is grained (aligned) on an 8-byte boundary.
2. It is inside the bounds of the heap.
3. The allocbit is on.

Objects that are referenced in this way are known as roots, and have their dosed bit set on to indicate that they cannot be moved. The setting of dosed bits is done only if the Garbage Collector is to perform a compaction. Tracing can now proceed accurately. That is, the Garbage Collector can find references in the roots to other objects and, because it knows that they are real references, it can move them during compaction because it can change the reference. The tracing process uses a stack that can hold 4 KB entries. All references that are pushed to the stack are marked at the same time by setting the relevant markbit to on. The roots are marked and pushed to the stack and then the Garbage Collector starts to pop entries off the stack and trace them. Normal objects (not arrays) are traced by using the mtpr to access the classblock, which tells where references to

other objects are to be found in this object. As each reference is found, if it is not already marked, it is marked and pushed.

Array objects are traced by looking at each array entry and, if it is not already marked, it is marked and pushed. Some additional code traces a small portion of the array at a time, to try to avoid mark stack overflow.

The above process continues repeatedly until the mark stack eventually becomes empty.

4.1.1 Mark stack overflow

Because the mark stack has a fixed size, it can overflow. Although mark stack overflow is a rare event, it has a negative impact on pause time when it occurs.

4.1.1.1 The overflow set

To remember the locations of untraced objects the Garbage Collector needs a bit array that maps the whole heap. The FR_bits array, which is used for Incremental Compaction (IC), does this with one bit for every possible reference slot in the heap (that is, one bit for every four bytes on 32-bit platforms, and one bit for every eight bytes on 64-bit platforms). Since the JVMObject header cannot contain any object references we know that the first two bits in the FR_bits array for a given object are never used by IC. The Garbage Collector can, therefore, use the first of these 'spare' bits in the FR_bits array to implement the Overflow set.

4.1.1.2 Handling mark stack overflow for non-system heap objects

When a thread tries to push a reference onto the mark stack and finds that the marks stack is full it will try to publish work to its local mark queue. If the publish fails then the thread will set the bit in the FR_bits bit array which corresponds to the object referenced by the reference being pushed and set a global flag to indicate an overflow has occurred.

Tracing can then continue and all other references which cannot be pushed have the associated bit in the FR_bits bit array set.

Once a thread has exhausted its mark stack it then tries to take control of the overflow set. This is done by setting the global flag indicating an overflow has occurred to False which guarantees sole ownership of the overflow set. Once ownership has been established the thread scans the bit array and for any non-zero bit. When such a bit is found it is cleared and the corresponding reference is pushed onto the markstack. Once a sufficient number of references have been pushed onto the mark stack they are published to the local mark queue. This allows other threads to assist with the processing of the overflow set.

It is possible that a mark stack overflow could occur whilst processing the overflow set, if this happens then the global flag is set to indicate an overflow has occurred and the process above is repeated.

4.1.1.3 The System heap overflow mechanism

When collecting the root set, the Garbage Collector pushes onto mark stacks the address of all classes that are in the system and ACS heaps. So, a mark stack overflow might occur. However, the FR_bits array maps only the non-system heaps, and therefore, it cannot be used to remember untraced objects in system and ACS heaps.

The list of loaded classes is not modified during the mark phase of a garbage collection cycle. Therefore when mark stack overflow occurs all the Garbage Collector needs to remember is the place that it reached in the chain before the mark stack overflow occurred. The Garbage Collector uses two new global variables for this purpose: ‘overflowSystemClasses’ and ‘overflowACSClasses’ for system and ACS heaps respectively. When the Garbage Collector does mark stack overflow processing, these variables tell it where it stopped.

4.1.1.4 Handling mark stack overflow for system heap objects

Once a thread has exhausted its mark stack during parallelMark it checks to see if either overflowSystemClasses or overflowACSClasses is set. If one of the values is set then the thread will attempt to get control of the associated list by setting the value back to NULL. Once a thread has sole control of the list it processes it as before, pushing the references onto the mark stack and once sufficient references have been pushed, publishing the work on the local mark queue allowing other threads to assist.

If a mark stack overflow occurs whilst processing the overflow list then the thread simply sets the relevant flag to point at where it had got to and repeats the process above.

4.1.2 Parallel Mark

With Bitwise Sweep and Compaction Avoidance, the majority of garbage collection time is spent marking objects. Therefore, a parallel version of Garbage Collector Mark has been developed. The goal of Parallel Mark is to not degrade performance on a uniprocessor machine and to increase typical mark performance fourfold on an 8-way machine.

The time spent marking objects is decreased through the addition of helper threads and a facility that shares work between those threads. A single application thread is used as the master coordinating thread, often known as the main gc thread. This thread has the responsibility for scanning C-stacks to identify root pointers for the collection. A platform with N processors also has N-1 new helper threads that work with the master thread to complete the marking phase of garbage collection. The default number of threads can be overridden with the **-Xgcthreadsn** parameter. A value of 1 results in no helper threads. Values of 1 through N are accepted.

At a high level, each marker thread is provided with a local stack and a sharable queue, both of which contain references to objects that are marked but not yet scanned. Threads do most of the marking work by using their local stacks, synchronizing on sharable queues only when work balance requires it. Mark bits are updated by using atomic primitives that require no additional lock.

Because each thread has a Mark Stack that can hold 4 KB entries and a Mark Queue that can hold 2 KB entries, the chances of a Mark Stack Overflow are reduced.

4.1.3 Concurrent Mark

Concurrent mark gives reduced garbage collection pause times when heap sizes increase. It starts a concurrent marking phase before the heap is full. In the concurrent phase, the Garbage Collector scans the roots by asking each thread to scan its own stack. These roots are then used to trace live objects concurrently. Tracing is done by a low-priority background thread and by each application thread when it does a heap lock allocation.

While the Garbage Collector is marking live objects concurrently with application threads running, it has to record any changes to objects that are already traced. To do this, it uses a write barrier that is activated every time a reference in an object is updated. The heap is divided into 512-byte sections and each section is allocated a byte in the card table. Whenever a reference to an object is updated, the card that corresponds to the start address of the object that has been updated with the new object reference is marked with 0x01. A byte is used instead of a bit for two reasons: a write to a byte is quicker than a bit change, and the other bits are reserved for future use.

A STW (Stop The World) collection is started when one of the following occurs:

- An allocation failure
- A System.gc
- Concurrent mark completes all the marking that it can do

The Garbage Collector tries to start the concurrent mark phase so that it completes at the same time that the heap is exhausted. The Garbage Collector does this by constant tuning of the parameters that govern the concurrent mark time.

In the STW phase, the Garbage Collector scans all roots, uses the marked cards to see what must be retraced, and then sweeps as normal. It is guaranteed that all objects that were unreachable at the start of the concurrent phase are collected. It is not guaranteed that objects that become unreachable during the concurrent phase are collected.

Reduced pause times are the benefit of concurrent mark but they come at a cost. Application threads must do some tracing when they are requesting a heap lock allocation. The overhead varies depending on how much idle CPU time is available for the background thread. Also, the write barrier has an overhead.

This parameter enables concurrent mark:

-Xgcpolicy: <*optthruput*|*optavgpause*>

Setting **-Xgcpolicy** to *optthruput* disables concurrent mark. If you do not have pause time problems (as seen by erratic application response times), you get the best throughput with this option. *Optthruput* is the default setting.

Setting **-Xgcpolicy** to *optavgpause* enables concurrent mark with its default values. If you are having problems with erratic application response times that are caused by normal garbage collections, you can reduce those problems at the cost of some throughput by using the optavgpause option.

4.2 Sweep Phase

After the mark phase, the markbits vector contains a bit for every reachable object that is in the heap, and must be a subset of the allocbits vector. The sweep phase identifies the intersection of the allocbits and markbits vectors; that is, objects that have been allocated but are no longer referenced.

In the bitsweep technique, the Garbage Collector examines the markbits vector directly and looks for long sequences of zeros, which probably identify free space. When such a long sequence is found, the Garbage Collector checks the length of the object at the start of the sequence to determine the amount of free space that is to be released. If this amount of free space is greater than 512 bytes plus the header size, this free chunk is put on the freelist.

The small areas of storage that are not on the freelist are known as "dark matter", and they are recovered when the objects that are next to them become free, or when the heap is compacted. It is not necessary to free the individual objects in the free chunk, because it is known that the whole chunk is free storage. When a chunk is freed, the Garbage Collector has no knowledge of the objects that were in it. During this process, the markbits are copied to the allocbits so that on completion, the allocbits correctly represent the allocated objects that are on the heap.

4.2.1 Parallel Bitwise Sweep

Parallel Bitwise Sweep improves sweep time by using all available processors. In Parallel Bitwise Sweep, the Garbage Collector uses the same helper threads that are used in Parallel Mark, so the default number of helper threads is also the same and can be changed with the **-Xgcthreadsn** parameter. The heap is divided into sections. The number of sections is significantly larger than the number of helper threads. The calculation for the number of sections is as follows:

- 32 x the number of helper threads, or
- The maximum heap size ÷ 16 MB

Whichever is larger. The helper threads take a section at a time and scan it, performing a modified bitwise sweep. The results of this scan are stored for each section. When all sections have been scanned, the freelist is built.

4.3 Compaction Phase

After the garbage has been removed from the heap, the Garbage Collector can compact the resulting set of objects to remove the spaces that are between them. The process of compaction is complicated because, if any object is moved, the Garbage Collector must change all the references that exist to it. If one of those references was from a stack, and therefore the Garbage Collector is not sure that it was an object reference (it might have been a float, for example), the Garbage Collector cannot move the object. Such objects that are temporarily fixed in position are referred to as *dosed* in the code, and have the dosed bit set in the header word to indicate this fact. Similarly, objects can be *pinned* during some JNI operations. Pinning has the same effect, but is permanent until the object is explicitly unpinned by JNI. Objects that remain mobile are compacted in two phases by taking advantage of the fact that the mptr is known to have the low three bits set to zero. One of these bits can therefore be used to denote the fact that it has been swapped. Note that this swapped bit is applied in two places: the size + flags field (where it is known as OLINK_IsSwapped) and also the mptr (where it is known as GC_FirstSwapped). In both cases, the least significant bit (x01) is being set.

The following analogy might help you understand the compaction process.

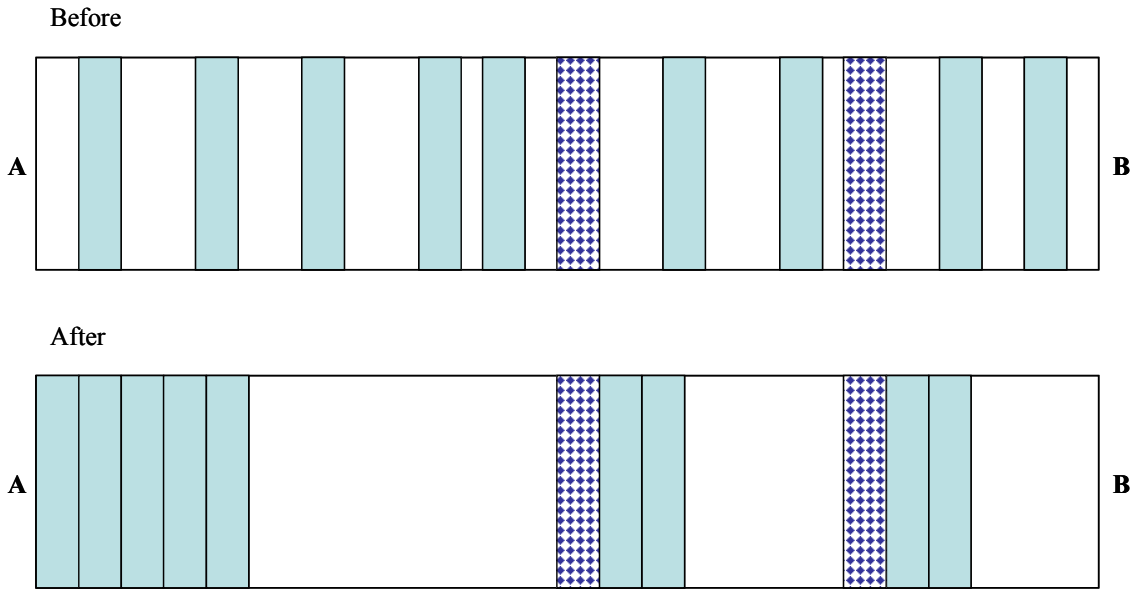

Figure 10. Compaction at work

Figure 10 shows the effects of compaction. If you imagine a corridor runs from A to B and contains several pieces of furniture (solid shapes), which represent objects, and gaps (clear shapes), which represent free space and dark matter. A couple of pieces of furniture have been nailed to the floor (Diamond checked shapes), which represent pinned or dosed objects. The goal of compaction is to move all the furniture to end A of the corridor. It does this by taking each piece of furniture from the left in turn and pushing it as far to the

end as possible. Unfortunately furniture cannot be lifted over the pieces which had been nailed down so the pieces to the right of these can only be moved to adjoin them.

4.3.1Compaction Avoidance

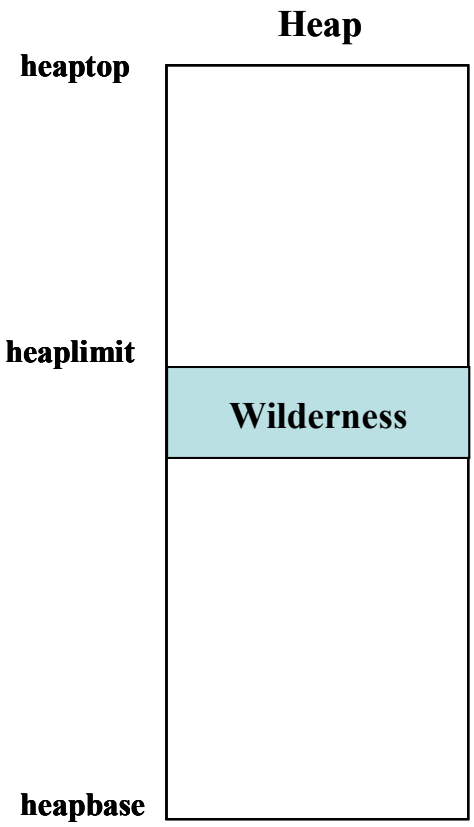

Figure 11. The wilderness

Compaction avoidance focuses on correct object placement. It therefore reduces, and in many cases removes, the need for compaction. An important point of this approach is a concept that is called wilderness preservation. Wilderness preservation attempts to keep a region of the heap in an unused state by focusing allocation activity elsewhere. It does this by making a boundary between most of the heap and a reserved wilderness portion. In typical cases, non-compacting garbage collection events are triggered whenever the wilderness is threatened. The wilderness is consumed (eroded) only when necessary to satisfy a large allocation, or when not enough allocation progress has been made since the previous garbage collection.

Figure 11 shows the wilderness on the heap. The wilderness is allocated at the end of the active part of the heap. Its initial size is 5% of the active part of the heap, and it expands and shrinks depending on usage. On heap lock allocation failure, if enough allocation progress has been made since the last garbage collection, and the size of the allocation request is less than 64 KB, the Garbage Collector runs. Enough progress means that at least 30% of the heap has been allocated since the last garbage collection. This is the default. It can be changed with the **-Xminf** parameter. If not enough progress has been made, or if the size of the allocation request is equal to or greater than 64 KB, the allocation is immediately satisfied from the wilderness if possible. Otherwise, a normal allocation failure occurs. Not enough progress has been made if the Garbage Collector gets an allocation request for a large object that cannot be satisfied before the free list is exhausted. In this condition, the reserved wilderness can satisfy the request, and avoid a garbage collection and a compaction.

Compaction occurs if any one of the following is true and **-Xnocompactgc** has not been specified:

- **-Xcompactgc** has been specified.
- Following the sweep phase, not enough free space is available to satisfy the allocation request.
- A System.gc() has been requested, and the last compaction occurred before the last allocation failure or concurrent mark collection.
- At least half the previously available memory has been consumed by TLH allocations (ensuring an accurate sample) and the average TLH size falls below 1000 bytes.
- Less than 5% of the active heap is free.
- Less than 128 KB of the active heap is free.

4.3.2 Incremental Compaction

4.3.2.1 Introduction

When objects are freed by garbage collection, the heap becomes fragmented. This fragmentation can cause a state in which enough free space is still available in the heap, but the free space is not contiguous, so it cannot be used for further object allocations.

Compaction defragments the Java heap. It is a process of moving scattered chunks of allocated spaces in the heap to one end of the heap, thereby creating a large, contiguous space at the other end. However the process of compaction can cause a considerable increase in the pause time of a garbage collection cycle, pause times of 40 seconds are quite possible for the compaction of a 1 GB heap. Such long pause times are often unacceptable for real-world applications. Incremental compaction is a way of spreading compaction work across garbage collection cycles, thereby reducing pause times.

Another important task for Incremental Compaction is the removal of dark matter. Dark matter is the term for small pieces of free space (currently less than 512 bytes in size) that are not on the free list and therefore are not available for allocation of objects. The level of the dark matter that is in the heap directly affects application throughput, because more dark matter means that less free space is available on the heap for object allocation, and less free space means that more garbage collection cycles will occur, having a serious affect on application performance. Such pieces might be scattered throughout the heap and might occupy a surprisingly large fraction of the total heap size.

4.3.2.2 Overview of Incremental Compaction

In incremental compaction, the Garbage Collector splits the heap into sections and compacts each section in the same way in which it does a full compaction. That is, the Garbage Collector moves all the moveable objects down the heap. This action retrieves all the dark matter and leaves large areas of free space. Individual sections on which incremental compaction runs are of fixed size, and therefore constrains the time required for compaction.

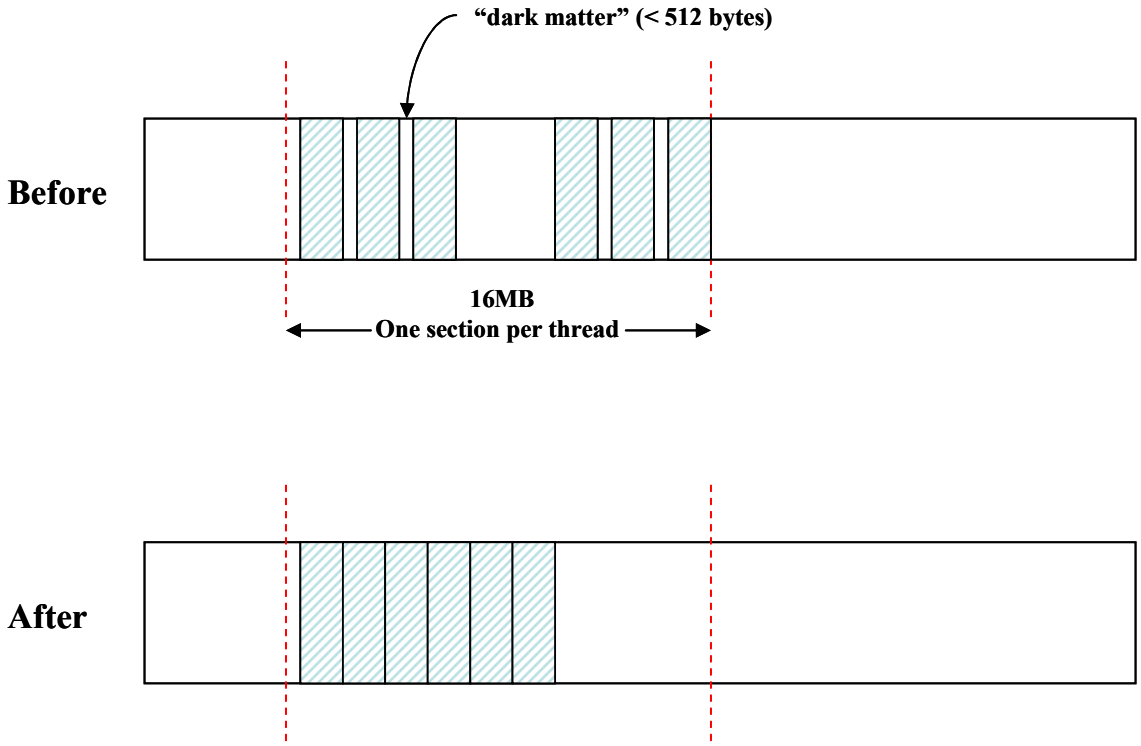

Figure 12. Incremental Compaction

In the upper diagram of Figure 12, marked portions are live objects, and unmarked portions represent free space. Dark matter might exist among the free spaces, depending on the size of the free chunk. The lower diagram of Figure 11 shows the condition after incremental compaction. Live objects have been moved to one end of the section, thereby freeing up space on the other end.

Incremental compaction is done only if the heap size is greater than a minimum value, currently 128 MB. If the heap size is less than 128 MB, incremental compaction fails to provide significant improvement in pause time compared to full compaction.

Incremental compaction has two main steps:

1) Identify and remember all references that point into the compaction region; this action is done during the mark phase. At the end of this stage, all free space that is in the sections can be identified.
2) Compute the new locations of objects and move them in the compaction region. Then set up pointers to those objects.

Incremental compaction operates in a cycle. An incremental compaction cycle is a cycle of successive garbage collection cycles that incrementally compacts the whole heap, a region at a time. The compaction spans multiple garbage collection cycles, therefore spreading compaction time over multiple garbage collections and reducing pause times.

4.3.2.3 Main parameters related to IC

Incremental compaction is set on by default. The decision to run incremental compaction in any given garbage collection cycle is made depending on a few triggers (see section "4.8.11 verbosegc for a concurrent mark collection with :Xgccon"). However, two parameters let the user decide whether to run with incremental compaction or with conventional compaction. Those parameters are:

- **-Xpartialcompactgc**, which initiates an incremental compaction every garbage collection cycle, unless a full compaction is required
- **-Xnopartialcompactgc,** which turns off incremental compaction for the life of the JVM.

However, note that these **-X** options are nonstandard and are subject to change without notice.

4.4 Reference Objects

Reference objects enable all references to be handled and processed in the same way. Therefore, the Garbage Collector creates two separate objects on the heap: the object itself and a separate reference object. The reference objects can optionally be associated with a queue to which they will be added when the referent becomes unreachable. Instances of SoftReference, WeakReference, and PhantomReference are created by the user and cannot be changed; they cannot be made to refer to objects other than the object that they referenced on creation. Objects that are associated with a finalizer are 'registered' with the Finalizer class on creation. The result is the creation of a FinalReference object that is associated with the Finalizer queue and that refers to the object that is to be finalized.

During garbage collection, these reference objects are handled specially; that is, the referent field is not traced during the marking phase. When marking is complete, the references are processed in sequence:

1) Soft
2) Weak
3) Final
4) Phantom

Processing of SoftReference objects is specialized; that is, the ST component can decide that these references should be cleared if the referent is unmarked (unreachable except for a path through a reference). The clearing is done if memory is running out and is done selectively on the principle of most recent usage. Usage is measured by the last time that the get method was called, which can give some unexpected, although valid, results. When a reference object is being processed, its referent is marked, ensuring that when, for example, a FinalReference is processed for an object that also has a SoftReference, when processing the FinalReference a marked referent is seen. The FinalReference, therefore, is not queued for processing. The result is that references are queued in successive garbage collection cycles.

References to unmarked objects are initially queued to the ReferenceHandler thread that is in the reference class. The ReferenceHandler takes objects off its queue and looks at their individual **queue** field. If an object is associated with a specific queue, it is requeued to it for further processing. Therefore, the FinalReference objects are requeued and eventually their finalize method is run by the finalizer thread.

4.4.1 JNI weak references

JNI weak references provide the same capability as WeakReference objects do, but the processing is very different. A JNI routine can create a JNI Weak reference to an object and later delete that reference. The Garbage Collector clears any weak reference where the referent is unmarked, but no equivalent of the queuing mechanism exists. Note that failure to delete a JNI Weak reference causes a memory leak in the table and performance

problems. This is also true for JNI global references. The processing of JNI weak references is handled last in the reference handling process. The result is that a JNI weak reference can exist for an object that has already been finalized and had a phantom reference queued and processed.

4.5 Heap Expansion

Heap expansion occurs after garbage collection and after all the threads have been restarted, but while the HEAP_LOCK is still held. The active part of the heap is expanded up to the maximum if any one of the following is true:

- The Garbage Collector did not free enough storage to satisfy the allocation request.
- Free space is less than the minimum free space, which you can set by using the **-Xminf** parameter. The default is 30%.
- More than 13% of the time is being spent in garbage collection, and expanding by the minimum expansion amount (**-Xmine**) does not result in a heap that is greater than the maximum percentage of free space (**-Xmaxf**).

The amount to expand the heap is calculated as follows:

- If the heap is being expanded because less than **-Xminf** (default 30%) free space is available, the Garbage Collector calculates how much the heap needs to expand to get **-Xminf** free space. If this is greater than the maximum expansion amount, which you can set with the **-Xmaxe** parameter (default of 0, which means no maximum expansion), the calculation is reduced to -**Xmaxe**. If this is less than the minimum expansion amount, which you can set with the **-Xmine** parameter (default of 1 MB), it is increased to **-Xmine**.
- If the heap is expanding because the Garbage Collector did not free enough storage and the JVM is not spending more than 13% in garbage collection, the heap is expanded by the allocation request.
- If the heap is expanding for any other reason, the Garbage Collector calculates how much expansion is needed to get 17.5% free space. This is adjusted as above, depending on **-Xmaxe** and **-Xmine**.
- Finally, the Garbage Collector must ensure that the heap is expanded by at least the allocation request if garbage collection did not free enough storage.

All calculated expansion amounts are rounded up to a 64 KB boundary on 32-bit architecture, or a 4 MB boundary on 64-bit architecture.

4.6 Heap Shrinkage

Heap shrinkage occurs after garbage collection, but when all the threads are still suspended. Shrinkage does not occur if any one of the following is true:

- The Garbage Collector did not free enough space to satisfy the allocation request.
- The maximum free space, which can be set by the **-Xmaxf** parameter (default is 60%), is set to 100%.
- The heap has been expanded in the last three garbage collections.
- This is a System.gc() and the amount of free space at the beginning of the garbage collection was less than **-Xminf** (default is 30%) of the live part of the heap.

If none of the above is true and more than **-Xmaxf** free space exists, the Garbage Collector must calculate by how much to shrink the heap to get it to **-Xmaxf** free space, without going below the initial (**-Xms**) value. This figure is rounded down to a 64 KB boundary on 32-bit architecture, or a 4 MB boundary on 64-bit architecture.

A compaction occurs before the shrink if all the following are true:

- A compaction was not done on this garbage collection cycle.
- No free chunk is at the end of the heap, or the size of the free chunk that is at the end of the heap is less than 10% of the required shrinkage amount.
- The Garbage Collector did not shrink and compact on the last garbage collection cycle

4.7 Resettable JVM

The resettable JVM was introduced in release 1.3.0, and is available only on z/OS.

Documentation on the Resettable JVM can be found in the “Persistent Reusable Java Virtual Machine” user’s guide.

This is available externally at http://www.s390.ibm.com/Java

4.8 verbosegc

A good way to see what is going on with Garbage Collection is to use verbosegc which is enabled by the **-verbosegc** option.

4.8.1 verbosegc output from a System.gc

```
<GC(3): GC cycle started Tue Mar 19 08:24:34 2002
<GC(3): freed 58808 bytes, 27% free (1163016/4192768), in 14 ms>
<GC(3): mark: 13 ms, sweep: 1 ms, compact: 0 ms>
<GC(3): refs: soft 0 (age >= 32), weak 0, final 0, phantom 0>
```

Figure 13. Example of verbosegc from a System.gc

Figure 13**Error! Reference source not found.** shows an example of a System.gc() collection, or forced garbage collection. All the lines start with GC(3), which indicates that this was the third garbage collection in this JVM. The first line shows the date and time of the start of the collection. The second line shows that 58808 bytes were freed in 14 ms, resulting in 27% free space in the heap. The figures in parentheses show the actual number of bytes that are free, and the total bytes that are available in the heap. The third line shows the times for the mark, sweep, and compaction phases. In this case, no compaction occurred, so the time is zero. The last line shows the reference objects that were found during this garbage collection, and the threshold for removing soft references. In this case, no reference objects were found.

4.8.2 verbosegc output from an allocation failure

```
<AF[5]: Allocation Failure. need 32 bytes, 286 ms since last AF>
<AF[5]: managing allocation failure, action=1 (0/6172496) (247968/248496)>
<GC(6): GC cycle started Tue Mar 19 08:24:46 2002
<GC(6): freed 1770544 bytes, 31% free (2018512/6420992), in 25 ms>
<GC(6): mark: 23 ms, sweep: 2 ms, compact: 0 ms>
<GC(6): refs: soft 1 (age >= 4), weak 0, final 0, phantom 0>
<AF[5]: completed in 26 ms>
```

Figure 14. Example of verbosegc from an allocation failure

Figure 14 shows an example of an allocation failure (AF) collection. An allocation failure does not mean that an error has occurred in the code; it is the name that is given to the event that triggers when it is not possible to allocate a large enough chunk from the heap. The output contains the same four lines that are in the System.gc() verbose output, and some additional lines. The lines that start with AF[5] are the allocation failure lines and indicate that this was the fifth AF collection in this JVM. The first line shows how many bytes were required by the allocation that had a failure, and how long it has been since the last AF. The second line shows what action the Garbage Collector is taking to solve

the AF, and how much free space is available in the main part of the heap, and how much is available in the wilderness. The possible AF actions are:

- 0 - The Garbage Collector has tried to allocate from the pinned free list, and failed.
- 1 - A garbage collection to avoid use of the wilderness. It is designed to avoid compactions by keeping the wilderness available for a large allocation request.
- 2 - The Garbage Collector has tried to allocate out of the wilderness, and failed.
- 3 - The Garbage Collector is going to attempt to expand the heap.
- 4 - The Garbage Collector is going to clear any remaining soft references. This occurs only if less than 12% free space is available in a fully expanded heap.
- 5 - This action applies only to resettable mode and means that garbage collection is going to try to take some space from the transient heap.
- 6 - This is not an action. It outputs a verbosegc message to say that the JVM is very low on heap space, or totally out of heap space.

The last line shows how long the AF took, including the time taken to stop and start all the application threads.

4.8.3 verbosegc for a heap expansion

```
<AF[11]: Allocation Failure. need 24 bytes, 182 ms since last AF>
<AF[11]: managing allocation failure, action=1 (0/6382368) (10296/38624)>
<GC(12): GC cycle started Tue Mar 19 08:24:49 2002
<GC(12): freed 1877560 bytes, 29% free (1887856/6420992), in 21 ms>
<GC(12): mark: 19 ms, sweep: 2 ms, compact: 0 ms>
<GC(12): refs: soft 0 (age >= 32), weak 0, final 0, phantom 0>
<AF[11]: managing allocation failure, action=3 (1887856/6420992)>
<GC(12): need to expand mark bits for 7600640-byte heap>
<GC(12): expanded mark bits by 16384 to 118784 bytes>
<GC(12): need to expand alloc bits for 7600640-byte heap>
<GC(12): expanded alloc bits by 16384 to 118784 bytes>
<GC(12): expanded heap by 1179648 to 7600640 bytes, 40% free>
<AF[11]: completed in 31 ms>
```

Figure 15. Example of verbosegc for heap expansion

Figure 15 shows an example of verbosegc for an AF collection that includes a heap expansion. The output is the same as a verbosegc output for an AF, plus some additional lines for the expansion. It shows by how much the markbits, the allocbits, and the heap are expanded, and how much free space is available. In the example, the heap was expanded by 1179648 bytes to give 40% free space.

4.8.4 verbosegc for a heap shrinkage

```
<AF[9]: Allocation Failure. need 32 bytes, 92 ms since last AF>
<AF[9]: managing allocation failure, action=1 (0/22100560) (1163184/1163184)>
<GC(9): may need to shrink mark bits for 22149632-byte heap>
<GC(9): shrank mark bits to 348160>
<GC(9): may need to shrink alloc bits for 22149632-byte heap>
<GC(9): shrank alloc bits to 348160>
<GC(9): shrank heap by 1114112 to 22149632 bytes, 79% free>
<GC(9): GC cycle started Tue Mar 19 11:08:18 2002
<GC(9): freed 17460600 bytes, 79% free (17509672/22149632), in 7 ms>
<GC(9): mark: 4 ms, sweep: 3 ms, compact: 0 ms>
<GC(9): refs: soft 0 (age >= 6), weak 0, final 0, phantom 0>
<AF[9]: completed in 8 ms>
```

Figure 16. Example of verbosegc for heap shrinkage

Figure 16 shows an example of verbosegc for an AF collection that includes heap shrinkage. This output is very similar to the verbosegc output for heap expansion. It shows by how much the markbits, the allocbits, and the heap are shrunk, and how much free space is available. In the example, the heap shrank by 1114112 bytes to give 79% free space. One other difference between the verbosegc output for heap expansion and heap shrinkage is the sequence of the output. This difference occurs because expansion happens after all the threads have been restarted and shrinkage happens before all the threads have been restarted.

4.8.5 verbosegc for a compaction

```
<AF[2]: Allocation Failure. need 88 bytes, 5248 ms since last AF>
<AF[2]: managing allocation failure, action=1 (0/4032592) (160176/160176)>
<GC(2): GC cycle started Tue Mar 19 11:32:28 2002
<GC(2): freed 1165360 bytes, 31% free (1325536/4192768), in 63 ms>
<GC(2): mark: 13 ms, sweep: 1 ms, compact: 49 ms>
<GC(2): refs: soft 0 (age >= 32), weak 0, final 3, phantom 0>
<GC(2): moved 32752 objects, 2511088 bytes, reason=2, used 8 more bytes>
<AF[2]: completed in 64 ms>
```

Figure 17. Example of verbosegc for compaction

Figure 17 shows an example of verbosegc for a compaction. The main difference between this and the outputs for a normal AF collection is the additional line that shows how many objects have been moved, how many bytes have been moved, the reason for the compaction, and how many additional bytes have been used. It is possible to use additional bytes if the Garbage Collector moves an object that has been hashed as it has to store the hash value in the object which might mean increasing the object size. The

“reason” will be “IC reason” if this was an incremental compaction. The possible reasons for a compaction are as follows:

- 1 - Following the mark and sweep phase, not enough free space is available for the allocation request.
- 2 - The heap is fragmented and will benefit from a compaction.
- 3 - Less than half the **-Xminf** value is free space (the default is 30% in which case this will be less than 15% free space), and the free space plus the dark matter is not less than **-Xminf**.
- 4 - A System.gc() collection.
- 5 - Less than 5% free space is available.
- 6 - Less than 128 KB free space is available.
- 7 - The **-Xcompactgc** parameter has been specified.
- 8 - The transient heap has less than 5% free space available.
- 11 - A compaction occurred before the attempt to shrink the heap.
- 12 - An incremental compaction occurred because of excessive dark matter
- 13 - The **-Xpartialcompactgc** parameter has been specified.
- 14 - An incremental compaction occurred because of wilderness expansion.
- 15 - An incremental compaction occurred because not enough free space is available in the wilderness.

4.8.6 verbosegc for concurrent mark kick-off

```
<CONCURRENT GC Free= 379544 Expected free space=    378884 Kickoff=379406>
<  Initial Trace rate is   8.01>
```

Figure 18. Example of verbosegc for concurrent mark kick-off

Figure 18 shows the two lines that are the verbosegc output that indicate that the concurrent phase has started. The first line shows how much free space is available, and how much will be available after this heap lock allocation. The Kickoff value is the level at which concurrent mark starts. In this example, the expected space is 378884, which is less than the Kickoff value of 379406. The second line shows the initial trace rate. In this example, it is 8.01, which means that for every byte that is allocated in a heap lock allocation, the Garbage Collector must trace 8.01 bytes of live data.

4.8.7 verbosegc for a concurrent mark System.gc collection

```
<GC(23): Bytes Traced =0 (Foreground: 0+ Background: 0) State = 3 >
<GC(23): GC cycle started Fri Oct 11 08:45:34 2002
<GC(23): freed 12847376 bytes, 94% free (127145208/134216192), in 975 ms>
<GC(23): mark: 408 ms, sweep: 70 ms, compact: 497 ms>
<GC(23): refs: soft 0 (age >= 32), weak 0, final 0, phantom 0>
<GC(23): moved 95811 objects, 6316896 bytes, reason=4>
```

Figure 19. Example of verbosegc for a concurrent mark System.gc

Figure 19 shows an example of verbosegc for a concurrent mark System.gc. The first line of the output with concurrent mark shows the state as a numeric value. The possible values for this field are:

- HALTED (0)
- EXHAUSTED (1)
- EXHAUSTED_BK_HELPER (2)
- ABORTED (3).

In this case, it is 3 (ABORTED) to show that concurrent mark did not complete the initialization phase and was therefore aborted. The output also shows the number of bytes which were traced during the concurrent phase, this is also split to show the amount of tracing in the foreground and background.

4.8.8 verbosegc for a concurrent mark AF collection

```
<AF[7]: Allocation Failure. need 528 bytes, 493 ms since last AF or CON>
<AF[7]: managing allocation failure, action=1 (0/3983128) (209640/209640)>
<GC(8): Bytes Traced =670940 (Foreground: 73725+ Background: 597215) State = 0
<GC(8): GC cycle started Tue Oct 08 13:43:14 2002
<GC(8): freed 2926496 bytes, 74% free (3136136/4192768), in 8 ms>
<GC(8): mark: 7 ms, sweep: 1 ms, compact: 0 ms>
<GC(8): refs: soft 0 (age >= 32), weak 0, final 0, phantom 0>
<AF[7]: completed in 10 ms>
```

Figure 20. Example of verbosegc for a concurrent mark AF collection

Figure 20 shows an example of verbosegc for an AF collection with concurrent mark running.

The *Traced* figures in parentheses show how much is traced by the application threads and how much is traced by the background thread. The total byte traced is the sum of the work that is done by the background and foreground traces. State is 0, which means concurrent is HALTED.

4.8.9 verbosegc for a concurrent mark AF collection with :Xgccon

```
<AF[19]: Allocation Failure. need 65552 bytes, 106 ms since last AF or CON>
<AF[19]: managing allocation failure, action=1 (83624/16684008) (878104/878104)>
<GC(20): Bytes Traced =1882061 (Foreground: 1292013+ Background: 590048) State = 0 >
<GC(20): Card Cleaning Done. Cleaned:27 (0 skipped). Estimation 593 (Factor 0.017)>
<GC(20): GC cycle started Fri Oct 11 10:23:49 2002
<GC(20): freed 8465280 bytes, 53% free (9427008/17562112), in 9 ms>
<GC(20): mark: 7 ms, sweep: 2 ms, compact: 0 ms>
<GC(20): In mark: Final dirty Cards scan: 41 cards
<GC(20): refs: soft 0 (age >= 6), weak 0, final 0, phantom 0>
```

Figure 21. Example of verbosegc for a concurrent mark AF collection with :Xgccon

Figure 21 shows an example of verbosegc for an AF collection with concurrent mark running and the **:Xgccon** parameter set. Line 3 shows a state of 0, which means concurrent is HALTED. Line 4 shows that concurrent card cleaning was performed for 27 cards, while estimation is the number of dirty cards found.

4.8.10 verbosegc for a concurrent mark collection

```
<CON[41]: Concurrent collection, (284528/8238832) (17560/17168), 874 ms since last CON or AF>
<GC(45): Bytes Traced =5098693 (Foreground: 555297+ Background: 4543396) State = 2 >
<GC(45): GC cycle started Tue Oct 08 12:31:14 2002
<GC(45): freed 2185000 bytes, 30% free (2487088/8256000), in 7 ms>
<GC(45): mark: 5 ms, sweep: 2 ms, compact: 0 ms>
<GC(45): refs: soft 0 (age >= 32), weak 0, final 0, phantom 0>
<CON[41]: completed in 9 ms>
```

Figure 22. Example of verbosegc for a concurrent mark collection

Figure 22 shows an example of verbosegc for a collection that was initiated by concurrent mark. It is very similar to the AF concurrent collection, except that CON is at the start of the lines instead of AF. In this case, the state is 2, meaning that no more work was available for the background threads to do.

4.8.11 verbosegc for a concurrent mark collection with :Xgccon

```
<CON[20]: Concurrent collection, (397808/131070464) (3145728/3145728), 5933 ms since last CON or AF>
<GC(26): Bytes Traced =11845976 (Foreground: 4203037+ Background: 7642939) State = 1 >
<GC(26): Card Cleaning Done. Cleaned:4127 (0 skipped). Estimation 3896 (Factor 0.015)>
<GC(26): GC cycle started Fri Oct 11 09:45:32 2002
<GC(26): wait for concurrent tracers: 1 ms>
<GC(26): freed 117639824 bytes, 90% free (121183360/134216192), in 20 ms>
<GC(26): mark: 10 ms, sweep: 10 ms, compact: 0 ms>
<GC(26): In mark: Final dirty Cards scan: 838 cards
<GC(26): refs: soft 0 (age >= 32), weak 0, final 0, phantom 0>
<CON[20]: completed in 21 ms>
```

Figure 23. Example of verbosegc for a concurrent mark collection with :Xgccon

Figure 23 shows the output for a concurrent mark collection when the **:Xgccon** parameter is specified. As with the AF collection we get the information on the bytes traced and card cleaning. An additional line (5) is displayed to show the time that is spent waiting for concurrent tracers to complete.

4.8.12 verbosegc and resettable

```
<TH_AF[8]: Transient heap Allocation Failure. need 64 bytes, 9716 ms since last TH_AF>
<TH_AF[8]: managing TH allocation failure, action=3 (0/4389888)>
<GC(25): need to expand transient mark bits for 4586496-byte heap>
<GC(25): expanded transient mark bits by 3072 to 71672 bytes>
<GC(25): need to expand transient alloc bits for 4586496-byte heap>
<GC(25): expanded transient alloc bits by 3072 to 71672 bytes>
<GC(25): expanded transient heap fully by 196608 to 4586496 bytes>
<TH_AF[8]: completed in 1 ms>
```

Figure 24. Example of verbosegc for a Transient heap AF

When running resettable, the JVM has a middleware heap and a transient heap. The verbosegc for the transient heap is slightly different, Figure 24 is an example, note the use of TH_AF instead of AF. The policy when running resettable is to expand the transient heap when an allocation failure occurs, instead of running garbage collection. This example shows a successful expansion.

```
<TH_AF[11]: Transient heap Allocation Failure. need 32 bytes, 16570 ms since last TH_AF>
<TH_AF[11]: managing TH allocation failure, action=3 (0/4586496)>
<TH_AF[11]: managing TH allocation failure, action=2 (0/4586496)>
<GC(29): GC cycle started Tue Mar 19 14:47:42 2002
<GC(29): freed 402552 bytes from Transient Heap 8% free (402552/4586496) and...>
<GC(29): freed 1456 bytes, 38% free (623304/1636864), in 1285 ms>
<GC(29): mark: 1263 ms, sweep: 22 ms, compact: 0 ms>
<GC(29): refs: soft 0 (age >= 6), weak 0, final 0, phantom 0>
<TH_AF[11]: completed in 1287 ms>
```

Figure 25. Example of verbosegc for a Transient Heap AF with unsuccesful expansion

Figure 25 shows what happens when the expansion is not successful. Here a garbage collection is necessary. The amount of space that is freed from each of the heaps is shown.

5 Messages

JVMST001: Cannot allocate memory in initWorkPackets
Explanation: Not enough virtual storage was available to allocate the concurrent data structures. The call to sysMalloc() failed. This can happen only during initialization.
System Action: The JVM is terminated.
User Response: Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

JVMST010: Cannot allocate memory for ACS area
Explanation: Not enough virtual storage was available to allocate the ACS heap. The call to sharedMemoryAlloc() failed. This can happen during the initialization or expansion of the ACS heap.
System Action: The JVM is terminated.
User Response: Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

JVMST011: Cannot allocate memory in initConcurrentRAS
Explanation: Not enough virtual storage was available to allocate the mirrored card table. The call to sysMapMem() failed. This can happen only in the debug build during initialization.
System Action: The JVM is terminated.
User Response: Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

JVMST012: Cannot allocate memory in concurrentInit()
Explanation: Not enough virtual storage was available to allocate the stop_the_world_mon monitor. The call to sysMalloc() failed. This can happen only during initialization.
System Action: The JVM is terminated.
User Response: Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

JVMST013: Cannot allocate memory in initGcHelpers(2)
Explanation: Not enough virtual storage was available to allocate the ack_mon monitor. The call to sysMalloc() failed. This can happen only during initialization.
System Action: The JVM is terminated.
User Response: Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

JVMST014: Cannot allocate memory in initConBKHelpers(3)
Explanation: Not enough virtual storage was available to start a concurrent background thread. The call to xmCreateSystemThread() failed. This can happen only during initialization.
System Action: The JVM is terminated.
User Response: Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

JVMST015: Cannot commit memory in initConcurrentRAS
Explanation: An error occurred during an attempt to commit memory for the mirrored card table. The call to sysCommitMem() failed. This can happen only in the debug build during initialization.
System Action: The JVM is terminated.
User Response: Contact your IBM service representative.

JVMST016: Cannot allocate memory for initial Java heap
Explanation: Not enough virtual storage was available to allocate the Java heap. The call to sysMapMem() failed. This can happen only during initialization.
System Action: The JVM is terminated.
User Response: Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

JVMST017: Cannot allocate memory in initializeMarkAndAllocBits(markbits1)
Explanation: Not enough virtual storage was available to allocate the markbits vector. The call to sysMapMem() failed. This can happen only during initialization.
System Action: The JVM is terminated.
User Response: Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

JVMST018: Cannot allocate memory for initializeMarkAndAllocBits(allocbits1)
Explanation: Not enough virtual storage was available to allocate the allocbits vector. The call to sysMapMem() failed. This can happen only during initialization.
System Action: The JVM is terminated.
User Response: Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

JVMST019: Cannot allocate memory in allocateToMiddlewareHeap
Explanation: An error occurred during an attempt to commit memory for the Java heap. The call to sysCommitMem() failed. This can only happen during expansion of the heap.
System Action: The JVM is terminated.
User Response: Contact your IBM service representative.

JVMST020: Cannot allocate memory in allocateToTransientHeap
Explanation: An error occurred during an attempt to commit memory for the transient heap. The call to sysCommitMem() failed. This can happen during initialization or during expansion of the transient heap.
System Action: The JVM is terminated.
User Response: Contact your IBM service representative.

JVMST021: Cannot allocate memory in initParallelMark(stackEnd
Explanation: Not enough storage was available in the Java heap to allocate the stackEnd object. The call to allocMiddlewareArray() failed. This can happen only during initialization.
System Action: The JVM is terminated.
User Response: Allocate more Java heap storage by increasing the -Xmx value. If the problem remains, contact your IBM service representative.

JVMST022: Cannot allocate memory in initParallelMark(pseudoClass)
Explanation: Not enough storage was available in the Java heap to allocate the pseudoClass object. The call to allocMiddlewareObject() failed. This can happen only during initialization.
System Action: The JVM is terminated.
User Response: Allocate more Java heap storage by increasing the -Xmx value. If the problem remains, contact your IBM service representative.

JVMST023: Cannot allocate memory in initializeGCFacade
Explanation: Not enough virtual storage was available to allocate the verbosegc buffer. The call to sysMalloc() failed. This can happen only during initialization.
System Action: The JVM is terminated.
User Response: Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

JVMST024: Cannot allocate memory in initWorkPackets
Explanation: Not enough virtual storage was available to allocate the concurrent data structures. The call to sysMalloc() failed. This can happen only during initialization.
System Action: The JVM is terminated.
User Response: Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

JVMST025: Cannot allocate memory in icDoseThread
Explanation: Not enough virtual storage was available to allocate a sys_thread_stack_segment. The call to sysCalloc() failed. This can happen only during garbage collection.
System Action: The JVM is terminated.
User Response: Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

JVMST026: Cannot allocate memory in initializeMiddlewareHeap (not enough memory)
Explanation: An error occurred during an attempt to allocate storage to the middleware heap. The call to allocateToMiddlewareHeap() failed. This can happen only during initialization.
System Action: The JVM is terminated.
User Response: Contact your IBM service representative.

JVMST027: Cannot allocate memory for System Heap area in allocateSystemHeapMemory
Explanation: Not enough virtual storage was available to allocate storage for the system heap. The call to sharedMemoryAlloc() failed. This can happen during initialization or when expanding the system heap.
System Action: The JVM is terminated.
User Response: Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

JVMST028: Cannot commit memory in RASinitShadowHeap
Explanation: An error occurred during an attempt to commit memory for the shadow heap. The call to sysCommitMem() failed. This can happen only during initialization when tracing st_shadowheap.
System Action: The JVM is terminated.
User Response: Contact your IBM service representative.

JVMST029: Cannot allocate memory in jvmpi_scan_thread_roots
Explanation: Not enough virtual storage was available to allocate a sys_thread_stack_segment. The call to sysCalloc() failed. This can happen only during garbage collection when jvmpi is running.
System Action: The JVM is terminated.
User Response: Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

JVMST030: Cannot allocate memory in initializeCardTable
Explanation: Not enough virtual storage was available to allocate the card table. The call to sysMapMem() failed. This can happen only during initialization.
System Action: The JVM is terminated.
User Response: Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

JVMST031: Cannot commit memory in initializeCardTable
Explanation: An error occurred during an attempt to commit memory for the card table. The call to sysCommitMem() failed. This can happen only during initialization.
System Action: The JVM is terminated.
User Response: Contact your IBM service representative.

JVMST032: Cannot allocate memory in initializeTransientHeap
Explanation: An error occurred during an attempt to allocate storage to the transient heap. The call to allocateToTransientHeap() failed. This can happen only during initialization.
System Action: The JVM is terminated.
User Response: Contact your IBM service representative.

JVMST033: Cannot allocate memory in initializeMarkAndAllocBits(markbits2)
Explanation: An error occurred during an attempt to commit memory for the markbits vector. The call to sysCommitMem() failed. This can happen only during initialization when running with -Xresettable.
System Action: The JVM is terminated.
User Response: Contact your IBM service representative.

JVMST034: Cannot allocate memory in initializeMarkAndAllocBits(allocbits2)
Explanation: An error occurred during an attempt to commit memory for the allocbits vector. The call to sysCommitMem() failed. This can happen only during initialization when running with -Xresettable.
System Action: The JVM is terminated.
User Response: Contact your IBM service representative.

JVMST035: Cannot allocate memory in initializeMiddlewareHeap (markbits)
Explanation: An error occurred during an attempt to commit memory for the markbits vector. The call to sysCommitMem() failed. This can happen only during initialization when -Xresettable is not running.
System Action: The JVM is terminated.
User Response: Contact your IBM service representative.

JVMST036: Cannot allocate memory in initializeMiddlewareHeap (allocbits)
Explanation: An error occurred during an attempt to commit memory for the allocbits vector. The call to sysCommitMem() failed. This can happen only during initialization when -Xresettable is not running.
System Action: The JVM is terminated.
User Response: Contact your IBM service representative.

JVMST039: Cannot allocate Shared Memory segment in initializeSharedMemory
Explanation: An error occurred during an attempt to create shared memory. The call to xhpiSharedMemoryCreate() failed. This can happen only during initialization when -Xjvmset is running.
System Action: A return code of JNI_ENOMEM will be passed back to the JNI_CreateJavaVM call.
User Response: Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

JVMST040: Cannot initialize Java heap in allocateToMiddlewareHeap
Explanation: An error occurred during an attempt to commit memory for the Java heap. The call to sysCommitMem() failed. This can happen only during initialization.
System Action: The JVM is terminated.
User Response: Contact your IBM service representative.

JVMST042: Cannot allocate memory in initParallelMark(base-Malloc)
Explanation: Not enough virtual storage was available to allocate the parallel mark data structures. The call to sysMalloc() failed. This can happen only during initialization.
System Action: The JVM is terminated.
User Response: Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

JVMST043: Cannot allocate memory in concurrentScanThread
Explanation: Not enough virtual storage was available to allocate a sys_thread_stack_segment. The call to sysCalloc() failed. This can happen only during concurrent marking.
System Action: The JVM is terminated.
User Response: Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

JVMST044: Cannot allocate memory in concurrentInitLogCleaning
Explanation: Not enough virtual storage was available to allocate the cleanedbits vector. The call to sysMapMem() failed. This can happen only during initialization.
System Action: The JVM is terminated.
User Response: Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

JVMST045: Cannot commit memory in concurrentInitLogCleaning
Explanation: An error occurred during an attempt to commit memory for the cleanedbits. The call to sysCommitMem() failed. This can happen only during initialization.
System Action: The JVM is terminated.
User Response: Contact your IBM service representative.

JVMST046: Cannot allocate storage for standalone jab in initializeSharedMemory
Explanation: Not enough virtual storage was available to allocate the JAB. The call to sysCalloc() failed. This can happen only during initialization when -Xjvmset is not running.
System Action: A return code of JNI_ENOMEM is passed back to the JNI_CreateJavaVM call.
User Response: Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

JVMST047: Cannot allocate memory in initParallelSweep
Explanation: Not enough virtual storage was available to allocate the parallel sweep data structure PBS_ThreadStat. The call to sysMalloc() failed. This can happen only during initialization.
System Action: The JVM is terminated.
User Response: Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

JVMST048: Could not establish access to shared storage in openSharedMemory
Explanation: An error occurred during an attempt to access shared memory. The call to xhpiSharedMemoryOpen() failed. This can happen only during initialization when -Xjvmset is running.
System Action: A return code of JNI_ENOMEM is passed back to the JNI_CreateJavaVM call.
User Response: Check whether the correct token is being passed in the JavaVMOption. If the problem remains, contact your IBM service representative.

JVMST049: Worker and Master JVM versions differ
Worker JVM version is <*versio***n>** build type is <*buil*d>
Master JVM version is <*versio***n>** build type is <*buil*d>
Where *version* is the JVM version (for example 1.3) and *build* is the build type (DEV, COL, or INT).
Explanation: A mismatch has occurred between the Master JVM and a Worker JVM. This can happen only during initialization when -Xjvmset is running.
System Action: A return code of JNI_ERR is passed back to the JNI_CreateJavaVM call.
User Response: Ensure that the Master and all Worker JVMs are at the same version level, and all are of the same build type. If the problem remains, contact your IBM service Representative.

JVMST050: Cannot allocate memory for initial Java heap
Explanation: An error occurred during an attempt to query memory availability. The call to DosQuerySysInfo() failed. This can happen only during initialization on OS/2.
System Action: The JVM is terminated.
User Response: Contact your IBM service representative.

JVMST051: Cannot allocate memory for initial Java heap
Explanation: Not enough virtual storage was available to allocate the Java heap. The call to sysMapMem() failed. This can happen only during initialization on OS/2.
System Action: The JVM is terminated.
User Response: Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

JVMST052: Cannot allocate memory for initial Java heap
Explanation: Not enough virtual storage was available to allocate the Java heap. The call to sysMapMem() failed. This can happen only during initialization on OS/2 and when JAVA_HIGH_MEMORY has been specified.
System Action: The JVM is terminated.
User Response: Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

JVMST053: Cannot allocate memory in initParallelMark(legacy list)
Explanation: Not enough virtual storage was available to allocate the legacyList. The call to sysMalloc() failed. This can happen only during initialization.
System Action: The JVM is terminated.
User Response: Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

JVMST054: Cannot allocate memory in initParallelMark(nursery bits)
Explanation: Not enough virtual storage was available to allocate the nurserybits vector. The call to sysMalloc() failed. This can happen only during initialization.
System Action: The JVM is terminated.
User Response: Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

JVMST055: Cannot allocate memory in initParallelSweep
Explanation: Not enough virtual storage was available to allocate the parallel sweep data structure pbs_scoreboard. The call to sysMalloc() failed. This can happen only during initialization.
System Action: The JVM is terminated.
User Response: Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

JVMST056: Cannot allocate memory in initConBKHelpers(1)
Explanation: Not enough virtual storage was available to allocate the bk_activation_mon monitor. The call to sysMalloc() failed. This can happen only during initialization.
System Action: The JVM is terminated.
User Response: Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

JVMST057: Cannot allocate memory in initGcHelpers(1)
Explanation: Not enough virtual storage was available to allocate the request_mon monitor. The call to sysMalloc() failed. This can happen only during initialization.
System Action: The JVM is terminated.
User Response: Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

JVMST058: Cannot allocate memory in initGcHelpers(3)
Explanation: Not enough virtual storage was available to start a gcHelper thread. The call to xmCreateSpecialSystemThread() failed. This can happen only during initialization.
System Action: The JVM is terminated.
User Response: Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

JVMST059: Cannot allocate memory in scanThread
Explanation: Not enough virtual storage was available to allocate a sys_thread_stack_segment. The call to sysCalloc() failed. This can happen only during garbage collection.
System Action: The JVM is terminated.
User Response: Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

JVMST060: Cannot allocate memory in concurrentInit
Explanation: Not enough virtual storage was available to allocate a backup thread in concurrent bk_threads. The call to sysCalloc() failed. This can happen only during garbage collection.
System Action: The JVM is terminated.
User Response: Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

JVMST061: Cannot allocate memory in concurrentInit
Explanation: Not enough virtual storage was available to allocate the concurrent tracer_mon monitor. The call to sysMalloc() failed. This can happen only during initialization.
System Action: The JVM is terminated.
User Response: Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

JVMST062: Cannot allocate memory in initializeFRBits
Explanation: Not enough virtual storage was available to allocate the FRBits. The call to sysMapMem() failed. This can happen only during initialization.
System Action: The JVM is terminated.
User Response: Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

JVMST063: Cannot allocate memory in initializeFRBits
Explanation: Not enough virtual storage was available to commit the FRBits in resettable code. The call to sysCommitMem() failed. This can happen only during initialization.
System Action: The JVM is terminated.
User Response: Contact your IBM service representative.

JVMST064: Cannot allocate memory in initializeMiddlewareHeap
Explanation: Not enough virtual storage was available to commit the FRBits in nonresettable code. The call to sysCommitMem() failed. This can happen only during initialization.
System Action: The JVM is terminated.
User Response: Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

JVMST065: Cannot allocate memory for break tables in initializeIncrementalCompaction
Explanation: Not enough virtual storage was available to create the break tables for incremental compaction. The call to sysMalloc() failed. This can happen only during initialization.
System Action: The JVM is terminated.
User Response: Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

JVMST066: Exception (sysGetExceptionCode()) received during openSharedMemory with token(token)
Explanation: Cannot access shared storage that is defined by the token that was returned by xhpiSharedMemoryOpen. This can happen only during initialization.
System Action: The JVM is terminated.
User Response: Check whether the token that is being passed by -Xjvmset is valid. If the problem remains, contact your IBM service representative.

JVMST067: Invalid method_type detected in heap allocation(allocObject)
Explanation: The class type that was detected during object allocation was not Middleware, Primordial, or Application.
System Action: The JVM is terminated.
User Response: If the problem remains, contact your IBM service representative.

JVMST068: Invalid method_type detected in heap allocation (allocArray)
Explanation: The class type that was detected during array allocation was not Middleware or Application.
System Action: The JVM is terminated.
User Response: If the problem remains, contact your IBM service representative.

JVMST069: Invalid method_type detected in heap allocation (allocConextArray)
Explanation: The class type that was detected during context array allocation was not Middleware or Application.
System Action: The JVM is terminated.
User Response: If the problem remains, contact your IBM service representative.

JVMST070: Invalid method_type detected in heap allocation (allocConextObject)
Explanation: The class type that was detected during context object allocation was not Middleware or Application.
System Action: The JVM is terminated.
User Response: If the problem remains, contact your IBM service representative.

JVMST080: verbose:gc is enabled
Explanation: Informational message**.**
System Action: None.
User Response: None.

JVMST081: file open failed for verbose:gc output file
Explanation: Cannot open the verbosegc log file.
System Action: Verbosegc log output will be written to the stderr log.
User Response: Check whether the entered file name is valid and whether open is a valid operation on this file.

JVMST082: -verbose:gc output will be written to (vgclogName)
Explanation: Informational message to display the location of the verbosegc output file.
System Action: None.
User Response: None.

JVMST083: Exception occurred while calculating freeList size for JVMMI
Explanation: Exception occurred while the jvmmiOutOfMemoryEvent was being set up.
System Action: The JVM is terminated.
User Response: If the problem remains, contact your IBM service representative.

JVMST084: Cannot allocate memory in stInit for segment_info
Explanation: Not enough virtual storage was available to create the sys_thread_stack_segment. The call to sysCalloc() failed. This can happen only during initialization.
System Action: The JVM is terminated.
User Response: Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

JVMST085: Cannot suspend threads in gc0
Explanation: An attempt by xmSuspendAllThreads to lock all threads before garbage collection was unsuccessful.
System Action: The JVM is terminated.
User Response: If the problem remains, contact your IBM service representative.

JVMST088: Cannot allocate memory in “initialiseSCCardTable”
Explanation: Not enough virtual storage was available to allocate the shared class card table. The call to sysMapMem() failed. This can happen only during initialization.
System Action: The JVM is terminated.
User Response: Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

JVMST089: Cannot commit memory in “initialiseSCCardTable”
Explanation: An error occurred during an attempt to commit memory for the shared class card table. The call to sysCommitMem() failed. This can happen only during initialization.
System Action: The JVM is terminated.
User Response: Contact your IBM service representative.

JVMST090: Incorrect usage of -Xverbosegclog
Explanation: The parameters that were passed with -Xverbosegclog are incorrect.
System Action: The JVM is terminated.
User Response: Review documentation on use of -Xverbosegclog. If the problem remains, contact your IBM service representative.

JVMST091: Incorrect usage of -Xverbosegclog
Explanation: The parameters that were passed with -Xverbosegclog are incorrect.
System Action: The JVM is terminated.
User Response: Review documentation on use of -Xverbosegclog. If the problem remains, contact your IBM service representative.

JVMST092: Cannot allocate memory in initializeGCFacade
Explanation: Not enough virtual storage was available to allocate the verbosegc trace buffer. The call to sysMalloc() failed. This can happen only during initialization.
System Action: The JVM is terminated.
User Response: Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

JVMST093: file open failed for verbose:gc output file
Explanation: Cannot open the verbosegc log file.
System Action: Verbosegc log output will be written to the stderr log.
User Response: Check whether the entered file name is valid and whether open is a valid operation on this file.

JVMST094: file open failed for verbose:gc output file
Explanation: Cannot open the verbosegc log file.
System Action: Verbosegc log output will be written to the stderr log.
User Response: Check whether the entered file name is valid and whether open is a valid operation on this file.

JVMST095: Incorrect usage of -Xverbosegclog
Explanation: The parameters that were passed with -Xverbosegclog are incorrect.
System Action: The JVM is terminated.
User Response: Review documentation on use of -Xverbosegclog. If the problem remains, contact your IBM service representative.

JVMST096: Out of memory in setVerbosegcRedirectionFormatScreen
Explanation: Not enough virtual storage was available to allocate the verbosegc buffer. The call to sysMalloc() failed. This can happen only during initialization.
System Action: The JVM is terminated.
User Response: Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

JVMST097: Concurrent GC is disabled
Explanation: An attempt has been made to use the dynamic switching interface to turn on concurrent verbosegc when concurrent gc is not enabled.
System Action: The JVM is terminated.
User Response: Review the dynamic switching interface

6 Command Line Parameters

The following list contains all the command line parameters related to allocation and garbage collection.

-verbosegc
-verbose:gc

Prints garbage collection information. The format for the generated information is not architected and therefore varies from platform to platform and release to release.

-verbose:Xgccon

Prints garbage-collection information, as supplied by **-verbose:gc**, and card-cleaning information. This can be set on only if **-Xgcpolicy** is set to optavgpause.

-Xverbosegclog:<path to file><filename>

Causes verbosegc output to be written to the specified file. If the file cannot be found, output is redirected to stderr.

-Xverbosegclog:<path to file><filename, X, Y>

Filename must contain a "#" (hash symbol), which is substituted with a generation identifier, starting at 1.

X and Y are integers. This option works like above but, in addition, the verbosegc output is redirected to Y files, each containing X gc cycles-worth of verbosegc output.

Note:
The environment variable IBM_JVMST_VERBOSEGC_LOG has been removed from release 1.4.1 onward.

-Xcompactgc

Compact the heap every garbage collection cycle. The default is false.

-Xdisableexplicitgc

Turns Java application calls to java.lang.System.gc() into no-ops.

Many applications in the field still make an excessive number of explicit calls to System.gc() to request garbage collection. In some cases, this can degrade performance time through premature garbage collection and compactions, but it is not always possible to remove the calls at source.

To allow the JVM to ignore these garbage collector suggestions, **-Xdisableexplicitgc** has been introduced. This would be used by system administrators with applications that show benefits with the new, nondefault setting.

-Xdisableexplicitgc should be used only in production where testing has shown this to be beneficial; for example, from performance testing in conjunction with verbose:gc output. The new flag should not be set when running:

- The zSeries JVM with CICS in resettable mode or DB/2 stored procedures
- Performance profilers that make explicit garbage collection calls to detect object freeing and memory leaks
- Performance benchmarks where explicit garbage collection calls are made between measurement intervals

-Xgcpolicy:<optthruput | optavgpause>

Setting gcpolicy to optthruput disables concurrent mark. Users who do not have pause time problems (as seen by erratic application response times) should get the best throughput with this option. Optthruput is the default setting.

Setting gcpolicy to optavgpause enables concurrent mark with its default values. Users who are having problems with erratic application response times caused by normal garbage collections can reduce those problems at the cost of some throughput when running with the optavgpause option.

-Xgcthreads

Sets the total number of threads that are used for garbage collection. On a system with N processors, the default setting for -Xgcthreads is 1 when in resettable mode, and N when not in resettable mode.

-Xinitacsh<*size*>

Sets the initial size of the application-class system heap. This option is available only in the resettable JVM. Classes that are in this heap exist for the lifetime of the JVM. They are reset during a ResetJavaVM(), and so are serially reusable by applications that are running in the JVM. Only one application-class system heap exists per Persistent Reusable JVM. In nonresettable mode, this option is ignored.
Example: **-Xinitacsh256k**
Default: 128 KB on 32-bit architecture, and 8 MB on 64-bit architecture.

-Xinitsh<*size*>

Sets the initial size of the system heap. Classes that are in this heap exist for the lifetime of the JVM. The system heap is never subjected to garbage collection. The maximum size of the system heap is unbounded.
Example: **-Xinitsh256k**
Default: 128 KB on 32-bit architecture, and 8 MB on 64-bit architecture.

-Xjvmset<*size*>

Creates a master JVM. An optional size in megabytes can be specified to set the total size of the shared memory segment. The default is 1 MB. When JNI_CreateJavaVM( )returns successfully, the "extrainfo" field of the JavaVMOption contains the token that is to be passed to each worker.

The **-Xresettable** option must be used with this option when starting a master JVM.

-Xjvmset

Creates a worker JVM. The "extrainfo" field of the JavaVMOption must contain the token that is returned on the **-Xjvmset** option that was used to create the master JVM.

-Xmaxe<*size*>

Specifies the maximum expansion size of the heap. The default is 0. In resettable mode,this sets the a maximum expansion size of <size> ÷ 2 for the middleware and transient heaps.

-Xmaxf<*number*>

This is a floating point number between 0 and 1 that specifies the maximum percentage of free space in the heap. The default is 0.6, or 60%. When this value is set to 0, heap contraction is a constant activity. When this value is set to 1, the heap never contracts. In resettable mode, this parameter applies to the middleware heap only.

-Xmine<*size*>

Specifies the minimum expansion size of the heap. The default is 1 MB. In resettable mode, this option sets a minimum expansion size of <size> ÷ 2 for the middleware and transient heaps.

-Xminf<*number*>

This is a floating point number between 0 and 1 that specifies the minimum free heap size percentage. The heap grows if the free space is below the specified amount. In resettable mode, this option specifies the minimum percentage of free space for the middleware and transient heaps. The default is 0.3 (that is 30%).

-Xms<*size*>

Sets the initial size of the heap. If this option is not specified, it defaults as follows:

- Windows, AIX, and Linux: 4 MB
- OS/390: 1 MB

-Xmx<*size*>

Sets the maximum size of the heap. In resettable mode, this option sets the maximum size of the combined middleware and transient heaps. The middleware heap grows from the bottom of this region, and the transient heap grows from the top of the region. If this option is not specified, it defaults as follows:

- ❖Windows: Half the real storage with a minimum of 16 MB and a maximum of 2 GB-1.
- ❖ OS/390 and AIX: 64 MB
- ❖ Linux: Half the real storage with a minimum of 16 MB and a maximum of 512 MB-1.

-Xnocompactgc

Never compact the heap. Default is “false”.

-Xnopartialcompactgc

Never run an Incremental Compaction. Default is “false”.

-Xpartialcompactgc

Run an Incremental Compaction every garbage collection cycle. Default is “false”.

-Xresettable

Specifies that this instance of the JVM can support the resettable JVM.

Appendix A. Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose

of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM United Kingdom Laboratories, MP146, Hursley Park, Winchester, Hampshire, SO21 2JN, United Kingdom. Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.
The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

The following terms are trademarks or registered trademarks of International Business Machines Corporation in the United States, or other countries, or both.

OS/390 IBM
AIX z/OS

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Other company, product and service names may be trademarks or service marks of others.