Understanding the IBM Java Garbage Collector

Author:	Sam Borman
	IBM United Kingdom Limited
	Java Technology Centre
	Mail Point 146
	IBM Hursley Park
	© Copyright IBM UK Limited, 2002
Revision level:	1
Last updated	28 June 2002

1 Introduction

This document describes the functions of the Storage (ST) component from release 1.2.2 to 1.3.1.

The ST component is responsible for the allocation of pieces of storage in the 'heap'. These pieces of storage are used to define objects, arrays and classes. Once allocated we assert that an object continues to be 'live' whilst 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 become 'garbage' and can be reclaimed for reuse. When this reclamation occurs we must take care of the process of **running** a possible finalizer and also ensure that any monitor 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 subtleties will be given later.

1.1 Object allocation

Object allocation is driven by calls to one of the allocation interfaces e.g. stCacheAlloc, stAllocObject, stAllocArray, stAllocClass. These 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 which 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 and hence it is very efficient. Objects allocated through the stAllocObject and stAllocArray interfaces will, if small enough (currently 512 bytes) also be allocated from the cache.

1.2 Reachable objects

The active state of the JVM is made up of the set of stacks representing the threads, the statics within Java classes, and the set of local and global JNI references. All invocations of functions within the JVM itself give rise to a frame on the C stack. We use this information to find what we call the roots, and then we use these roots to find references to other objects. This process is repeated until we find all reachable objects.

1.3 Garbage Collection

When we are unable to allocate an object from the current heap due to lack of space the first task is to collect all the 'garbage' in the heap. This process starts when any thread calls stGC which can either be a result of allocation failure or the result of a specific call to System.GC. The first step is to acquire all the locks which the GC process will need since we need to ensure that other threads are not suspended holding critical locks. All the other threads are then suspended using 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 and represents the state which we need to trace for object references. Garbage collection can then commence and is done in three phases; mark, sweep and optionally compact.

1.3.1 Mark Phase

In the mark phase we identify all the objects referenced from the thread stacks, statics, interned strings and JNI references. This forms the 'root set' of objects which are referenced by the JVM. Each of those objects may in turn reference others and thus the second part of the process is to scan each object for other references which it makes. In fact we combine these two processes by maintaining a bit vector which 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 we are tracing the stacks we first compare the 'pointer' against the low and high limits of the heap. We then ensure that it points to an object on an 8 byte boundary (GRAIN) and finally that the appropriate allocbit is set indicating that we really are pointing at an object. We now set a bit in the markbits vector to indicate that the object has been referenced. Finally we scan the fields of the object to search for other object references which it makes. This scan of the objects is done 'accurately' since we know the class of the object from the method pointer which is stored in its first word. This gives us access to a vector of offsets which the classloader builds at class linking time (prior to the creation of the first instance). The offsets vector gives the offset of fields in the object containing object references.

1.3.2 Sweep Phase

After the mark phase the markbits vector contains a bit for every reachable object in the heap and must be a subset of the allocbits vector. The task of the sweep phase is to identify the intersection of the allocbits and markbits vectors; objects which have been allocated but are no longer referenced. The 'original' technique for this was to start a scan from the bottom of the heap visiting each object in turn, the length of each object is held in the word immediately preceding it on the heap. At each object we test the appropriate allocbit and markbit to locate the garbage. The 'bitsweep' technique avoids this need to scan the actual objects in the heap and thus avoids the associated paging overhead. In this technique we examine the markbits vector directly and look for 'long' runs of zeros which probably identify free space. Once such a long run is found we look at the length of the object at the start of the run to determine the amount of free space to be released. We also take advantage of the fact that objects are not normally allocated from the heap itself but instead from thread local allocation buffers which are allocated from the heap and subsequently used by an individual thread to meet any allocation demands.

1.3.3 Compaction Phase

Once the garbage has been removed from the heap we can consider compacting the resulting set of objects to remove the spaces between them. As compaction can take a long time we try to avoid it if possible and in reality compaction is a rare event. Compaction avoidance is explain in more detail later on.

The process of compaction is made complex by the fact that we no longer have handles in the JVM. Thus if we move any object we must alter all the references which currently exist to it. If one of those references was from a stack and thus we are not certain that it was an object reference, it may have been a float for example, we clearly are unable to move the object. Such objects which are temporarily fixed in position are referred to as 'dosed' (e.g. dozed) in the code and have the 'dosed' bit set in their header word to indicate this fact. Similarly objects can be 'pinned' during some JNI operations. This has the same effect but is 'permanent' until the object is explicitly unpinned by JNI (see stPinObject, stUnpinObject). Objects which remain mobile are compacted in two phases by taking advantage of the fact that the 'mptr' is known to have the low three bits zero and thus we can use one of these to denote the fact that we have 'swapped' it. 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 using an XM interface.

2 Data areas

2.1 An object

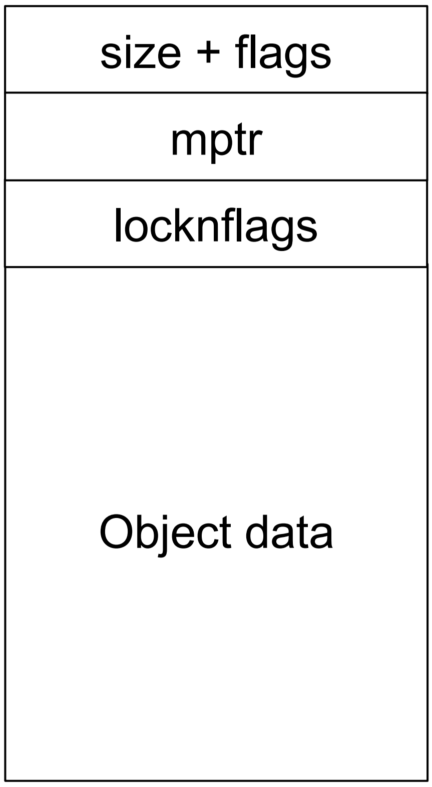

Figure 1 An object

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

- size + flags

The size + flags slot is 4 bytes on 32 bit architecture and 8 bytes on 64 bit architecture. The main purpose of this slot is to contain the length of the object. As all objects start on an 8 byte boundary, and the size is divisible by 8, the bottom 3 bits are not used for the size and so we use them for some flags to indicate different states of the object. Also, as the size of objects is limited the top 2 bits can be used for flags. It should be noted that it is not this slot that is grained on an 8 byte boundary, it is the mptr. The flags are as follows:

❖ Bit 1 is the swapped bit and is only used during compaction. More on this later. Bit 1 is also the NotYetScanned bit and is only used during mark stack overflow. More on this later. Bit 1 is also the multi-pinned bit. This is used to indicate that this object has been pinned multiple times. During a GC the multi-pinned bit will be removed and restored to allow the other uses of this overloaded bit.

❖ Bit 2 is the dosed bit. The dosed bit is set on if the object is referenced from the stack or registers. This means that the object cannot be moved in this GC cycle as we cannot fix up the reference as it may not be a real reference but simply an integer that happens to have the same value as an object on the heap.

❖ 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 as the hash value is the address of the object and we need to maintain this if we move the object.

- mptr

The mptr slot is 4 bytes on 32 bit architecture and 8 bytes on 64 bit architecture. Is is the mptr slot that 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 block, from where we can get to the class block. This is how we can tell what class an object is an instantiation of. The method block 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 the number of array entries in this object.

- locknflags

The locknflags slot is 4 bytes on 32 bit architecture and 8 bytes on 64 bit architecture, although only the lower 4 bytes are used. Its main use is to contain data for the LK component when locking. It also contains some flags of which the following is of interest to us:

  - ❖ Bit 2 is the array flag. If this bit is on the object is an array and the mptr field will contain a count of the number of elements in the array.
  - ❖ Bit 4 is the hashed and moved bit. If this bit is on it tells us 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 collectively know as the header.

2.2 The heap

Figure 2 shows the layout of the heap. The heap is a contiguous piece of storage that is obtained from the operating system at JVM initialisation. 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. This can expand and shrink, which will be discussed later. The size from heapbase to heaptop is controlled by the -Xmx option. If this option is not specified then the default will apply as follows:

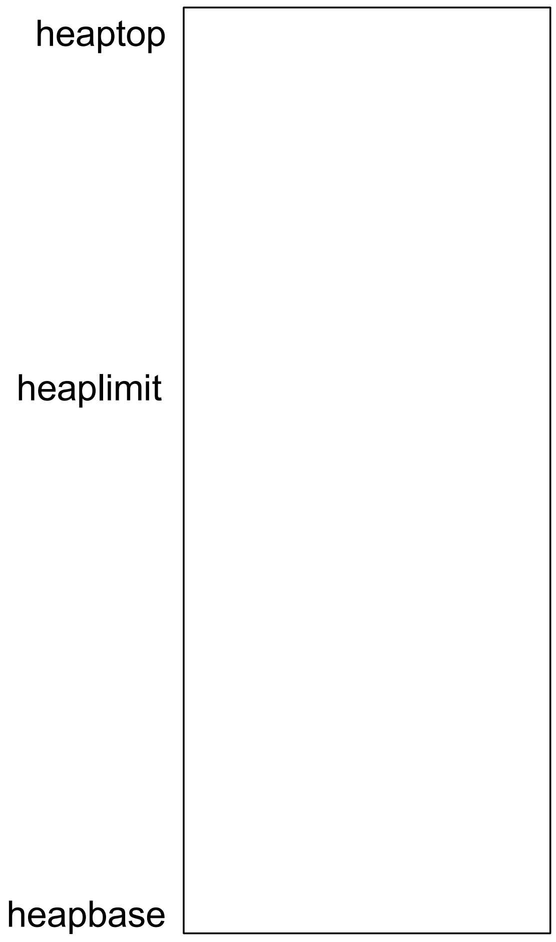

Figure 2 The heap

- 1.2.2 and 1.3.0
  - ❖ Windows - Half the real storage with a minimum of 16M and a maximum of 2G-1.
  - ❖ All other platforms - 64M.
- 1.3.1
  - ❖ Windows - Half the real storage with a minimum of 16M and a maximum of 2G-1.
  - ❖ OS/390 and AIX - 64M.
  - ❖ Linux - Half the real storage with a minimum of 16M and a maximum of 512M-1.

The initial size from heapbase to heaplimit is controlled by the -Xms option. If this option is not specified then the default will apply as follows:

- 1.2.2 and 1.3.0
  - ❖ Windows - 4M.
  - ❖ All other platforms - 1M.
- 1.3.1
  - ❖ Windows, AIX, and Linux - 4M.
  - ❖ OS/390 - 1M

2.2.1 Setting the heap size

For the majority of applications the default settings will work well. The heap will expand until it reaches a steady state and will remain in this state which should give a heap occupancy (the amount of live data on the heap at any given time) of 70%. At this level the frequency of GCs, and the pause time of GCs, should be at an acceptable level.

For some applications the default settings may not give the best results. Here are some problems that may occur, and some suggested actions to take. Verbosegc should be used to help monitor the heap.

- The frequency of GCs is too high until the heap reaches a steady state.
  Use verbosegc to determine the size of the heap at a steady state and 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, but for best performance try to make sure that the heap never pages. The maximum heap size should if possible fit in physical memory.
- At 70% occupancy the frequency of GCs is too great.
  Change the setting of -Xminf. The default is 0.3 which will try to maintain 30% free space by expanding the heap. A setting of 0.4, for example, will increase this free space target to 40%, thereby reducing the frequency of GCs.
- Pause times are too long.
  Try using -Xgcpolicy:optavgpause. This will reduce the pause times and make them more consistent as the heap occupancy rises. There is a cost to pay in throughput which will vary with applications and will be in the region of 5%.

Here are some tips that work well in practice.

- Make sure that the heap never pages (i.e., the maximum heap size must fit in physical memory).
- Avoid finalizers: A developer can never be guaranteed when a finalizer will run, and often they lead to problems. If you do use finalizers try to avoid allocating objects in the finalizer method. A verbosegc trace shows if finalizers are being called.
- Avoid compaction: A verbosegc trace will show if compaction is occurring. Compaction is usually caused by requests for large memory allocations. Analyse requests for large memory allocations and avoid them if possible, if they are large arrays for example then try to split them into smaller pieces.

2.3 Alloc bits and Mark bits

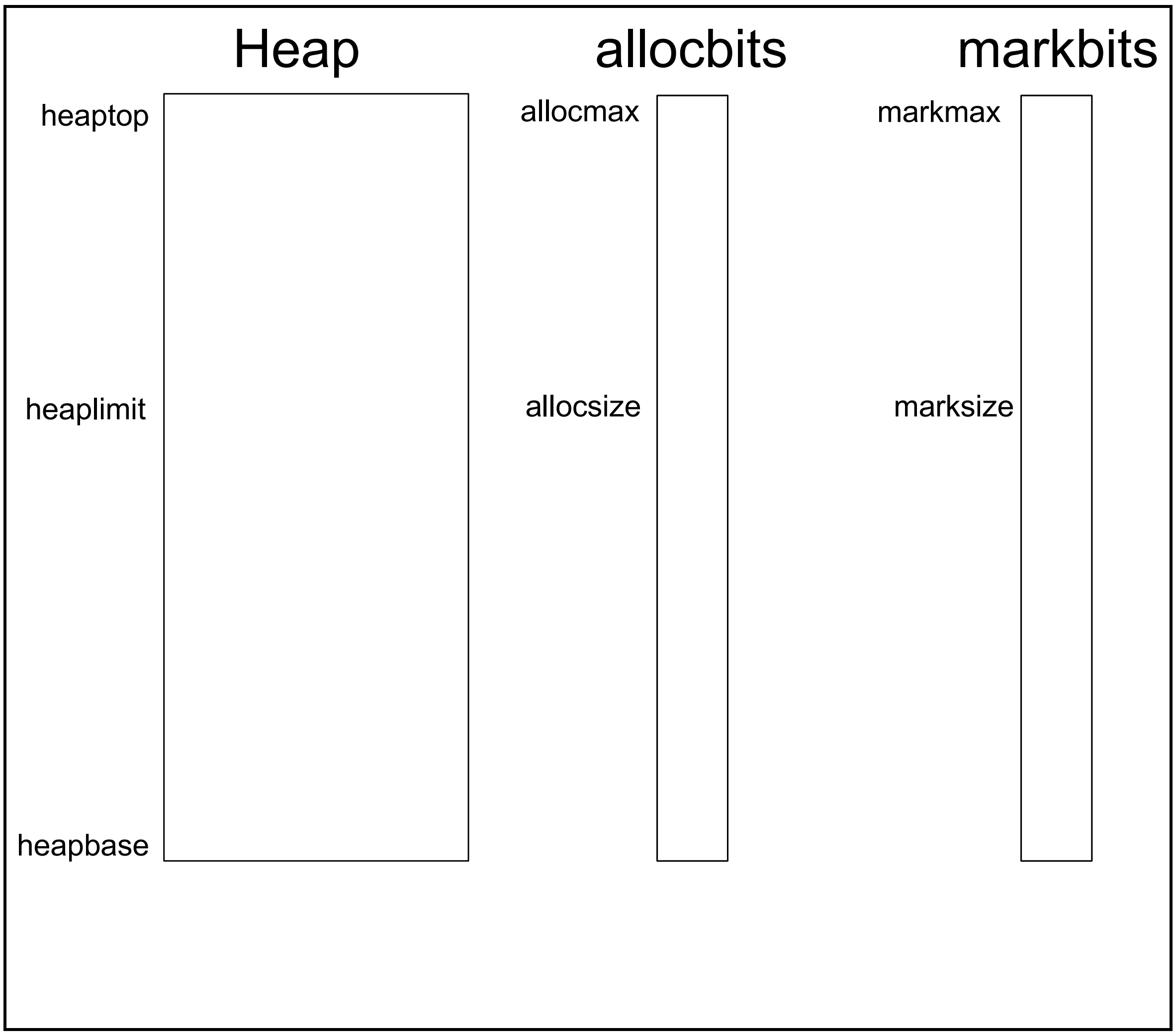

Figure 3 The heap with allocbits and markbits

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

As objects are allocated in the heap a bit is set on in allocbits to indicate the start of the object. This is used to tell us where allocated objects are, but it does not tell us whether the object is alive or not. 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 with allocbits on for both objects. During the mark phase Object 2 was found to be referenced, but Object 1 was not, so we have a markbit on for Object 2. Object 1 will be collected during the sweep phase.

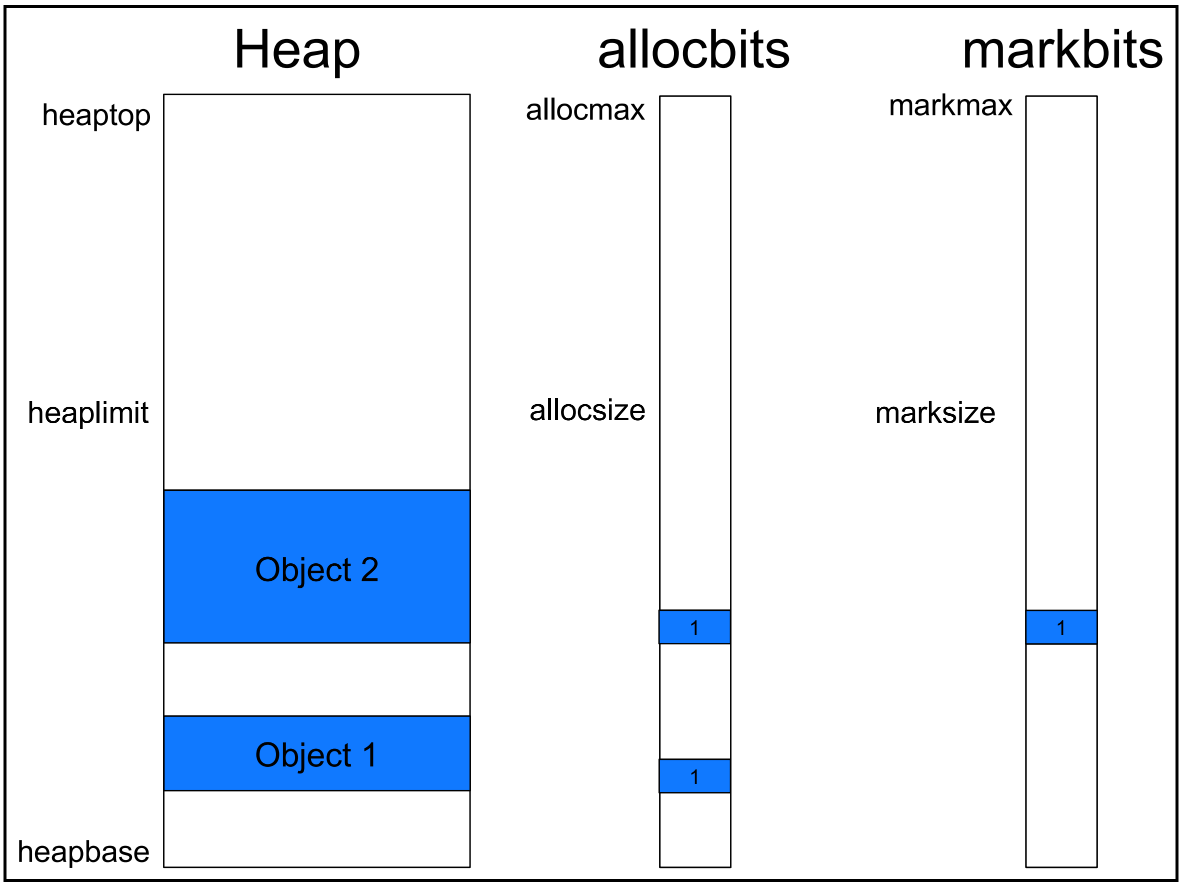

Figure 4 Some objects in the heap

2.4 The System Heap

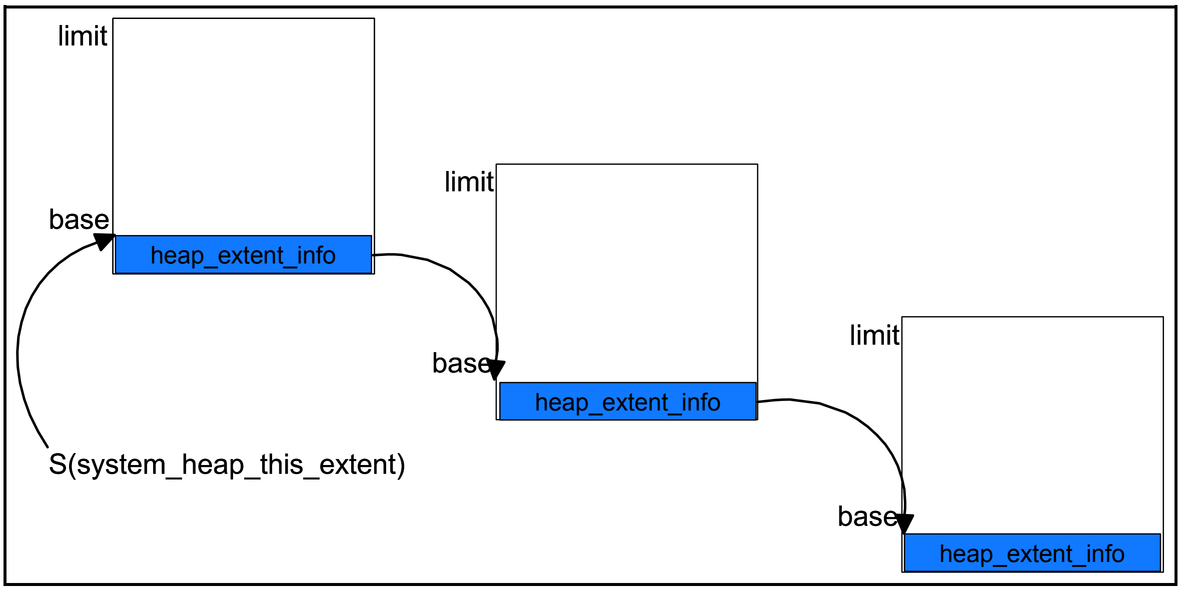

Figure 5 The System Heap

The System Heap was introduced in 1.3.0.

The System Heap contains only objects which have a life-expectancy of the life of the JVM. The objects in this heap are the class objects for system and shareable middleware and application classes. The System Heap is never Garbage Collected as all objects in it will either be 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. As we can see the System Heap is a chain of noncontiguous pieces of storage. The initial size of the System Heap is 128k in 32 bit architecture, and 8M is 64 bit architecture. If this fills then the System Heap will obtain another extent and chain the extents together.

2.5 The free list

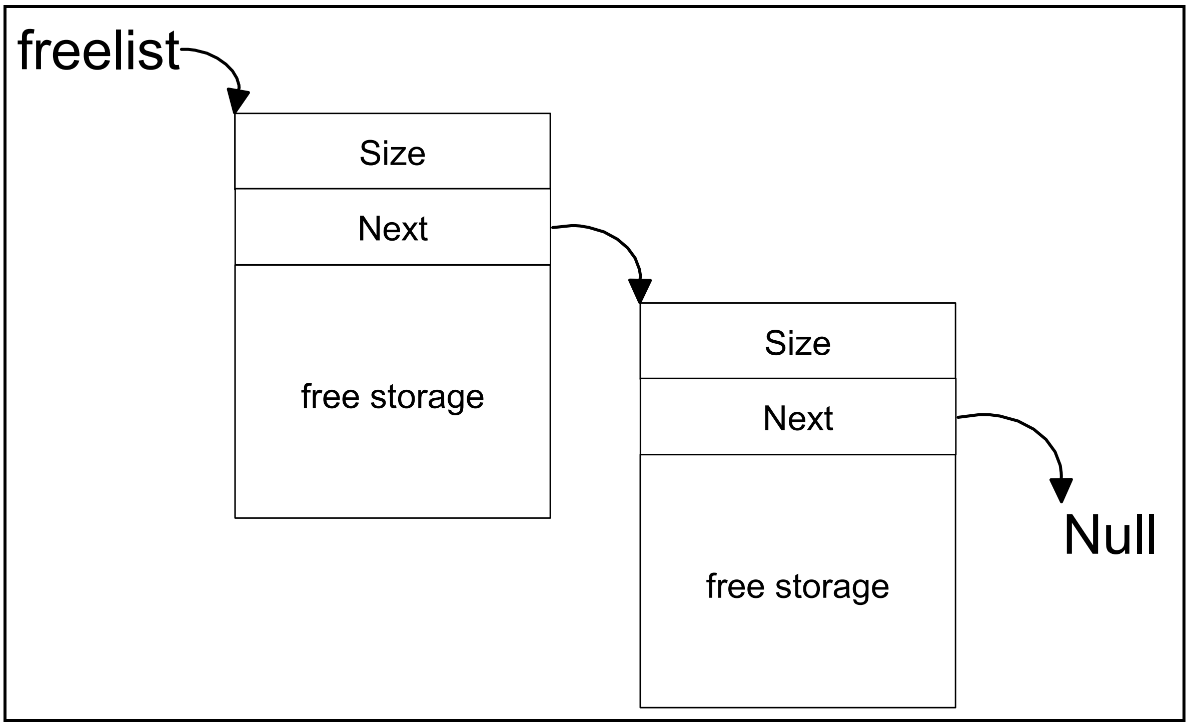

Figure 6 The free list

Figure 6 shows the free list chain. The head of list is in global storage and points to the first free chunk on the heap. Each chunk of free storage has a size field and a pointer which points to the next free chunk. The free chunks are in address order. 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 will not fit into the existing cache (more on caches later). As its name implies heap lock allocation requires a lock and is therefore avoided if possible by using the cache.

```
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
      Take it
      goto Gotit
   else
      manageAllocFailure
      If any error
         goto GetOut
End do
Gotit:
Initialise object
GetOut:
HEAP_UNLOCK
```

Figure 7 Heap Lock allocation

Figure 7 shows some pseudo code for heap lock allocation in 1.3.1. We first check the size of the allocation request and if it is less than 512 or will fit in the existing cache, we will try to allocate using cache allocation. If we do not use cache allocation, or cache allocation failed to find free space, then we will get the HEAP_LOCK. We now search the freelist for a chunk of free storage big enough to satisfy the allocation request. If we find one we take it and initialise the object, returning any remaining free storage to the freelist. It should be noted that if the remaining free storage is less than 512 bytes plus the header size (12 on 32 bit architecture and 24 on 64 bit architecture) we don’t put it on the freelist. These small pieces of storage that are not on the freelist are known as ‘dark matter’, and they will be recovered when the objects adjacent to them become free or when we compact the heap. If we could not find a big enough chunk of free storage we now have an allocation failure and we will perform a garbage collection (GC). If the GC created enough free storage then we will search the freelist again and pick up the free chunk. If GC did not find enough free storage we will return out of memory. The HEAP_LOCK is released after we have either allocated the object or failed to find enough free space.

3.1.1 Hints

Hints was introduced in 1.3.0.

In certain scenarios, in large heaps where the freelist has lots of small free spaces, in an application that is allocating a lot of larger allocations, the heap lock allocation scheme has the problem of always starting at the beginning and searching most of the long list to find a freespace big enough to satisfy an allocation. The quick freelist hint algorithm was introduced to solve this problem.

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

- A search count of the number chunks on the freelist examined before finding a freespace large enough to fit the desired allocation of size n.

- The size of the largest freespace chunk in the freelist preceding the freespace used for allocation is also recorded. In other words, the largest chunk not large enough to satisfy the request.

When a freespace capable of satisfying the allocation is found, if the search count is larger than #define HINTS_SEARCH_MAX_INIT 20 then it is desirable to create a new active hint pointing 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 as chunks are allocated from the freelist.

3.2 Cache allocation

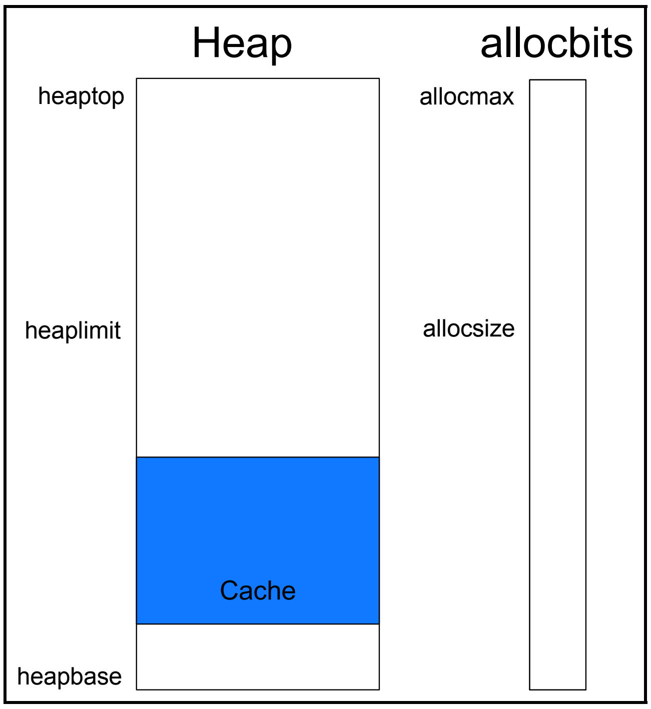

Figure 8 A cache block on the heap

Cache allocation is specifically designed to deliver optimal allocation performance for small objects. Objects are allocated directly from a thread local allocation buffer which 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 and hence it is very efficient. The criteria for using cache allocation has changed over the releases as follows:

- 1.2.2 and 1.3.0 - Use cache allocation if the size of the object is less than 384.
- 1.3.1 - Use cache allocation if the size of the object is less than 512, or the object will fit in the current cache block.

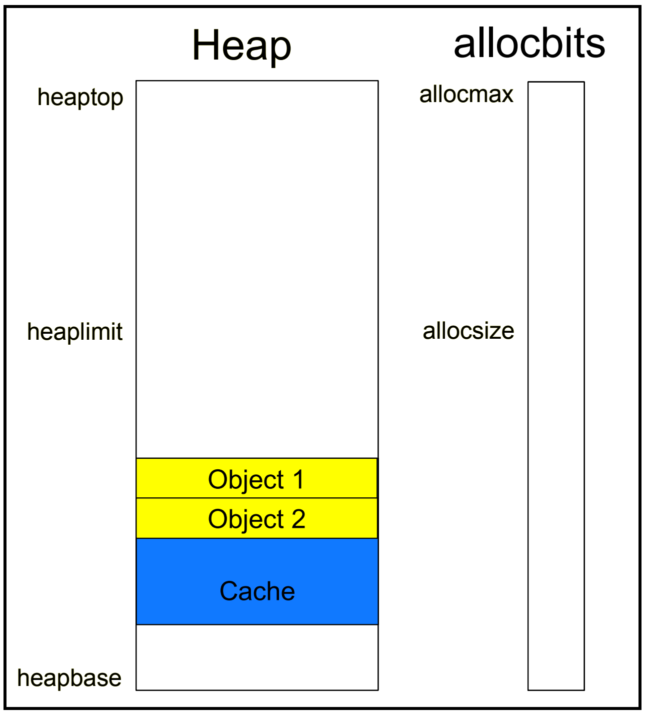

Figure 9 Some objects in a cache

Figure 8 shows a cache block on the heap. The cache block is sometimes called a thread local heap (TLH). When we allocate a TLH for a thread we go through heap locked allocation and reserve 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 on for the TLH. We set allocbits bits for a TLH when the TLH is full or when a GC occurs. To increase the efficiency of allocating a TLH we always take the next free chunk on the free list up to a maximum size which has changed with releases as follows:

- 1.2.2 and 1.3.0 - 6K
- 1.3.1 - Variable from 2K to 164K depending on the usage of the TLH.

Figure 9 shows some objects that have been allocated in the TLH. Here we can see that objects are allocated from the back of the TLH. This can be done more efficiently than allocating from the front. We can also see that there are still no allocbits set. They will be set for all objects in the TLH when the cache is full or when a GC occurs.

4 Garbage Collection

Garbage collection (GC) is performed when we get an allocation failure in heap lock allocation, or if there is a specific call to System.GC. The thread that has the allocation failure or the System.GC call takes control and performs the GC. It first acquires all the locks required for a GC and then suspends all the other threads. GC is then done in three phases; Mark, Sweep, and optionally Compaction. We call this a stop-the-world (STW) collection as all application threads are stopped while we collect the garbage.

4.1 Mark phase

In this phase we are going to mark all the live objects. We have no way of individually identifying unreachable objects so we have to identify all the reachable objects and then we know that everything else is 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 representing the threads, the statics within Java classes, the set of local and global JNI references. All invocations of functions within the JVM itself give rise to a frame on the C stack. This frame may itself contain instances of objects either as a result of an assignment to a local variable or a parameter passed from the caller. All these references are treated equally by the tracing routines. In essence we view the stack of a thread as a set of 4 byte fields (8 bytes in 64 bit architecture) and scan them from the top to the bottom of each of the stacks. We assume that the stacks are 4 byte aligned (8 byte aligned in 64 bit architecture). Each slot is examined to see if it points at an object in the heap. Note that this does not make it necessarily a pointer to an object since it may be just an unlucky combination of bits in a float or integer. Hence when we make this scan of the stacks we must be 'conservative'. Anything which 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 considered to be a pointer to an object if it fulfills the following three criteria:
1. It is grained on an 8 byte boundary.
2. It is within the bounds of the heap.
3. The allocbit is on.

Objects referenced in this way are known as roots and will have their dosed bit set on to indicate that they cannot be moved.

Tracing can now proceed ‘accurately’. What this means is that we can find references within the roots to other objects and as we know that they are real references we will be able to move them during compaction as we can fix up these references.

The tracing process uses a stack which can hold 4K entries. All references that are pushed to the stack are marked at the same time by setting on the relevant markbit. The roots are marked and pushed to the stack and then we start to pop entries off the stack and trace them.

Normal objects (not arrays) are traced by using the mptr to access the classblock which will tell us where references to other objects are to be found in this object. As we find each reference, if it is not already marked, we mark and push it.

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

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

4.1.1 Mark stack overflow

As we have a fixed size for the mark stack it is possible for it to overflow. If this occurs we take the following action:

- Set a global flag to indicate that mark stack overflow has occurred.
- Set the NotYetScanned bit in the object that could not be pushed.

Tracing can then continue with all other objects that could not be pushed having their NotYetScanned bit set. When all tracing is complete we will then walk the heap by starting at the first object and using the size field to navigate to the next object. All objects that we find that have their NotYetScanned bit set are marked and pushed to the mark stack. The NotYetScanned bit is set off and tracing continues as before. It is possible to get another mark stack overflow, in which case we go through the whole process again until all reachable objects are marked.

4.1.2 Parallel Mark

Parallel Mark was introduced in 1.3.0.

Given Bitwise Sweep and Compaction Avoidance, we spend the majority of our GC time marking objects. This leads immediately to developing a parallel version of GC Mark. The goal of Parallel Mark is to not degrade Mark performance on a uni-processor, and to scale typical Mark performance better than 4 on an 8-way machine.

The basic idea is to augment object marking through the addition of helper threads and a facility for sharing work between them. In the original implementation, a single application thread was stolen to perform GC. Parallel Mark still demands the participation of one application thread, which is used as the master coordinating agent. This thread performs very much as it always did, including having the responsibility for scanning C-stacks to identify root pointers for the collection. A platform with *N* processors will also have *N*-1 new helper threads which work along with the master thread to complete the marking phase of GC. The default number of threads can be overridden with the -Xgcthreads*n* parameter. A value of 1 will result in no helper threads. Values of 1 to *N* are accepted.

At a high level, the idea is to provide each marker thread with a local stack and a sharable queue, both of which contain references to objects which are marked but not yet scanned. Threads do the bulk of marking work using their local stacks, synchronising on sharable queues only when work balance requires it. Mark bits are updated using atomic primitives requiring no additional lock As each thread has a Mark Stack that can hold 4K entries, and a Mark Queue that can hold 2K entries, the chances of a Mark Stack Overflow are reduced.

4.1.3 Concurrent Mark

Concurrent mark was introduced in 1.3.1.

Concurrent mark is designed to give reduced and consistent GC pause times as heap sizes increase. It works by starting a concurrent marking phase before the heap is full. In the concurrent phase we scan the roots by asking each thread to scan it's own stack. We then use these roots 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.

As we are marking live objects concurrently with application threads running we have to record any changes to objects we have already traced. This is achieved by using a write barrier that is activated every time we change a reference in an object. The write barrier is required to tell us when an object reference update has taken place so that we know we have to rescan a part of the heap. It is the same write barrier as required by resettable. 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 beginning address of the object that has been updated with the new object reference, is marked with 0x01. A byte is used rather than a bit for two reasons. It is quicker to do a write to a byte than to do bit changes, and we want to be able to use the other bits in future.

A STW collection will be started when one of the following occurs:
- An allocation failure.
- A System.gc.
- Concurrent mark completes all the marking it can do.

We try to start the concurrent mark phase so that it will complete at the same time as the heap is exhausted. This is achieved by constant tuning of the parameters that govern the concurrent mark time.

In the STW phase we again scan all roots, we use the marked cards to see what needs to be retraced, and then we sweep as normal. We guarantee that all objects that were unreachable at the start of the concurrent phase are collected. What we don't guarantee is that objects that become unreachable during the concurrent phase are collected.

The benefits of concurrent mark are reduced and consistent pause times, but there is a cost to pay. Application threads have to ‘pay a tax’ by doing some tracing when they are requesting a heap lock allocation. The amount of tax they have to pay will vary depending on how much idle CPU time there is for the background thread. There is also an overhead for the write barrier.

There is a new parameter to enable concurrent mark.

-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 alleviate those problems at the cost of some throughput when running with the optavgpause option.

4.2 Sweep phase

After the mark phase the markbits vector contains a bit for every reachable object in the heap and must be a subset of the allocbits vector. The task of the sweep phase is to identify the intersection of the allocbits and markbits vectors; objects which have been allocated but are no longer referenced. In the 'bitsweep' technique we examine the markbits vector directly and look for 'long' runs of zeros which probably identifies free space. Once such a long run is found we look at the length of the object at the start of the run to determine the amount of free space to be released. If this amount of free space is greater than 512 (384 in 1.3.0 and earlier) plus the header size, then we put this free chunk on the freelist. The small pieces of storage that are not on the freelist are known as ‘dark matter’, and they will be recovered when the objects adjacent to them become free or when we compact the heap. There is no need to free the individual objects in the free chunk as we know that the whole chunk is free storage. Indeed, when we free a chunk we have 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 reflects the allocated objects on the heap.

4.2.1 Parallel Bitwise Sweep

Parallel Bitwise Sweep was introduced in 1.3.1.

Parallel Bitwise Sweep was introduced for the same reason as Parallel Mark; to improve sweep time by using available processors. We use the same helper threads as Parallel Mark, so the default number of helpers threads is also the same and can be changed with the -Xgcthreads*n* parameter.

The heap is divided into sections with the number of sections being significantly larger than the number of helper threads. The calculation for the number of sections is as follows:

- 32 * the number of helper threads, or the maximum heap size / 16M, which ever 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 and when all sections have been scanned the freelist is built.

4.3 Compaction phase

Once the garbage has been removed from the heap we can consider compacting the resulting set of objects to remove the spaces between them. The process of compaction is made complex by the fact that if we move any object we must alter all the references which currently exist to it. If one of those references was from a stack and thus we are not certain that it was an object reference, it may have been a float for example, we clearly are unable to move the object. Such objects which are temporarily fixed in position are referred to as 'dosed' in the code and have the 'dosed' bit set in their header word to indicate this fact. Similarly objects can be 'pinned' during some JNI operations. This has the same effect but is 'permanent' until the object is explicitly unpinned by JNI. Objects which remain mobile are compacted in two phases by taking advantage of the fact that the 'mptr' is known to have the low three bits zero and thus we can use one of these to denote the fact that we have 'swapped' it. 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 may help in understanding the compaction process.

Think of the heap as a warehouse partly full of pieces of furniture of different sizes. The free space is the gaps between the furniture. The free list contains only gaps above a certain size. Compaction pushes everything in one direction and closes all the gaps. It starts with the object closest to the wall and puts it against the wall, takes the second object in line and puts it against the first, takes the third and puts it against the second, and so on. At the finish, all the furniture is up one end and all the free space up the other. Pinned and dosed objects which cannot be moved complicate the picture but don't change the overall idea.

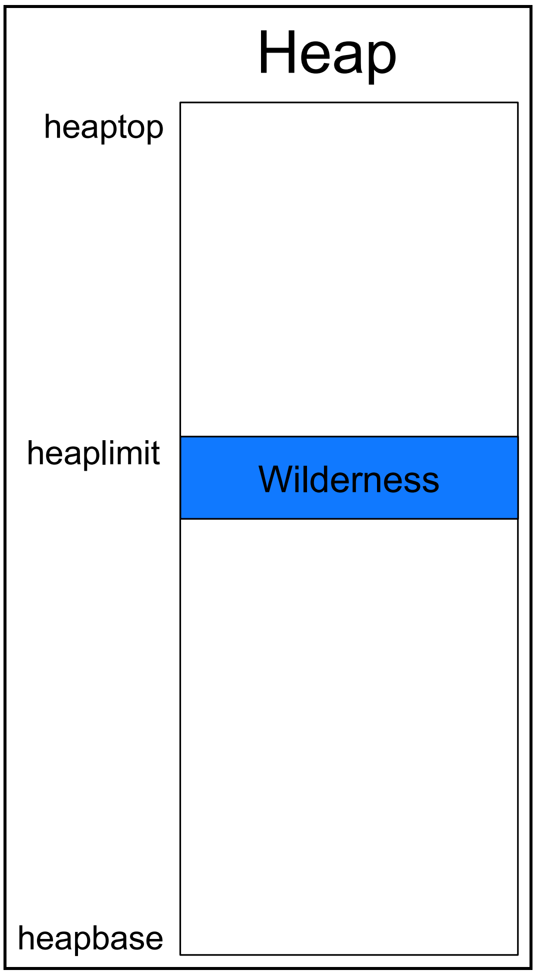

Figure 10 The Wilderness

4.3.1 Compaction Avoidance

Compaction Avoidance seeks to focus on proper object placement, and thereby reduce, and in many cases eliminate, the need for compaction. A key aspect of this approach is a concept called Wilderness Preservation. The essential idea is to attempt to preserve a region of the heap in a pristine state by focusing allocation activity elsewhere. In practice, this is accomplished by establishing a boundary between most of the heap and a reserved wilderness portion. In typical cases, non-compacting GC events are triggered whenever the wilderness is threatened. The wilderness is consumed (eroded) only when necessary to satisfy a large allocation, or when insufficient allocation progress has been made since the previous GC.

Figure 10 shows the Wilderness on the heap. The Wilderness is allocated at the end of the active part of the heap. It’s size is 5% of the active part of the heap, with a maximum of 3M. On heap lock allocation failure, if sufficient allocation progress has been made since the last GC, then a GC is performed. Sufficient progress is defined as at least 30% of the heap being allocated since the last GC. This is the default and can be changed with the -Xminf parameter. If sufficient progress has not been made the allocation is immediately satisfied from the wilderness if possible. Failing this, a "normal" allocation failure occurs. Insufficient progress will have been made if we get an allocation request for a large object which cannot be satisfied, before we have exhausted the free list. This is where the reserved wilderness will be able to satisfy the request, avoiding a GC and a Compaction.

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

- -Xcompactgc has been specified.
- Following the sweep phase there is insufficient free space to satisfy the allocation request.
- A System.GC has been requested and the last allocation failure GC did not compact.
- If at least half of the previously available memory has been consumed by TLH allocations (ensuring an accurate sample) and the average TLH size falls below 1000 bytes.
- If less than 5% of the active heap is free.
- If less than 128K of the active heap is free.

4.4 Reference Objects

In 1.2.2 the whole area of reference handling changed considerably. The aim is to treat all References equally and process them in the same way. Thus we actually create two separate objects on the heap; the object itself and a separate 'reference' object. The 'reference' objects may 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 are immutable; they cannot be made to refer to other than the object they referenced on creation. Objects associated with a finalizer are 'registered' with the Finalizer class on creation. The result is the creation of a FinalReference object which is associated with the 'Finalizer' queue and refers to the object to be finalized.

During GC these Reference objects are handled specially in that the 'referent' field is not traced during the marking phase. Once marking is complete the References are processed in order Soft, Weak, Final and Phantom. Processing of SoftReference objects is specialised in that the ST component is able to 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 basis of most recent usage; in fact 'usage' is measured by the last time the 'get' method was called which can lead to some surprises which are entirely valid. As a Reference object is processed its referent is marked which ensures that when for example a FinalReference is processed for an object which also has a SoftReference the FinalReference will see a marked referent and thus the FinalReference will not be queued for processing. The result is that References will be queued in successive GC cycles.

References to unmarked objects are initially queued to the ReferenceHandler thread in the Reference class. That takes objects off its queue and looks at their individual 'queue' field. If an object is thus associated with a specific queue it is requeued to it for further processing. Thus the FinalReference objects are requeued and eventually their finalize method is run by the Finalizer thread.

4.4.1 JNI Weak References

These provide the same capability as WeakReference objects although the processing is very different. A JNI routine can create a JNI Weak reference to a jobject and later delete that reference. The ST component will clear any weak reference where the referent is unmarked but there is no equivalent of the queuing mechanism. Note that failure to delete a JNI Weak reference will cause 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 which has already been finalized and had a phantom reference queued and processed.

4.5 Heap expansion

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

- The GC did not free sufficient storage to satisfy the allocation request.
- Free space is less than the minimum free space, which can be set by using the -Xminf parameter. The default is 30%.
- More than 13% of the time is being spent in GC, and expanding by the minimum expansion amount (-Xmine) will not result in a heap greater than the maximum percentage of free space (-Xmaxf).

The amount to expand by is calculated as follows:

- If we are expanding because there is less than -Xminf (default 30%) free space then we calculate how much we need to expand by to get -Xminf free space. If this is greater than the maximum expansion amount, which can be set by the -Xmaxe parameter (default of 0 which means no maximum expansion), then we reduce it to -Xmaxe. If this is less than the minimum expansion amount, which can be set by the -Xmine parameter (default of 1M), then we increase it to -Xmine.
- If we are expanding because GC did not free sufficient storage and we are not spending more than 13% in GC, then we expand by the allocation request.
- If we are expanding for any other reason then we calculate how much we need to expand by to get 17.5% free space. We again adjust this as above, depending on -Xmaxe and -Xmine. We also need to make sure we expand by at least the allocation request if GC did not free sufficient storage.

All calculated expansion amounts are rounded up to a 64K boundary on 32 bit architecture, or a 4M boundary on 64 bit architecture.

4.6 Heap shrinkage

Heap shrinkage occurs after GC but when all the threads are still suspended. Shrinkage will not occur if any of the following are true:

- The GC did not free sufficient 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 within the last 3 GCs.
- This is a System.GC and the amount of free space at the beginning of the GC was less than -Xminf (default is 30%) of the live part of the heap.

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

A compaction will occur before the shrink if all of the following are true:

- A compaction was not done on this GC cycle.
- There is not a free chunk at the end of the heap, or the size of the free chunk at the end of the heap is less than 10% of the desired shrinkage amount.
- We didn't shrink and compact on the last GC cycle.

4.7 Resettable JVM

The resettable JVM was introduced in 1.3.0 and is only available on Z/OS.

Documentation on the Resettable JVM can be found in the book SC34-6034-01 entitled "New IBM Technology featuring Persistent Reusable Java Virtual Machines".

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. This is enabled by using the -verbosegc option. It should be noted that verbosegc can, and usually does, change between releases. The following examples are taken from 1.3.1.

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>
```

The above verbosegc output is an example of a System.gc collection, or forced GC. All the lines start with GC(3), which indicates that this was the 3rd GC 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 14ms, resulting in 27% free space in the heap. The figures in parenthesis show the actual number of bytes free, and the total bytes available in the heap. The third line shows the times for the mark, sweep, and compaction phases. In this case there was no compaction so the time is zero. The last line shows the reference objects that were found during this GC, and the threshold for removing soft references. In this case there were no reference objects 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>
```

The above verbosegc output is an example of an allocation failure (AF) collection. An Allocation failure does not mean there has been an error in the code, it is the name we give to the event that triggers when we cannot allocate a large enough chunk from the heap. As can be seen, we have the same four lines as the System.gc verbose output, plus some additional lines. The lines starting with AF[5] are the allocation failure lines and indicate that this was the 5th 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 we are taking to resolve 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 as follows:

- 1 - This action will perform a GC without using the wilderness. It is designed to avoid compactions by keeping the wilderness available for a large allocation request.
- 2 - This action means that we have tried to allocate out of the wilderness and failed.
- 3 - This action means we are going to attempt to expand the heap.
- 4 - This action means we are going to clear any remaining soft references. This is only done if there is less than 12% free space in a fully expanded heap.
- 5 - This action only applies to resettable mode and means we are going to try to steal some space from the transient heap.
- 6 - This is not actually an action. It give a verbosegc message to say we are very low on heap space, or totally out of heap space.

The last line shows how long the AF took. This include the time it took 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>
```

The above verbosegc output is an example of 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. We show how much the mark bits, the alloc bits, and the heap are expanded by, and how much free space is available. In the example we expanded the heap by 1179648 bytes which gave us 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>
```

This is very similar to the verbosegc output for heap expansion. We show how much the mark bits, the alloc bits, and the heap are shrunk by, and how much free space is available. In the example we shrank the heap by 1114112 bytes which gave us 79% free space. One other difference between the verbosegc output for heap expansion and heap shrinkage is the order of the output. This is because expansion happens after all the thread 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>
```

The above verbosegc example shows a compaction. The main difference 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 added. It is possible to have additional bytes because if we move an object that has been hashed we have to store the hash value in the object which may mean increasing the object’s size. The possible reasons for a compaction are as follows:

- 1 - Following the mark and sweep phase there is insufficient free space 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 available.
- 6 - Less than 128K free space available.
- 7 - The -Xcompactgc parameter has been specified.
- 8 - The transient heap has less than 5% free space available.
- 9 - The heap is fragmented. This is different from reason 2 because it uses an additional heuristic to determine if compaction should be done.

4.8.6 verbosegc and concurrent mark kick-off

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

The above two lines are the verbosegc output indicating that the concurrent phase has started. The first line shows how much free space is available, and how much there will be after this heap lock allocation. The *Kickoff* value is the level at which concurrent mark will start. In this example we can see that the expected space is 378884, which is less than the kick-off value of 379406. The second line shows the initial trace rate. In this example it is 8.01 which means that for every byte allocated in a heap lock allocation we must trace 8.01 bytes of live data.

4.8.7 verbosegc and a concurrent mark System.gc collection

```
<GC(3): Concurrent ABORTED. Target=3025707 Traced=0 (0+0) Free=885720>
<GC(3): GC cycle started Tue Mar 19 13:04:02 2002
<GC(3): freed 30008 bytes, 26% free (1125368/4192768), in 15 ms>
  <GC(3): mark: 14 ms, sweep: 1 ms, compact: 0 ms>
  <GC(3): refs: soft 0 (age >= 32), weak 0, final 0, phantom 0>
```

The System.gc verbosegc output with concurrent mark has one additional line, line 1. The state of ABORTED shows that concurrent mark did not complete the initialisation phase and was therefore aborted. This is not an error condition, it just means that we did not get time to initialise concurrent mark before the System.gc. The state could also be HALTED which would indicate that initialisation completed and concurrent mark was interrupted by the System.gc. *Target* is the estimated amount of tracing to be done by the concurrent phase. *Traced* is the actual amount traced which is zero in the above example as the concurrent phase was aborted. *Free* is the amount of free space available when concurrent was aborted. All the above amounts are in bytes.

4.8.8 verbosegc and a concurrent mark AF collection

```
<AF[6]: Allocation Failure. need 4000016 bytes, 86 ms since last AF>
<AF[6]: managing allocation failure, action=2 (1596416/23263744)>
<GC(6): Concurrent HALTED (state=128). Target=21435225 Traced=3625624
(3625624+0) Free=1587376>
<GC(6): No Dirty Cards cleaned (Factor 0.130)>
<GC(6): GC cycle started Tue Mar 19 13:42:52 2002
<GC(6): freed 16834536 bytes, 79% free (18430952/23263744), in 23 ms>
  <GC(6): mark: 20 ms, sweep: 3 ms, compact: 0 ms>
  <GC(6): In mark: Final dirty Cards scan: 10 ms 2211 cards (total:36 ms)
  <GC(6): refs: soft 0 (age >= 1), weak 0, final 0, phantom 0>
<AF[6]: completed in 25 ms>
```

The above is an example of an AF collection when concurrent mark is running. Line 3 shows that concurrent was HALTED, and what state it was in at the time. The possible states are:

- 64 - Concurrent collection was tracing
- 128 - Concurrent collection was tracing and cleaning dirty cards

The *Traced* figures in parenthesis show how much is traced by the application threads and how much is traced by the background thread. In this example there was no background thread tracing which was probably due to this being run on a busy uniprocessor.

Line 4 shows that no dirty cards had been cleaned concurrently when the AF occurred. *Factor* is used to estimate the number of dirty cards with respect to the heap size. Line 8 shows the time it took to do final card cleaning and how many cards were cleaned. In the above example it took 10ms to clean 2211 cards. The *total* time is the total card cleaning time for this JVM.

4.8.9 verbosegc and a concurrent mark collection

```
<CON[2]: Concurrent collection, (905720/22100560) (1163184/1163184), 115 ms since
last CON>
<GC(8): Concurrent EXHAUSTED. Target=15506720 Traced=4300312 (4300312+0)
Free=905720>
<GC(8): Cards cleaning Done. cleaned:1922 (0 skipped). estimation 5468 (Factor 0.120)>
<GC(8): GC cycle started Tue Mar 19 13:42:53 2002
<GC(8): freed 16554096 bytes, 80% free (18623000/23263744), in 6 ms>
  <GC(8): mark: 3 ms, sweep: 3 ms, compact: 0 ms>
  <GC(8): In mark: Final dirty Cards scan: 0 ms 258 cards (total:37 ms)
  <GC(8): refs: soft 0 (age >= 4), weak 0, final 0, phantom 0>
<CON[2]: completed in 7 ms>
```

The above verbosegc output is an example of a collection initiated by concurrent mark. Instead of AF at the start of the lines we have CON. The first line shows how much space is used and available in the main part of the heap and the wilderness. It also shows how long since the last collection. The second line shows that concurrent is EXHAUSTED which means that there was no more work for concurrent to do so it initiated the GC. We have already discussed the other fields on this line. Line 3 shows that concurrent card cleaning was done; *cleaned* is the number of dirty cards that required cleaning and were cleaned, *estimation* is the number of dirty cards found.

4.8.10 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>
```

When running resettable we have a middleware heap and a transient heap. The verbosegc for the transient heap is slightly different. In the above example we can see the we use TH_AF instead of AF. The policy when running resettable is to expand the transient heap when an allocation failure occurs, rather than GC it. 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>
```

This example shows what happens when the expansion is not successful. Here we have to do a GC and we can see how much space is freed from each of the heaps.

5 JVMSet

The JVMSet JVM was introduced in 1.3.1 and is only available on Z/OS.

Documentation on the JVMSet JVM can be found in the book SC34-6034-01 entitled "New IBM Technology featuring Persistent Reusable Java Virtual Machines".

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

6 Acknowledgments

In writing this document I have used text from many different sources, written by different people. I would like to acknowledge input from the following people:

- Bob Dimpsey
- Luis Ostdiek
- Matthew Peters
- John Rankin
- Martin Trotter

7 Messages

JVMST010: Cannot allocate memory for ACS area
Explanation: There was insufficient virtual storage available to allocate the ACS heap. The call to sharedMemoryAlloc() failed. This can happen during initialisation 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 persists contact your IBM service representative.

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

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

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

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

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

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

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

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

JVMST019: Cannot allocate memory in allocateToMiddlewareHeap
Explanation: An error occurred trying to commit memory for the Java heap. The call to sysCommitMem() failed. This can happen during initialisation or 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 trying to commit memory for the transient heap. The call to sysCommitMem() failed. This can happen during initialisation 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: There was insufficient storage available in the Java heap to allocate the stackEnd object. The call to allocMiddlewareArray() failed. This can only happen during initialisation.
System Action: The JVM is terminated.
User Response: Allocate more Java heap storage by increasing the -Xmx value. If the problem persists contact your IBM service representative.

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

JVMST024: Cannot allocate memory in concurrentInit(base-Malloc)
Explanation: There was insufficient virtual storage available to allocate the concurrent data structures. The call to sysMalloc() failed. This can only happen during initialisation.
System Action: The JVM is terminated.
User Response: Allocate more virtual storage to the JVM region. If the problem persists contact your IBM service representative.

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

JVMST027: Cannot allocate memory for System Heap area in allocateSystemHeapMemory
Explanation: There was insufficient virtual storage available to allocate storage for the system heap. The call to sharedMemoryAlloc() failed. This can happen during initialisation 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 persists contact your IBM service representative.

JVMST028: Cannot commit memory in RASinitShadowHeap
Explanation: An error occurred trying to commit memory for the shadow heap. The call to sysCommitMem() failed. This can only happen during initialisation when using the trace option -Xtgc8388608.
System Action: The JVM is terminated.
User Response: Contact your IBM service representative.

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

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

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

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

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

JVMST036: Cannot allocate memory in initializeMiddlewareHeap (allocbits)
Explanation: An error occurred trying to commit memory for the allocbits vector. The call to sysCommitMem() failed. This can only happen during initialisation when not running with -Xresettable.
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 trying to create shared memory. The call to xhpiSharedMemoryCreate() failed. This can only happen during initialisation when running with -Xjvmset.
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 persists contact your IBM service representative.

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

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

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

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

JVMST046: Cannot allocate storage for standalone jab in initializeSharedMemory
Explanation: There was insufficient virtual storage available to allocate the JAB. The call to sysCalloc() failed. This can only happen during initialisation when not running with -Xjvmset.
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 persists contact your IBM service representative.

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

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

JVMST049: Worker and Master JVM versions differ
Worker JVM version is <*version*> build type is <*build*>
Master JVM version is <*version*> build type is <*build*>
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 only happen during initialisation when running with -Xjvmset.
System Action: A return code of JNI_ERR will be 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 persists contact your IBM service representative.

JVMST050: Cannot allocate memory for initial Java heap
Explanation: An error occurred trying to query memory availability. The call to DosQuerySysInfo() failed. This can only happen during initialisation 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: There was insufficient virtual storage available to allocate the Java heap. The call to sysMapMem() failed. This can only happen during initialisation on OS/2.
System Action: The JVM is terminated.
User Response: Allocate more virtual storage to the JVM region. If the problem persists contact your IBM service representative.

JVMST052: Cannot allocate memory for initial Java heap
Explanation: There was insufficient virtual storage available to allocate the Java heap. The call to sysMapMem() failed. This can only happen during initialisation 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 persists contact your IBM service representative.

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

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

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

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

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

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

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

8 Command Line Parameters

The following list contains all the command line parameters available up to release 1.3.1.

-verbosegc
-verbose:gc
Prints garbage collection information. The format for the generated information is not architected and hence varies from platform to platform and release to release.

-Xcompactgc
Compact the heap every GC cycle. The default is false.

-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 alleviate those problems at the cost of some throughput when running with the optavgpause option.

-Xgcthreads
Sets the number of total number of threads 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 only available in the resettable JVM. Classes in this heap exist for the lifetime of the JVM. They are reset during a ResetJavaVM(), and so are serially reusable by applications running in the JVM. There is only one application-class system heap per Persistent Reusable JVM. In non-resettable mode, this option is ignored. Example: **-Xinitacsh256k** Default: 128K on 32 bit architecture, and 8M on 64 bit architecture.

-Xinitsh<*size*>
Sets the initial size of the system heap. Classes 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: 128K on 32 bit architecture, and 8M on 64 bit architecture.

-Xinitth<*size*>
Sets the initial size of the transient heap within the nonsystem heap. This option is only available in the resettable JVM. If this is not specified and **-Xms** is, the initial size is taken to be half the **-Xms** value. If **-Xms** is not specified, a value of half the platform-dependent default value is used. Example: **-Xinitth2M** Default: 1M/2 = 512K

-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 1MB. When JNI_CreateJavaVM()returns successfully, the "extrainfo" field of the JavaVMOption contains the token to be passed to each worker. An attempt to create two master JVMs with the same token will fail.

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 returned on the **-Xjvmset** option 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 both the middleware and transient heaps.

-Xmaxf<*number*>

This is a floating point number between 0 and 1 which 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. With a value of 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 1M. In resettable mode, this option sets a minimum expansion size of <size>/2 for both the middleware and transient heaps.

-Xminf<*number*>

This is a floating point number between 0 and 1 which 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 .3 (that is 30%).

-Xms<*size*>

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

- 1.2.2 and 1.3.0
  - ❖ Windows - 4M.
  - ❖ All other platforms - 1M.
- 1.3.1
  - ❖ Windows, AIX, and Linux - 4M.
  - ❖ OS/390 - 1M

-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 will default as follows:

- 1.2.2 and 1.3.0
  - ❖ Windows - Half the real storage with a minimum of 16M and a maximum of 2G-1.
  - ❖ All other platforms - 64M.
- 1.3.1
  - ❖ Windows - Half the real storage with a minimum of 16M and a maximum of 2G-1.
  - ❖ OS/390 and AIX - 64M.
  - ❖ Linux - Half the real storage with a minimum of 16M and a maximum of 512M-1.

-Xnocompactgc

Never compact the heap. Default is “false”.

-Xresettable

Specifies that this instance of the JVM will be capable of supporting the resettable JVM.