The GC Interface in the EVM[1]

Derek White and Alex Garthwaite

The GC Interface in the EVM[1]

Derek White and Alex Garthwaite

SML TR-98-67 December 1998

Abstract:

This document describes how to write a garbage collector (GC) for the EVM. It assumes that the reader has a good understanding of garbage collection issues and some familiarity with the Java™ language.

The EVM is part of a research project at Sun Labs. The interfaces described in this document are under development and are guaranteed to change. In fact, the purpose of this document is to solicit feedback to improve the interfaces described herein. As a result, specific product plans should not be based on this document; everything is expected to change.

[1]EVM, the Java virtual machine known previously as ExactVM, is embedded in Sun's Java 2 SDK *Production* Release for Solaris™, available at http://www.sun.com/solaris/java/.

M/S MTV29-01
901 San Antonio Road
Palo Alto, CA 94303-4900

email address:
derek.white@east.sun.com

© 1998 Sun Microsystems, Inc. All rights reserved. The SML Technical Report Series is published by Sun Microsystems Laboratories, of Sun Microsystems, Inc. Printed in U.S.A.

Unlimited copying without fee is permitted provided that the copies are not made nor distributed for direct commercial advantage, and credit to the source is given. Otherwise, no part of this work covered by copyright hereon may be reproduced in any form or by any means graphic, electronic, or mechanical, including photocopying, recording, taping, or storage in an information retrieval system, without the prior written permission of the copyright owner.

TRADEMARKS
Sun, Sun Microsystems, the Sun logo, Java, and Solaris are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

For information regarding the SML Technical Report Series, contact Jeanie Treichel, Editor-in-Chief <jeanie.treichel@eng.sun.com>.

Table of Contents

1. Introduction 1
    - 1.1 Organization of the EVM 2
    - 1.2 Java Language Issues 3
    - 1.3 Implementation Issues 5
2. Collector Cookbook 9
    - 2.1 Preliminary Information 9
    - 2.2 Core Collector Routines 11
    - 2.3 Read/Write Barriers 20
    - 2.4 Supporting Collector Routines 22
3. Reference Section 25
    - 3.1 Header Files 25
    - 3.2 Collector Configuration 25
    - 3.3 Sizes and Casts 26
    - 3.4 Object Header Information 27
    - 3.5 Class Information 28
    - 3.6 Marking Macros 29
    - 3.7 Root-processing and Iteration Support 30
    - 3.8 Collection-related Support 33
    - 3.9 Fast-allocation Support 34
    - 3.10 Locking Support 34
    - 3.11 Miscellaneous Support 35
    - 3.12 Debugging and Verification 36
    - 3.13 Memory Allocation Support 37
4. Open Issues and Future Directions 41
5. More Information and Feedback 43
    - 5.1 References and Further Reading 43
    - 5.2 Credits 43
    - 5.3 Feedback 43
6. About the Authors 45

The GC Interface in the EVM[1]

Derek White and Alex Garthwaite
Sun Microsystems Laboratories
901 San Antonio Road
Palo Alto, California 94303

1. Introduction

This document describes how to write a garbage collector (GC) for the EVM. It assumes that the reader has a good understanding of garbage collection issues and some familiarity with the Java™ programming language.

The EVM is part of a research project at Sun Labs. The interfaces described in this document are under development and are guaranteed to change. In fact, the purpose of this document is to solicit feedback to improve the interfaces described herein. As a result, specific product plans should not be based on this document; everything is expected to change.

This section defines some terms used in the document, describes how the EVM is organized with respect to memory management, and describes how the Java programming language and the Java virtual machine (JVM) implementation affect the design of collectors.

A Java virtual machine executes abstract instructions in one or more threads of control. There may be other threads running in the system, but they are not relevant to this document. A JVM may execute bytecodes by interpreting them directly, or by first compiling the bytecodes into a native form and executing the native code ("compiled code"). Both forms of the code executed are known as "Java code." Each thread has two stacks: a "Java stack" to run bytecodes and a "native stack" to run native code. The JVM contains explicit support for objects, which are dynamically allocated out of a garbage collected heap ("Java heap"). The JVM also supports manipulating variables associated with classes ("Java static variables").

---

1. EVM, the Java virtual machine known previously as ExactVM, is embedded in Sun's Java 2 SDK *Production* Release for Solaris™, available at http://www.sun.com/solaris/java/

The EVM is a Java runtime environment (JRE) optimized for Solaris and server applications. Particular attention has been paid to scalability and reliability issues, including garbage collection. The EVM has an interface that makes it simple to implement and to use different garbage collectors. A collector for the EVM must be chosen at build time, to allow the collector to supply macros and inline functions that the rest of the system can use.

The GC interface has been used to implement mark and sweep, copying, compacting, generational, and concurrent garbage collectors. The interface will not handle simple reference counting collectors due to limited information about Java stacks, but will support deferred reference counting collectors. Work still needs to be done to support weak references with concurrent collectors in a GC independent way.

1.1 Organization of the EVM

A Java runtime environment consists of class files, shared libraries containing the native methods for the class files, and a JVM shared library proper. A JRE is the smallest set of executables and files that constitute the standard Java platform.

A JVM can load and execute class files, and native methods can call JVM code via the Java native interface (JNI) [2] or the private JVM interface. Native methods have no direct access to the Java heap.

For the purpose of describing the memory system and the collector interface, we can further divide the EVM into normal Java execution (interpreter, JIT, class loading, JNI/JVM interfaces, etc.), and the memory system and collector:

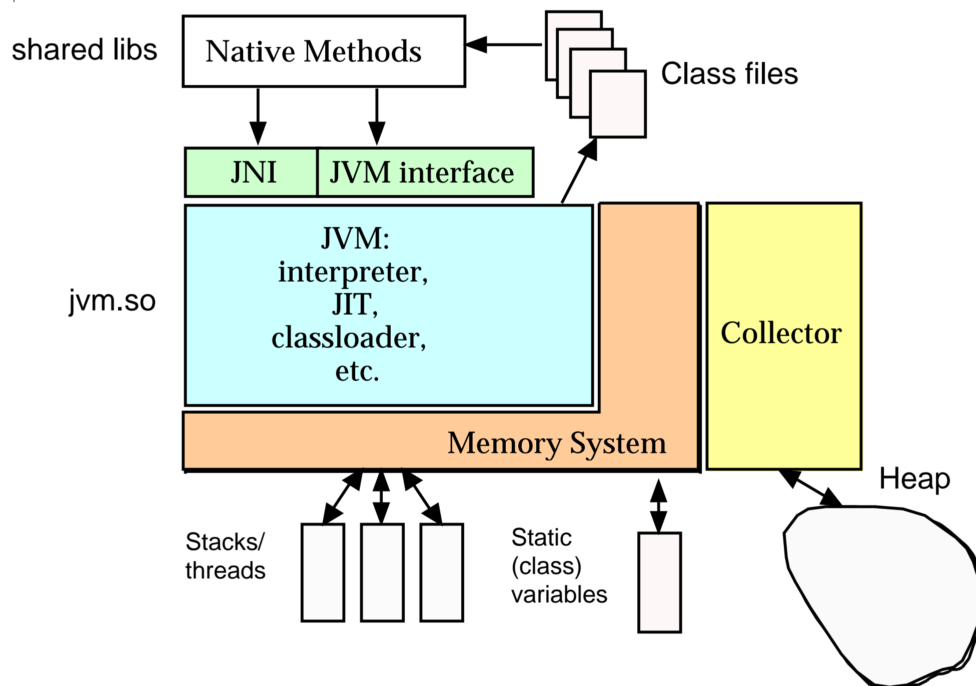


FIGURE 1. JVM Organization

The memory system sits between the collector and the rest of the JVM. It provides an internal interface to the JVM for object allocation and garbage collection (`gc.h`), as well as access to objects in the heap (`memsys.h`). The memory system also has a private interface (`gc/gcimpl.h`) that supports all collectors. The private interface provides routines to iterate over all "roots" in the JVM, suspend and resume Java threads, and handle most details related to locking, class unloading, and weak references.

A collector has to implement only the details of a specific GC algorithm. It does this by defining the `specific`*XXX* functions in the abstract collector interface, and optionally defining read/write barriers. The barriers ensure that no code can access or modify the heap without the collector knowing about it. The abstract collector interface is private to the memory system. For the most part, the collector has access to the rest of the JVM only through the memory system. Object layout and class information (defined in JVM headers) are notable exceptions to this.

1.2 Java Language Issues

The Java language specifies some language features that impact memory management.

1.2.1 Object Issues

The Java programming language is mostly object-oriented. Values in the language can be references to objects or they can be primitive values (`boolean`, `byte`, `short`, `int`, `long`, `char`, `float`, or `double`). The language specifies the precision of these values explicitly (often 32-bit or 64-bit signed numbers), which precludes using tagging on most hardware. To deduce which values are references and which are primitives, the EVM uses the fact that the Java language is strongly typed and that objects carry runtime type information.

1.2.2 Finalization

A class may define a `finalize()` method. If an object has a `finalize()` method, the method will be called sometime after the collector has determined that the object is unreachable, but before the object is collected. See section 12.6 in the Java Language Specification (JLS) [1] and section 2.16.7 in the Java Virtual Machine Specification (JVM spec) [5] for more details. In JVMs based on the Java Platform 1.2 API [3], finalization is implemented using weak references.

1.2.3 Weak References

New to Java Platform 1.2 are public weak reference classes defined in the `java.lang.ref` package. This introductory section gives an overview of the language feature. "Collecting Weak References," page 14, provides more information and examples. The basic idea is that each weak reference object has a special field called the `referent`, which is a weak pointer to some object. Weak reference objects have another field that refers to the queue that the weak reference should be placed on for further processing. When the memory system notices that there are no stronger pointers to a weak reference object's referent, it places the weak reference object onto its queue. Various Java threads are waiting on the queues for objects to process. There are built-in threads to handle finalization and `SoftReference` objects, but users may define their own queue and thread to handle weak reference objects.

There are several predefined references classes: `Reference` (the abstract superclass of the following classes), `SoftReference`, `WeakReference`, and `PhantomReference`. Note that in this document, the term "weak reference" is used to describe all subclasses of `java.lang.ref.Reference`. The term `WeakReference` refers to the specific class `java.lang.ref.WeakReference`. There is also a private subclass of `Reference` called `FinalReference` which is used to implement finalization.

Note that weak references in other languages usually require the collector to break the weak link to the dying referent. But weak references in the Java language often preserve the weak link. The referent of a `PhantomReference` or a `FinalReference` is not cleared until Java code calls the `clear()` method on it. This means that a referent may still be reachable after processing if the weak reference object stays reachable, or if processing the weak reference object stores the referent somewhere reachable. The built-in processing code for `FinalReference`s calls the `clear()` method and drops all references to the processed `FinalReference`. The `referent` fields of `SoftReference`s and `WeakReference`s are cleared by the memory system before enqueuing the weak reference object.

Some weak reference objects are stronger than others. This means that if there are both "stronger" weak reference objects and "weaker" weak reference objects that share the same dying referent, only the strongest weak reference objects will be processed in a particular collection. All weak reference objects of the same strength that share the same dying referent will be queued for processing in the same GC. From strongest to weakest, weak reference objects are ordered as follows:

1. `SoftReference` and subclasses. These weak reference objects are not processed as aggressively as others, and are used to create caches that are cleared as a last resort. The referent fields of these weak reference objects are cleared before the reference objects are enqueued for processing.
2. `WeakReference` and subclasses. The referent fields of these weak reference objects are cleared before the reference objects are enqueued for processing.
3. `FinalReference`. These weak reference objects are enqueued in a finalization queue. Another Java thread will call the `finalize()` method on the referent of each `FinalReference` object in the queue.
4. `PhantomReference` and subclasses. These weak reference objects do not have access to the referent, but since they are processed last, they can be used to trigger "postmortem finalization" actions. Because users don't have access to the `PhantomReference`'s referent field, users usually create subclasses of `PhantomReference` that contain interesting state.

The class `Reference` is both abstract and has package-private constructors, so there can be no weak reference objects in the system other than those described above.

It may take a while for a dying referent to be collected: it will only be collected in the GC after the weakest reference object that refers to the dying referent has been enqueued and processed by its queue, and only if the Java code processing the queue has not stored the pointer to the dying referent somewhere. In a generational collector, the referent and all weak reference objects that refer to it must reside in the same generation before the weak reference objects will be processed (otherwise, the cross generation references from the weak reference objects to the referent look like

strong references). See Java Software's documentation [3] [6] and "Collecting Weak References," page 14, in the Cookbook section for more information.

1.2.4 Class Unloading

A class or interface may be unloaded if and only if its class loader is unreachable (the definition of unreachable is given in section 12.6 in the JLS [1]). Classes loaded by the default system loader may not be unloaded. This rule implies that a JVM cannot unload a class if any instances (direct or indirect) of the class are reachable or if the object representing the class is reachable. In the EVM, class unloading is only possible during a full GC. Generational collectors in the EVM need to treat classes as if they are part of the oldest generation. See section 12.8 in the JLS [1] and section 2.16.8 in the JVM spec [5] for more details.

1.3 Implementation Issues

A collector needs to be designed with knowledge of certain implementation details of the EVM and its memory system. The memory system takes care of many issues that are common to all garbage collectors, but it requires some cooperation.

1.3.1 Object Layout

All objects begin with a header. The header contains a pointer to class information and a word that hold the object's hash code, lock fields, and bits that can hold age information (for generational collectors). The rest of an object contains zero or more 32-bit words to hold object fields, or a word holding the array length followed by array elements. `long` and `double` object fields and array elements are not double word aligned.

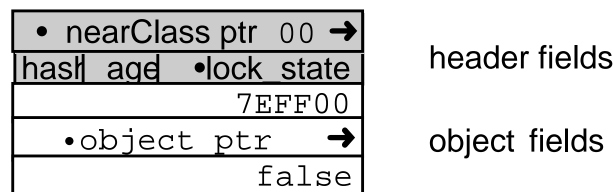


FIGURE 2. An Example of a 3-field Object's Layout

An object's class information is stored in a `NearClass` structure. A `NearClass` stores the basic information needed to call a virtual method or perform GC. This includes the object's method table, size, and "instance layout map." The `NearClass` also contains a pointer to a more complete representation of a class called the `Class` structure. Neither the `NearClass` nor `Class` structures are objects. Note that an object does not have an ordinary reference to its `java.lang.Class` instance, so simply scanning an object's reference fields will not scan its class (an additional operation is required). See "Object Header Information," page 27, for more details on header words.

All objects are at least two words long, and are minimally aligned at a word boundary (4-byte alignment). This means that a collector can count on the low-order bits (at least two) of an object pointer being zero, and can set those bits during garbage collection. A collector can use the

`OBJECT_ALIGNMENT_MASK` to get access to those bits. Similarly, the `NearClass` is aligned, so a collector can use low-order bits of the `nearClass` pointer field in an object's header to record things like mark bits or indirection bits during collection. A collector can use the `HEADER_ALIGNMENT_MASK` to get access to those bits.

There are predefined macros that test, mask, and include some of these low-order bits which are described in "Marking Macros," page 29.

1.3.2 Roots and Memory Consistency

The collector needs to know the locations of all the pointers to objects in the system: in Java stacks, Java static variables, JVM internal global and local roots, etc. But recording where all pointers are in the system at every instant would take too much time and space. So the JVM arranges execution such that each thread will know where its pointers are at frequent intervals. If a thread is at a point where all pointers in the thread's stack are known, it is called "consistent." Otherwise, it is called "inconsistent." A thread's stack is separated into a native stack provided by the OS where VM code, compiled methods, and native methods run, and a Java stack managed by the JVM where interpreted methods run.

When some part of the JVM decides to do a garbage collection, the memory system suspends all threads. Then it resumes all inconsistent threads after arranging it so that each thread will suspend itself when it becomes consistent. When all of the threads are stopped consistently, the memory system calls the collector to do actual collection. Inconsistent threads suspended themselves by either polling a variable and suspending explicitly when the thread becomes consistent, or by the memory system inserting trap instructions at the points where threads will become consistent.

The JVM code itself (such as class loading) normally is consistent when it runs. Most JVM code accesses objects using the low level native interface (LLNI). The LLNI is a layer over the memory system that allows the JVM to refer to objects using LLNI handles (pointers to "local roots" which point to an object), and provides many accessor operations. When an LLNI operation needs to dereference a handle, it becomes inconsistent temporarily. The JVM code has to register all local roots so the memory system will know where to find direct pointers to objects.

Java code (either running in the interpreter loop or as compiled code) normally is inconsistent when it is executed, so it can use pointers to objects without informing the memory system. But Java code must become consistent at frequent intervals or threads that need to allocate objects may be arbitrarily delayed. So the interpreter and JIT compiler generate "stack maps" for every method call and backward branch, which describe where the pointers are for a particular method and program counter (pc).

Native methods almost always run while consistent. The only JNI function that makes a thread inconsistent is `GetPrimitiveArrayCritical`. The only JNI functions that are callable while a thread is inconsistent are more calls to `GetPrimitiveArrayCritical` and `ReleasePrimitive-ArrayCritical`.

1.3.3 Locking

In order to support multiple threads, the memory system uses a "heap lock" around any allocation from the global heap and around GC itself. Local allocation buffers (LABs), described in "Allocation," page 11, reduce contention on this lock. The memory system also depends on various JVM data structures that are protected by locks, such as the table of all classes and the thread queue. So the memory system takes care of acquiring and releasing these locks around garbage collection.

If a collector needs other locks around collection, the collector can create and register them with the memory system. The memory system takes care of acquiring and releasing these "GC locks" in the correct order. See "Locking Support," page 34, for more information.

1.3.4 Differences Between EVM and JDK 1.x Classic

Some readers may have knowledge of the Java Development Kit (JDK) 1.x classic VM architecture. The EVM has many similarities to the JDK classic VM architecture, but some differences are worth pointing out:

- Java code runs without handles. All object-to-object references use direct pointers. C code usually refers to objects via JNI or LLNI handles, however.
- The Java heap (the memory managed by the memory system) contains only objects. There are no C structures in the Java heap. This means that the `NearClass` and `Class` structures (similar to `ClassClass` in JDK classic) are in the C heap, while the corresponding `java.lang.Class` instance is in the Java heap.
- Array objects have object headers like other objects do. In particular, the headers have a pointer to a `NearClass` that describes the "Array of *Foo*" class, where *Foo* is the class of the array elements (and there is not any information lurking at the end of the array).
- Object locking does not use OS-level monitors, but an optimized light-weight locking protocol.
- Because references in the heap and much of the VM are direct references and because garbage collection may cause objects to be relocated, work must be done to update these references to the objects' new locations.

2. Collector Cookbook

A collector must support five core routines, and several supporting routines. In addition, it may supply read and write barriers that the rest of the JVM must use. This section describes the basic header files and types that a collector deals with, and the routines and barriers that a collector should support. It also describes how the collector must use major routines that the memory system provides.

2.1 Preliminary Information

Before we look at what a collector should provide, it is useful to explain where the definitions and interfaces can be found and what the types are that are used in these interfaces.

Each macro or function definition described in the Cookbook and Reference sections falls into one of four categories.

- `[must define]`. The collector must define the macro or function.
- `[may define]`. The collector may define the macro or function to enable optional or non-default behavior.
- `[must use]`. The collector must use the macro or call the function to implement basic collector behavior.
- `[may use]`. The collector may use the macro or call the function as needed.

All definitions are in the `[may use]` category unless otherwise noted.

2.1.1 Code Organization

Any headers files that are part of a collector should be placed in a subdirectory of `src/share/javavm/include/gc` named for the collector that uses them. This directory name should also be used for the source directory for the collector under `src/share/javavm/runtime/gc`. For example, for the ReallySimpleHeap collector described below, the collector's header files and source files would be placed in the following two directories:

```
src/share/javavm/include/gc/reallysimple
src/share/javavm/runtime/gc/reallysimple
```

The organization and build process for the EVM as a whole is beyond the scope of this document.

2.1.2 Definitions

The definitions used by a collector come from several header files. To simplify the development of collectors, all of these header files have been grouped together in a common `gc/specific_gc.h` header file. Collectors should include this header file along with any collector-specific header files. See “Header Files,” page 25, for more details.

Note that the names of all the routines that the collector must define start with “`specific`,” and the names of all the barrier macros end with “`_BARRIER`.”

2.1.3 Types

There are a large number of types defined by the whole JVM. Fortunately, there are only a handful that a collector needs to use.

2.1.3.1 Primitive Types

The following primitive (non-object) types are commonly used in the interfaces for the collector:

- **`bool_t`** - a simple boolean type with two values: `TRUE` or `FALSE` (1 or 0).
- **`java_int`** - the integral type for Java language integers (always 32-bit).
- **`uintptr_t`** - unsigned integers big enough to hold pointer values.

2.1.3.2 Object Types

The primary type used for representing objects is: **java_lang_Object**. It defines the basic structure of the start of any object in the heap:

```
struct {
    ObjHeader h;
    word32 fields[1]; /* actually zero or more fields */
}
```

This structure is described in “Object Layout,” page 5 in the Introduction section. There are three pointer types defined for each class that the JVM needs intimate knowledge of, but collectors typically pass references to objects either as `java_lang_Object*` or `Ijava_lang_Object`.

- **`java_lang_Object*`** is a direct pointer to an object, and should only be used by the memory system, the collector, or in inconsistent regions of the JVM.
- **`DEREFERENCED_Ijava_lang_Object`** is a volatile direct pointer to an object, and is usually only used to construct the following type:
- **`Ijava_lang_Object`** is a handle to an object (a pointer to a `DEREFERENCED_Ijava_lang_Object`). This type is used in the “mutator” portions of the JVM to protect the JVM from the GC moving objects (otherwise the C compiler might create a cached temporary pointer somewhere).
- **word32** - a union of one-word primitive types and `DEREFERENCED_Ijava_lang_Object`. In general, a collector uses this as an opaque type and should not examine the contents.

In addition, macros are provided for accessing the fields of an object and for querying information about its state. Collectors also need access to class information stored in a **NearClass**. Macros for accessing this information are listed in “Class Information,” page 28.

2.1.3.3 Function Types

In order to support the collection of information about objects in a heap, there are functions and macros that allow a collector to iterate over the objects in the heap applying a callback function to each object in turn. The type of these callback functions is: **`ObjIterator`**. Its definition is:

```
void (*ObjIterator)(java_lang_Object *obj, void *data)
```

where the `data` field is used to store the inputs, current state, and results of the process of applying the callback function to all the objects in question.

```
void (*RefIterator)(java_lang_Object **objp, void *data)
```

The `RefIterator` type is like `ObjIterator`, but it accepts pointers to a location that points to an object. This allows the callback to update pointers to objects. Another name for this type is **`RootCallback`**.

Another callback type can be used for any function that accepts a `void*` and returns `void`:

```
void (*SimpleFct)(void *data)
```

2.2 Core Collector Routines

There are 16 routines that a collector must implement, but only five that must interact with the memory system in complex ways. The collector must provide support for initialization, allocation, collection, expansion, and iteration. These functions are the core of the collector.

2.2.1 Initialization

Early in the initialization of the VM (before loading any classes), the memory system will call:

```
[must define]
    uintptr_t specificInitHeap(uintptr_t minWords, uintptr_t maxWords)
```

The collector is responsible for doing whatever initialization it needs, and allocating a heap large enough to hold at least `minWords`. The collector must not expand the heap larger than `maxWords` of object space. `minWords` will be maxWords. The sizes refer to the number of words available for objects—it doesn't limit the overhead that the collector might also need (for semi-spaces, remembered sets, mark bits, etc.). `specificInitHeap()` must return the number of words available for objects or zero if initialization failed. A collector will often use the `MemoryArea` utilities to allocate address space for the heap. See “MemoryArea,” page 37, for more information.

If a collector needs its own locks, it should initialize the locks here, and register the locks with the memory system. This allows the memory system to obtain all locks in the proper order before a garbage collection. See “Locking Support,” page 34, for more information.

2.2.2 Allocation

The memory system can allocate objects either by calling the collector for each allocation, or by using a per-thread cache if the collector supports it.

```
[must define]
    word32 * specificAllocateWords(java_int numWords)
```

The collector should return a pointer to new memory. It does not need to clear the memory. If there is not enough room, the collector should initiate a GC by calling `getLocksThenGC()` (declared in gcimpl.h). If there is still not enough space, it should return NULL. The memory sys-

tem is responsible for grabbing and releasing the heap lock, calculating the object size, initializing the object, dealing with heap expansion, and out of memory signaling.

2.2.2.1 "Fast" Allocator

If the collector supports linear allocation (allocation of various size objects in contiguous memory), it can define the macro FAST_ALLOCATOR to true. The memory system will then use a two-level allocation cache. The collector must provide a global cache of memory to the memory system bounded by the variables `fastAllocaterFreePtr` and `fastAllocaterLimit`. Essentially, the collector pre-allocates a block of space that the memory system may divide into objects on its own. A thread can create a local allocation buffer (LAB) by allocating out of the global cache (avoiding calls to the collector, but still requiring the heap lock). Each thread can then allocate objects out of its private LAB without synchronizing with other threads.

The collector needs to update `fastAllocaterFreePtr` and `fastAllocaterLimit` at the end of each call to `specificInitHeap()`, `specificGC()`, and `specificExpandHeap()`. `fastAllocaterFreePtr` should be set to point to the heap's "freeLimit" variable, so the memory system and the heap are kept in sync.

During normal execution, a thread's LAB is an unformatted array of words. When threads are suspended for GC or object iteration, the memory system formats the unused portion of the LAB to look like an array of `int`, and the thread clears its pointer to the LAB. The collector will then reclaim the unused LAB normally.

2.2.3 Collection

A collector cleans up the garbage in close cooperation with the rest of the memory system. The memory system does a lot of the work before calling the collector's GC routine, and provides essential services to the collector.

A collector can do a "full" collection (collect all objects in the heap), or a "partial" collection. Some collectors can only do full collections, but generational collectors (and others) may do partial collections. Classes cannot be unloaded unless the collector does a full collection.

> **Note**: Concurrent collectors may do most collection work as part of allocation and/or read/write barriers. The collection steps described below may only apply to the non-concurrent portion of collection (or not at all).

Collections may be initiated by Java code explicitly (using `java.lang.Runtime.gc()`), by the JVM for profiling or debugging, or more commonly by the collector when an allocation fails. All cases end up calling the memory system routine `getLocksThenGC()`.

2.2.3.1 Memory System's Responsibilities

`getLocksThenGC()` will:

1. Get the heap lock.
2. Start a GC thread to do the rest of collection:
3. Get the registered GC locks in order.

4. Suspend all threads.
5. Run the inconsistent threads until they are consistent and they suspend themselves.
6. Release the “early” GC locks.
7. Call the collector's `specificGC()`.
8. Unload unreferenced classes (if `specificGC()` did a full GC).
9. Call the collector's `specificExpandHeap()` if collection did not free enough memory.
10. Resume the threads.
11. Release the remaining (“late”) GC locks.
12. Hand discovered weak reference objects that have dying referents to Java code for processing.

2.2.3.2 Collector's Responsibilities

```
[must define]
    bool_t specificGC(java_int wordsRequested, bool_t fullCollection)
```

The collector should attempt to remove at least `wordsRequested` words of garbage from the heap. The memory system may request a full collection or a partial collection, but the collector can ignore this argument. The collector returns `TRUE` if it did a full collection and `FALSE` otherwise.

Most collectors work by walking the object graph from roots, keeping all reachable objects alive, and recycling the space used by unreachable objects. The memory system provides functions to iterate over all roots—Java language local and global variables, VM-registered handles, and JNI handles. A macro to iterate over all reference fields of a given object is also provided. The collector is free to use some low-order bits in the header of each object, and in each pointer for marking, etc. (see “Marking Macros,” page 29). In fact, it may modify the pointers in the object graph in any way it pleases as long as it restores the graph at the end of the collection and as long as it is able to recover the NearClass for a given object during the collection.

To iterate over the roots, the collector must call:

```
[must use]
void processStrongRoots(RootCallback globalRootCB, RootCallback localRootCB,
                        RootCallback stackRootCB,  RootCallback classRootCB,
                        bool_t allClasses, void *data)
```

The collector must provide callback functions for processing each VM global root, VM local root, Java stack root, JNI handle, and root in classes that are active on the stack or otherwise not unloadable. Often the same callback function can be used for all root types. The `data` parameter is passed unchanged to the callback functions. If a collector is only doing a partial collection, then it must treat all classes as roots. The collector should use `allClasses` `TRUE` in this case.

RootCallbacks

The signature of `RootCallback` is the same as `RefIterator`:

```
void (*RootCallback)(java_lang_Object **objp, void *data)
```

A root callback is responsible for processing the references in the object referred to by the object pointer `objp`, and updating the object pointer if the collector moves the object. The memory sys-

tem provides two support routines to help the callback. The first takes care of references in the object's class:

```
[must use]
static void scanNearClasses(java_lang_Object *obj, NearClass *ncls,
                            RootCallback classRootCB, void *data)
```

If the NearClass structure referenced by `ncls` is unmarked, this function will mark it and call the classRootCB callback on references found in the `Class` structure referred to by `ncls`, including:

- Various objects referred to directly by the class structure (class loader, name string, etc).
- All static object variables of the class.
- All of the class' constant pool references.

This function will also be called recursively on all of the unmarked classes referred to by the class, including:

- The superclass of the class.
- The class representing an "array of" the class if it exists.
- If the class is an array class, the class representing the array's declared element class.

To process the references in the object itself, the collector should use the following macro provided by the memory system:

```
[must use]
macro OBJ_REFS_DO(java_lang_Object *obj, NearClass *ncls, statements)
```

This macro creates an iteration variable named `refPtr` (of type `java_lang_Object**`) and executes *statements* once for each reference in the object, with `refPtr` set to the location of the reference. As a side-effect, `OBJ_REFS_DO()` may discover that `obj` is a weak reference object that needs to be treated specially (described in detail below).

2.2.3.3 Collecting Weak References

Weak reference objects were described in "Weak References," page 3. Weak reference objects have two fields and an implicit state that are of interest to the memory system:

- `referent`. The `referent` field is treated specially by the memory system only during the discovery phase and only if the state is `ACTIVE`.
- `next`. This field is used to link weak reference objects into memory system and then user-level queues. The `next` field is always `null` until the memory system begins processing the weak reference object.
- There are several implicit states, `ACTIVE`, `PENDING`, `ENQUEUED`, and `INACTIVE`, but the memory system is only concerned about reference objects in the `ACTIVE` state. This is encoded as `next` = `null`.

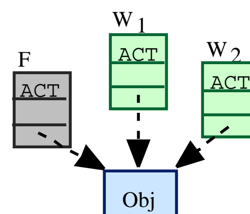
To implement weak references, the memory system and the collector collaborate in several phases:

1. **Discovery** phase. Discover weak reference objects and link them together into a `discoveredRefs` queue—a list of potential weak reference objects to process. This is accomplished by invoking the `startDiscoveringWeakRefs()` and then calling `OBJ_REFS_DO()` to walk over the objects in the heap. At the end of the traversal, `stopDiscoveringWeakRefs()` is invoked. The collector must call these in order to do the "scanning" phase of the collection algorithm.
2. **Processing** phase. Once the collector has determined all of the strongly reachable objects, the memory system and collector process the `discoveredRefs` queue and create a `tempPendingQ` of weak reference objects that have referents that are only weakly reachable. Processing actually happens once for each strength of weak reference object. Processing is done by `processWeakRoots()`, which the collector must call.
3. **Updating** phase (optional). If the discovery phase found weak reference objects in their old locations, then the `tempPendingQ` refers to a list of objects at the old locations. The memory system set the `next` fields but the collector has not yet seen these fields. The updating phase updates all of the links to refer to the objects' new locations using `updateTempPendingQ()`. The collector does not need to call `updateTempPendingQ()` if it is a non-moving collector, or if the collector calls `OBJ_REFS_DO` (discovers weak reference objects) on objects in their new locations.
4. **Queuing** phase. At the end of the garbage collection, the elements of the `tempPendingQ` are pushed onto the queue rooted in the Java static variable `java.lang.ref.Reference.pending`. This queue is drained by Java code that separates the weak reference objects into other various user-level queues (including the finalization queue). The reason that a temporary queue is used in the previous phases is that the processing and updating phases do not have to deal a potentially non-empty global queue. Queuing is done by `queuePendingRefs()`, which the collector must call.
5. **Notification** phase. If any weak reference objects were added in the queuing phase, the Java code (described above) that processes the global queue is notified. Notification is done by the memory system at the end of a collection (in `getLocksThenGC()`).

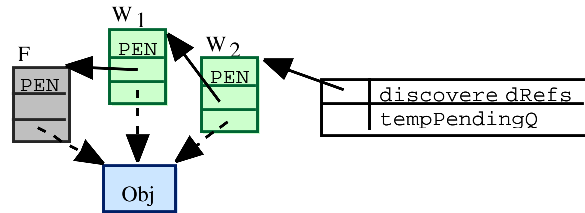
2.2.3.4 Weak References Example

This example shows the discovery, processing, and queuing phases in more detail. Consider a heap that consists of an object **`obj`** with a `finalize()` method, and two `WeakReference` objects (**`W1`** and **`W2`**) that weakly refer to **`obj`**. The JVM creates a `FinalReference` object **`F`** to handle the `finalize()` method, and links it into a global list.

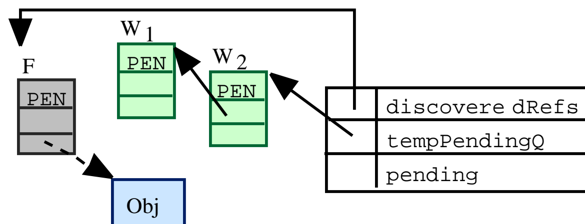
Initial state. Assume that **`W1`** and **`W2`** are rooted by Java static variables and there are no longer any strong references to **`obj`**. If we only look at each weak reference object's implicit state and its `next` and `referent` fields, the heap looks like this before a collection. All the weak reference objects are `ACTIVE` ("ACT").



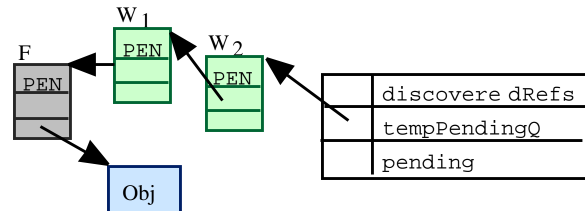
Discovery. While scanning from roots, the collector eventually calls `OBJ_REFS_DO()` on each of the three weak reference objects. `OBJ_REFS_DO()` notices that the weak references are `ACTIVE`, so they are enqueued and the collector does not yet see the `referent` field (or **`obj`**). Since the memory system sets the `next` fields of these weak reference objects, their states are now `PENDING` ("`PEN`").



Processing (part A). First the `WeakReference` objects are processed. The memory system and collector notice that **`obj`** is not strongly referenced, so **`W1`** and **`W2`** are enqueued on the `tempPendingQ` and the `referent` fields are cleared.



Processing (part B). Next, the `FinalReference` object is processed. Since **`obj`** is still not strongly reachable, **`F`** is enqueued and the memory system makes a collector callback (`weakRefCB`) on its `referent` field. The collector will mark or copy **`obj`** and update the `referent` field if necessary.



Queuing & Notification. At the end of the collection, the weak reference objects are removed from the `tempPendingQ` and placed in the `java.lang.ref.Reference.pending` queue. The memory system notifies Java code that the `pending` queue needs processing.

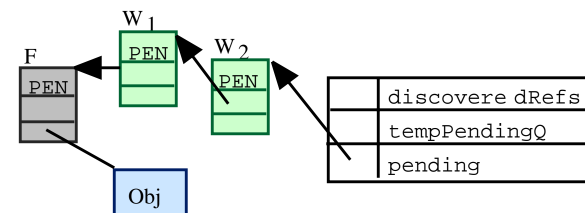


FIGURE 3. Weak Reference Processing

Subsequent states. After the collection completes and the mutator threads are resumed, Java code will eventually process the `pending` queue. The `FinalizerThread` will process the `FinalReference` and clear its `referent` field. A future collection will then notice that there are no references to **`obj`**, so it may be reclaimed.

As described earlier, the `OBJ_REFS_DO()` macro will treat weak reference objects specially during the "discovery phase" (controlled by `{start|stop}DiscoveringWeakRefs()`). In discovery mode, `obj` will be linked into a `discoveredRefs` queue if it is a weak reference object that is `ACTIVE`. The macro will not execute *statements* on the `referent` field of a discovered weak reference object. The collector should start the discovery phase while discovering all live objects starting from strong roots (the mark phase of a Mark & Sweep collector, for example), and stop the discovery phase before processing weak roots. If the collector makes additional iterations over the object graph, it can use the "non-discovering" version of the macro called `OBJ_REFS_DO_IGNORING_WEAKNESS()`, which treats all fields of weak reference objects as strong references.

After completing a pass through all objects reachable from strong roots, the collector should call `stopDiscoveringWeakRefs()`, then process the weak roots by calling:

```
[must use]
void processWeakRoots(WeakRootPredicate isDying,
                      RootCallback weakVMRootCB, RootCallback weakRefCB,
                      SimpleFct transitiveScannerCB, void *data)
```

This function will process all weak roots in the JVM, which includes the `discoveredRefs` queue of weak reference objects discovered by `OBJ_REFS_DO()` as well as certain JVM data structures like the string interning table. The collector needs to pass a predicate function as the `isDying` parameter, which should return `TRUE` if the object passed is dying (not marked, or not in "to space," etc.) The collector also needs to supply several callbacks as described below. The memory system processes the weak roots as follows:

1. For each "strength" of weak reference (from strongest to weakest: soft, weak, final, phantom) on the `discoveredRefs` queue discovered by `OBJ_REFS_DO()` do:

   1.1. Invoke `isDying()` on each `referent` field of each reference object of that strength.

   - If `isDying()` returns false, update the `referent` field of the weak reference object by calling the `weakRefCB()` on the field address. Then remove the reference object from the `discoveredRefs` queue (which return the reference to the `ACTIVE` state).
   - If `isDying()` returns true and if processing `SoftReference` objects (and memory is tight), or `WeakReference` objects, then clear the referent field and link the weak reference object into the `tempPendingQ`. The `tempPendingQ` is a global root which is always NULL at the beginning of GC, but subsequent calls to `processStrongRoots()` or `processRootsRound2()` will see it.

   1.2. If processing `FinalReference` or `PhantomReference` objects, then in a second pass, scan and update the `referent` field of each reference object by calling the `weakRefCB()` on the field address. Then link the weak reference object into the `tempPendingQ`. Note that the first pass removed all of the weak reference objects with non-dying referents.

   1.3. If the `weakRefCB()` does not scan the transitive closure of the objects reachable from the `referent` fields, then the caller must provide a `transitiveScannerCB` to do this. Otherwise, the collector can pass NULL as the `transitiveScannerCB` parameter.

2. For each internal weak data structure in the VM (note that these are weaker that Java weak references):

2.1. Invoke `isDying()` on each weak reference.

2.2. In a second pass, invoke `weakVMRootCB()` on each weak reference that the memory system wants to keep alive.

The function `processWeakRoots()` creates a `tempPendingQ` of weak reference objects by linking the objects together using their `next` field. But the collector has never seen these fields. If the collector called `OBJ_REFS_DO()` on objects before they were moved, or if all pointers in the heap need to be "reversed" or otherwise processed, then the collector must call `updateTempPendingQ()` passing a `weakRefUpdaterCB()` callback.

2.2.3.5 Before and After Collection

There is some bookkeeping that the collector must do before and after a collection. Before starting an iteration over reachable objects (for example, a marking phase), the collector must call `clearClassReferencedBits()` to mark each class structure as "unreferenced," so that `scanNearClasses()` can mark the referenced classes correctly. The collector must also call `startDiscoveringWeakRefs()` to tell the memory system (`OBJ_REFS_DO()` in particular) to begin discovery phase mode.

After the collection has been completed (and the object graph has been restored), the collector needs to call `queuePendingRefs()` to merge the weak reference objects on the temporary pending list onto the Java static variable `pendingQ`. This global queue may not have been empty at the beginning of GC. If this GC added items to the global queue, the memory system will notify Java code to take care of the queue.

If the collector has moved any objects, it needs to increment `gcMoveCount`.

2.2.3.6 Example Collector: ReallySimpleHeap

ReallySimpleHeap is a mark/sweep collector—a simplified version of the "simpleheap" collector in the EVM. Here is the code for collection (but not allocation or free lists). Note that this is a non-moving collector:

```
void scanObjectCB(java_lang_Object **root, void* data) {
    java_lang_Object *obj = *root;
    NearClass *ncls = getNearClass(obj);

    if (hasLiveMark(ncls)) return;                    /* Already done.  */
    setNearClass(obj, setLiveMark(ncls));             /* Mark it.       */
    scanNearClasses(obj, ncls, scanObjectCB, data);
    OBJ_REFS_DO(obj, ncls, if (*refPtr) scanObjectCB(refPtr, data));
    /* note that "refPtr" is defined by OBJ_REFS_DO macro */
}

bool_t isDyingCB(java_lang_Object *obj, void *data) {
    return !hasLiveMarkObj(obj);
}
```

```
void mark() {
    clearClassReferencedBits();
    startDiscoveringWeakRefs();
    processStrongRoots(scanObjectCB, scanObjectCB, scanObjectCB, scanObjectCB,
                       FALSE,     /* Don't scan all classes        */
                       NULL);     /* No data needed                */
    stopDiscoveringWeakRefs();
    processWeakRoots(isDyingCB, scanObjectCB, scanObjectCB,
                     NULL,      /* scanObjectCB did transitive closure */
                     NULL);     /* No data needed                      */
}

void freeDeadObjectCB(java_lang_Object *obj, void *data) {
    if (isDyingCB(obj, NULL))
        myFreeObject(obj, ...); /* link onto free list */
}

void sweep() {
    /* walk over heap, putting unmarked objects onto a free list. */
    specificObjectsDo(freeDeadObjectCB, NULL);
}

bool_t specificGC(java_int wordsRequested, bool_t fullCollection) {
    mark();
    sweep();
    queuePendingRefs(); /* Tell Java code about weak reference objects */
    /* NOTE: This collector does not support FAST_ALLOCATION. */
    return TRUE; /* It was a full GC, so sweep classes. */
}
```

The use of recursion in `scanObjectCB` above would result in poor performance and stack use, but makes the example simpler.

2.2.4 Expansion

A collector typically expands the heap when asked to by the memory system (see the description of `getLocksThenGC()` above). Some collectors also call their `specificExpandHeap()` as a way to allocate the initial heap in `specificInitHeap()`. Either way, the collector can count on having the heap lock and that all threads are suspended.

```
[must define]
    uintptr_t specificExpandHeap(uintptr_t minWords,
                                 uintptr_t suggestedWords)
```

The collector should attempt to expand the heap by at least `minWords` up to `suggestedWords`. The memory system currently uses an aggressive heap expansion strategy, and suggests doubling the heap size at each expansion. The result should be 0 if the expansion failed, otherwise the actual number of words added to the heap. The sizes refer to the number of words available for objects.

2.2.5 Iteration

The memory system provides a function `allObjectsDo()` that iterates over all objects in the heap. The memory system will suspend all threads, update the threads' LABs, then call the collector's heap iterator:

```
[must define]
    void specificObjectsDo(ObjIterator fct, void *data)
```

The collector must call `fct` exactly once for each object in the heap, passing it a reference to the object and `data`. The collector is allowed to call `fct` with objects that would be "dead" if a GC was done before the call to specificObjectsDo.

2.3 Read/Write Barriers

A collector may also implement barrier macros, which give the collector hooks into the JVM to monitor various activities. Barriers don't actually do a read, write, or allocate; they just do any necessary bookkeeping. There are barriers for Java heap and Java static variable access, but not for Java stack access (because of type-indeterminate bytecodes like "dup").

A collector must define all of the following macros, but it may leave the macro bodies empty for all unnecessary operations. There are several general rules for barriers:

- Read barrier macros must be an expression returning a value, though this value will be ignored. This allows callers to use read barriers in comma-expressions.
- Write barrier macros take the object being modified (heap barrier only), the address of the field/variable to be modified, and the new value. Write barriers are called before the new value is written so the barrier can see the old value and the new value.
- All barrier macros can only be called when the caller is inconsistent.

```
[must define]
    bool_t READ_HEAP_BARRIER(java_lang_Object *obj,
                             java_lang_Object **refAddr)
    void  WRITE_HEAP_BARRIER(java_lang_Object *obj,
                             java_lang_Object **refAddr,
                             java_lang_Object *rr)
    bool_t READ_STATIC_BARRIER(java_lang_Object **refAddr)
    void  WRITE_STATIC_BARRIER(java_lang_Object **refAddr,
                               java_lang_Object *rr)
```

The read/write barriers are enforced by the convention in the VM of only accessing objects through the `memsys` interface or the LLNI interface (which uses `memsys`). The `memsys` interface provides a variety of object class and hash code accessors, object field accessors, array element accessors, and static field accessors. The only code which is allowed to directly manipulate object state is the memsys interface and the collector itself.

If a collector does not need any read or write barriers, or it can do its bookkeeping in batches, it should define the macros `canBatchReadBarrier` and/or `canBatchWriteBarrier` to `TRUE`. If a collector can do its bookkeeping in batches, then it should also define one or both of these macros:

```
[must define]
    void  BATCH_HEAP_READ_BARRIER(java_lang_Object **startAddr,
                                  java_int numWords)
    void BATCH_HEAP_WRITE_BARRIER(java_lang_Object **startAddr,
                                  java_int numWords)
```

If supported, these macros will be called before copying object arrays (in `JVM_ArrayCopy()`), and before cloning objects with reference fields (in `inconsistentClone()`). `startAddr` points to the first word to be read or written. Unlike the other write barriers, `BATCH_HEAP_WRITE_BARRIER` does not get to see the new values.

> Note that there is no fast way to get exact information about where pointers are located when cloning objects, so batch barriers are best suited for collectors that use inexact remembered sets (which remember the object accessed rather than the word accessed).

There are other barriers that give the collector hooks into the JVM.

```
[must define]
    void      NEW_INSTANCE_BARRIER(ExecEnv* ee, java_lang_Object *refAddr)
    void FAST_NEW_INSTANCE_BARRIER(ExecEnv* ee, java_lang_Object *refAddr)
```

The new instance hooks are called after every object has been allocated and initialized. The object's class information will be set. The barrier is called within the same inconsistent region as the allocation. This barrier allows a collector to do any per-object bookkeeping. The `FAST_NEW_INSTANCE_BARRIER` is only called during an allocation out of a thread's LAB, so the collector may make certain assumptions (such as the LAB is always in the youngest generation).

Finally, the handle barrier detects writes to global roots by being called every time an LLNI handle is updated. It is called before the write takes place.

```
[must define]
    void WRITE_HANDLE_BARRIER(ExecEnv* ee, Ijava_lang_Object refAddr,
                              java_lang_Object *rr)
```

2.3.1 Example Barriers

ReallySimpleHeap doesn't require any barriers, so the following example comes from the EVM's generational heap. The generational heap uses card tables for remembered sets, so it defines the following barriers:

```
#define READ_HEAP_BARRIER(obj, refAddr) TRUE
#define WRITE_HEAP_BARRIER(obj, refAddr, rr) \
    setCTEModPtr((word32*)(refAddr))

#define canBatchReadBarrier  TRUE
#define canBatchWriteBarrier TRUE

#define BATCH_HEAP_READ_BARRIER(startAddr, len)
#define BATCH_HEAP_WRITE_BARRIER(startAddr, len) \
    setCTEModRange((word32*)(startAddr), (len))
```

Note that `setCTEModPtr()` is a static inline function in the generational collector. This combines the better type-checking and debugging of function calls with the efficiency of macros.

2.4 Supporting Collector Routines

In addition to the core routines, there are also a number of support routines that must be defined. These provide support for information about the state of the collector, verification, descriptions, and a few object-specific functions.

2.4.1 State of the Collector

There are several functions that need to be defined in order for the VM to know how much memory is available for allocation and how much is in use:

```
[must define]
    uintptr_t specificTotalObjectWords()
    uintptr_t specificTotalHeapWords()
```

These two functions report how much memory has been used. The first reports the total amount of memory in words that is available for allocating objects. The second reports the amount of memory in words used by the collector including memory for supporting data structures including "to" space, card tables or remembered sets, and similar structures. `specificTotalHeapWords()` is used by debugging and heap analysis tools, and does not need to track every single byte.

```
[must define]
    uintptr_t specificMaxAllocation()
```

This function reports the size of the largest available chunk of free memory in the heap in words. The memory system calls this after a collection to determine if the allocation request that triggered the collection will succeed. This function is allowed to return a conservative estimate at the cost of expanding the heap earlier than needed on occasion.

```
[must define]
    uintptr_t specificFreeHeapWords()
```

This function returns the approximate number of words free in the heap at a point in time. The result is approximate because the collector must calculate this quantity without locking the heap.

2.4.2 Collector Verification

An important part of debugging the behavior of the VM is the ability to verify that the collector and the heap it manages are in good shape. To support this process, three functions must be defined. While these functions do not return anything, they may abort if there are any discrepancies.

```
[must define]
    void specificVerifyHeap()
```

This function walks over the heap and makes sure that any collector-specific invariants hold for the heap. It can be very slow since it is only used for debugging purposes, and it may abort the VM.

```
[must define]
    void specificVerifyReference(java_lang_Object *obj)
```

This function does any collector-specific checks to make sure that a reference is valid. This can range from verifying that the pointer points into the heap that the collector is managing to verifying bits in the objects header and class, etc. It assumes that `obj` is not null and is properly aligned.

```
[must define]
    void specificVerifyIsFree(word32 *from, word32 *to)
```

This function verifies that words in the range `[from, to)` are free and is typically called by the collector itself in debug mode before allocating words.

2.4.3 Descriptive Functions

There are two functions that provide some level of description of a given collector. These functions are:

```
[must define]
    char* specificDescribeMemSys()
    char* specificName()
```

The first function, `specificDescribeMemSys()`, returns a description of the specific memory system. This description may consist of one or more lines, each terminated with a newline character. The second, `specificName()`, returns a simple name for the collector. The name should not contain a newline character.

2.4.4 Object-specific Functions

Some collectors make use of the headers of objects during a collection. For example, some collectors move an object's class field to the side in order to link related objects together. Because other parts of the VM may need to access the class information of an object while a collection is in progress, two functions must be provided:

```
[must define]
    NearClass* specificGetNearClass(java_lang_Object *obj)
    bool_t     specificIsObjectInGCNow(java_lang_Object *obj)
```

The first function will return the actual class of an object even if that class information has been temporarily replaced or obliterated in the object's header. So, unlike IBM punch cards, objects can be marked, mangled, folded, mutilated, or spindled as long as we can use this function to recover the information we need. The second function will return `TRUE` if and only if the object is in an area that is currently being collected. In both cases, the caller is expected to check any general properties, and, in particular, handle the case where the argument is NULL.

3. Reference Section

This section describes many of the macros and routines supplied by the memory system and the JVM that the collector can use (or redefine). All definitions in the Reference section have `[may use]` status unless described otherwise.

3.1 Header Files

The header files used by collectors are subject to change, but this is where definitions currently live:

TABLE 1. Header files used by collectors

Defined by	File	Contents
collector	`gc/abstractcollector.h`	prototypes of functions that collectors provide
collector	`gc/barriers.h`	collector can define barriers here
memory system	`gc.h`	public gc interface
memory system	`gc/gcimpl.h`	most support routines
memory system	`gc/references.h`	support for processing references
memory system	`gc/memarea.h`	mapped memory utilities
collector & memory system	`memsysconfig.h`	collector may set some of the definitions in this file, and may use the others
JVM	`classunloading.h`	class scanning
JVM	`common_exceptions.h`	exception-throwing functions
JVM	`llni.h`	LLNI interface `NearClass`
JVM	`memsys.h`	accessor functions for object headers and fields
JVM	`monitor.h`	locks
JVM	`oobj.h`	`NearClass` and `Class`
JVM	`sys_api.h`	sysAssert, sysMalloc, etc.
JVM	`threads.h`	threads-related functions
JVM	`typedefs.h`	object layout
JVM	`util.h`	assertions, i/o, and copy utilities

All of these files are included by `gc/specific_gc.h`, which should be used by all collectors.

3.2 Collector Configuration

The collector needs to set a couple of macro definitions, and may change some default definitions. The definitions below are usually controlled by a single environment variable `GCCONFIG`, which is read by the makefiles `JVM.gmk` and `garbage.gmk`.

```
[may define]
macro int EXACT_STACKS  /* default is true (1) */
```

The memory system usually relies on a stack map computer to be able to decode both interpreted and compiled Java stacks. When developing new JIT compilers it may be useful to scan Java

stacks conservatively. The collector can define `EXACT_STACKS` to be false (0). In that case, `processStrongRoots()` will call the collector's `stackRootCB()` callback on all values in the Java stack, and the collector must deduce probable object references. Such a collector must be of the non-moving variety.

Note that the memory system always knows the exact location of references in the JVM C frames and in native method frames, due to the LLNI and JNI interfaces.

```
[must define]
macro int HEAP_CHOICE
```

The JVM is compiled with `HEAP_CHOICE` set to a unique integer constant representing the collector to use. The collector needs to choose a constant which will represent it, and ensure that `HEAP_CHOICE` gets set to that value. See `memsysconfig.h` for existing heap choice values.

```
[must define]
macro int BARRIER_CHOICE
```

The collector needs to compile the JVM with `BARRIER_CHOICE` set to a unique integer constant representing the barrier required by the collector, or `NO_BARRIER` if the collector doesn't require a barrier. See `memsysconfig.h` for existing barrier choice values.

```
[may define]
macro long DEFAULTHEAPSIZE
```

The collector may change the default value for the initial heap size. Overridden by `args->minHeapSize` (`-ms` on the command line).

```
[may define]
macro long MINHEAPSIZE
```

The collector may change the lower bound on `args->minHeapSize`.

```
[may define]
macro long MAXHEAPSIZE
```

The collector may change the default value for the maximum heap size. Overridden by `args->maxHeapSize` (`-mx` on the command line).

3.3 Sizes and Casts

```
macro java_lang_Object*     asObject(any obj)
macro java_lang_Object** asObjectPtr(any objp)
macro NearClass*         asNearClass(any ncls)
```

These macros are meant to serve as casts to the appropriate type. They are used by the masking macros (below).

```
macro int JAVA_INT_BITSIZE
```

This macro simply defines the number of bits in a value of type `java_int`, which is always 32.

```
java_int objectSize(java_lang_Object *obj)
```

Return size in words of an object (including arrays).

```
java_int objectSizeGivenNearClass(java_lang_Object *obj, NearClass *ncls)
```

Like `objectSize()` but also takes near class to be more efficient and will work even if near class in object header is marked.

3.4 Object Header Information

Macros and functions that access the information stored in an object's header.

```
NearClass *getNearClass(java_lang_Object *obj)
NearClass *setNearClass(java_lang_Object *obj, NearClass *ncls)
```

These functions return or set the `NearClass` field stored in the object's header. Neither of these functions strip off any of the low-order bits in the header that the collector may have set. If the collector uses any bits from the near class pointer, the collector can use the `clearAllMarks` macro defined below to easily strip them off.

```
NearClass *getNearClassFromObject(java_lang_Object *obj)
```

Some collectors mangle or otherwise obfuscate the `NearClass` field in an object while a collection is ongoing. This is a heavier-duty function that can be used to recover the `NearClass` field in these cases. This function calls the collector's `specificGetNearClass()` function.

```
typedef uint32_t BitField;
typedef union multiUseWord_s {
    BitField  allBits;
    word32   *forwardingPtr;      /* Used by some GC algorithms. */
} multiUseWord;
```

The second word in an object's header is a `multiUseWord` that contains the hash code, age (gc), and locking bits encoded in a `BitField`. The locking protocol may swap out the hash code and age bits if the object is in the middle of a locking operation (lock, unlock, wait, etc).

```
[may define]
macro int HASH_BITS /* 25 */
macro int AGE_BITS  /*  5 */
```

A collector may change the number of bits in the `BitField` allocated to hash code and age information.

```
int getLockState(BitField allBits)
int getAge(BitField allBits)
int getHash(BitField allBits)
```

These are low-level macros that decode a `BitField`.

```
int  getHashCode(ExecEnv *ee, Ijava_lang_Object obj)
void setHashCode(ExecEnv *ee, Ijava_lang_Object obj, int ii)
```

Gets or sets an object's hash code. These functions go through the locking code's synchronization process to get access to the real bits. Unlike the other functions listed here, these functions must be called while the caller is consistent (because it may block). This precludes calling them from most collector functions.

```
BitField  getHeaderWord    (java_lang_Object *obj)
void      setHeaderWord    (java_lang_Object *obj, BitField ii)
```

These low-level routines get or set the raw header word that contains the hash code, age bits, and lock bits. Note that if the object is locked, the hash code and age bits may be swapped out. These routines should be called only in single-threaded mode (such as during garbage collection).

```
BitField *otherBitsPtrNoSync(java_lang_Object *obj)
```

This function gives non-synchronized access to the bits in the `multiUseWord`. This function can only be called in single-threaded mode (such as during garbage collection). It will attempt to return a pointer to the location that currently contains the `multiUseWord` of the object. There are three cases depending on the lock state of the object:

- `lsNeutral`: the `multiUseWord` is in the object
- `lsWaiters`, `lsLocked`: the `multiUseWord` is in the first lock record
- `lsBusy`: the `multiUseWord` is inaccessible (return NULL).

```
int incAgeNoSync(java_lang_Object *obj)
```

Return the current age of the object and update the age in the object to be one greater (if it is not already pegged at MAX_AGE). The age returned will be in the interval [0, MAX_AGE]. If lock state is `lsBusy`, the real age can't be found, so returns 1.

```
void setAgeNoSync(java_lang_Object *obj, int i)
```

Set the age of the object if the lock state is not `lsBusy`.

3.5 Class Information

Class information is represented in several forms, but the primary way in which collectors access class information about an object is through the function `getNearClass`, described in “Object Header Information,” page 27.

Given a `NearClass` pointer, there are several macros that may be used to get specific class information:

TABLE 2. NearClass accessor macros

NearClass macro	Type	Description
`ncFarClass(ncls)`	`Class *`	the Class associated with a NearClass
`ncFlags(ncls)`	`uint32_t`	Flags used for purposes like class unloading and GC
`ncSizeInfo(ncls)`	`int32_t`	if this field is: > 0, it is the size of an object of this class in words <= 0, the class is an array type and the value is -log2(number of bytes per element)
`ncMap0(ncls)`	`uint32_t *`	A map of the location of references in an object of this NearClass
`ncMap2(ncls)`	`uint32_t *`	A map of the location of weak references in an object of this NearClass
`ncOK(ncls)`	`void *`	When `DEBUG` is defined, this can be used to test if it is equal to a fixed pattern (`nearClassOKBits`)
`ncIntfMethodTable(ncls)`	`imethodtable *`	The method table for interfaces
`ncInterpMethodTable(ncls)`	`Method **`	The table of method blocks for the interpreter
`ncComMethodTable(ncls)`	`void *`	The table of compiled method blocks

The `ncFarClass` macro may be used to access additional class-related information. Here are the more commonly used macros on `Class` pointers:

TABLE 3. Class accessor macros

Class macro	Type	Description
`cbNearClass(cls)`	`NearClass *`	the NearClass associated with a Class
`cbName(cls)`	`char *`	the name of the class (e.g., "java/lang/Object")
`cbHasRefs(cls)`	`bool_t`	TRUE iff an instantiation contains references
`cbSuperClass(cls)`	`Class *`	the superclass associated with the Class

3.6 Marking Macros

As described in “Object Layout,” page 5, a collector can use several (at least two) bits in an object's header. As a convenience, the memory system predefines macros that use the two low-order bits as a “live” mark and an “indirection” mark. A collector is free to use these bits (during collection) for any purpose, however.

```
macro int OBJECT_ALIGNMENT_MASK
macro int HEADER_ALIGNMENT_MASK
```

The first mask governs how objects should be aligned while the second governs how `NearClass`es (which are pointed to by object headers) should be aligned. Objects and `NearClass`es are minimally aligned on word boundaries, so the masks are at least `0x3` and should have a value (2^n - 1) for some n.

```
macro NearClass* clearAllMarks(NearClass* ncls)
macro bool_t        hasAnyMark(NearClass* ncls)
```

Catch-all macros that mask or query all of the non-reference bits from a class reference so that it may safely be used to access the referred-to class structure.

```
macro int LIVE_MARK
macro int INDIRECTION_MARK
```

These macros define bits that are used to indicate that an object is live and/or has been moved, or that a pointer is a member of an indirection list (during a collection).

```
macro NearClass*        clearLiveMark(NearClass* ncls)
macro bool_t              hasLiveMark(NearClass* ncls)
macro NearClass*          setLiveMark(NearClass* ncls)

macro NearClass* clearIndirectionMark(java_lang_Object* ptr)
macro bool_t       hasIndirectionMark(java_lang_Object* ptr)
macro NearClass*   setIndirectionMark(java_lang_Object* ptr)
```

Given a pointer value, these three macros mask, query, and “add” the `LIVE_MARK` or `INDIRECTION_MARK` on this value. Note that these don't read or write the heap at all—they simply return new values based on pointer values.

```
macro bool_t hasLiveMarkObj(java_lang_Object* obj)
```

This macro tests if an object has been marked live by looking at its header.

3.7 Root-processing and Iteration Support

```
[must use]
void processStrongRoots(RootCallback globalRootCB, RootCallback localRootCB,
                        RootCallback stackRootCB,  RootCallback classRootCB,
                        bool_t allClasses, void *data)
```

This function processes strong roots in the following fashion. It

- invokes globalRootCB on locations of all strong global roots,
- invokes localRootCB on locations of all local roots in any thread,
- invokes stackRootCB on all reference-holding locations in thread stacks. If `EXACT_STACKS` is defined to be true, we get exactly the reference-holding locations; otherwise we get a conservative approximation.
- If `allClasses` is `TRUE`, invokes classRootCB on locations of all object references in all classes in the system. Otherwise, it invokes classRootCB on all object references in

classes that are reachable from Java stack frames (the classes of the methods of each frame), and all other non-unloadable classes.

See "Collection," page 12, for more information.

```
[must use]
typedef bool_t (*WeakRootPredicate)(java_lang_Object *weakRootObj,
                                    void *data)
void processWeakRoots(WeakRootPredicate isDying,
                      RootCallback weakVMRootCB, RootCallback weakRefCB,
                      SimpleFct transitiveScannerCB, void *data)
```

This function processes weak roots. See "Collecting Weak References," page 14, for more information.

```
void processRootsRound2(RootCallback globalRootCB, RootCallback localRootCB,
                        RootCallback stackRootCB,  RootCallback classRootCB,
                        void *data)
```

This function iterates over all weak and strong roots and all references in classes that have been marked. This function is used for garbage collection algorithms that require multiple passes over roots (such as the "updating roots" phase of the generational mark-compact collector). The initial pass over the roots should be done with `processStrongRoots()` and `processWeakRoots()`. Subsequent passes should be done with this function.

```
[must use]
void startDiscoveringWeakRefs()
```

Call this before starting "mark phase" (i.e., calling `processStrongRoots()`, etc).

```
[must use]
void stopDiscoveringWeakRefs()
```

Call this after finished "marking" strong roots, but before calling `processWeakRoots()`.

```
void updateTempPendingQ(RootCallback weakRefUpdaterCB, void *data)
```

Call this after calling `processWeakRefs()` IFF weak reference objects were discovered by `OBJ_REFS_DO()` in their OLD locations. The `tempPendingQ` was created based on those addresses so the `next` fields need to be updated to point to the new locations (or otherwise processed by the gc so it can update them later). The `weakRefUpdaterCB` can often by the same function as the one passed as `weakRefCB` to `processWeakRoots()`. If weak reference objects are discovered by `OBJ_REFS_DO()` in their NEW locations, do NOT call this function.

```
[must use]
void queuePendingRefs()
```

Copy newly found pending weak reference objects to the real pendingQ. Call in your `specificGC()` function after all object references have been fixed up.

[must use]
```
macro OBJ_REFS_DO(java_lang_Object *obj, NearClass *ncls, statements)
```

This macro creates an iteration variable named `refPtr` (of type `java_lang_Object**`) and executes *statements* once for each reference in the object, with `refPtr` bound to the location of the reference in the object. The `ncls` parameter provides the requisite class information to identify the references in the object. See "Collecting Weak References," page 14, for a description of how `OBJ_REFS_DO()` may also "discover" weak references.

```
macro OBJ_REFS_DO_IGNORING_WEAKNESS(java_lang_Object *obj,
                                    NearClass *ncls,
                                    statements)
```

Like `OBJ_REFS_DO()`, but it never does weak reference discovery.

[must use]
```
void scanNearClasses(java_lang_Object *obj, NearClass *ncls,
                     RootCallback classRootCB, void *data)
```

Scan the class structure of an object and, it if is an instance of java.lang.Class, also scan the class structure that it represents.

```
void scanRootClasses(RootCallback classRootCB, void *data, bool_t scanAll)
```

Scan the root classes, invoking `classRootCB` on every object reference in these classes (the object references are found in static variables, constant pools, and other internal fields of classes), marking the classes as we encounter them. It is expected (but not required) that `classRootCB` calls back into the class scanning code using the `scanNearClasses` macro to cause other non-root near classes to have their "Referenced" bit set and be scanned. Any class that we can't unload is considered a root and must be scanned. Normally, these are only the sticky classes, but at times all classes are treated this way (e.g., when the verifier or profiler is active). If `scanAll` is true, scan every class, regardless of whether it is a root or not in the above sense. This is used in a generational memory system where, if we are not doing a full GC, we must scan all classes (if we did not scan a specific class, the objects it points to could become garbage while the class survives). Note: this function only scans classes which have a clear "Referenced" bit (if the "Referenced" bit is set, it is taken as an indication that the class has already been processed). If you need to scan all classes, call `clearClassReferencedBits()` first.

[must use]
```
void clearClassReferencedBits()
```

Clear the "Referenced" status in all classes (the bit is actually in the `NearClass`).

```
void scanObjectRefsInMarkedClasses(RootCallback classRootCB, void *data)
```

Process all object references in classes that have been marked. This function is useful for GC algorithms that employ a separate pointer updating phase after marking live objects.

```
void objToRefIterator(java_lang_Object *obj, RefIterator fct, void *data)
```

This function invokes `fct` on all non-null reference fields of `obj`. It is a "functional" version of the `OBJ_REFS_DO()` macro. The callback `fct`, is passed two arguments: the location of the reference and a pointer to the data.

```
void referencesDo(RefIterator fct, void *data, java_int stackSize)
```

This function invokes `fct` on all non-null references in the system. It does this by using `fct` to process all the roots and then walking over all the objects in the heap, applying `fct` to the references in each object in turn. If called while threads are running, this function stops the world consistently (as if during a GC) and iterates over the heap in a new thread. `stackSize` can be used to specify the number of bytes to allocate for the thread's stack, or zero may be passed to get a default size.

```
void scanObjectsDo(ObjIterator liveFct, void *liveData,
                   ObjIterator deadFct, void *deadData,
                   int depth, java_lang_Object **objList,
                   int objListLength, java_int stackSize)
```

This function applies a pair of functions to all objects in the heap based on whether the objects are alive or dead. If called while threads are running, this function stops the world consistently (as if during a GC), cleans up LABs (if in use), and iterates over the heap in a new thread. Note that if the world is already stopped, it is assumed that LAB's are already cleaned up. `stackSize` can be used to specify the number of bytes to allocate for the thread's stack, or zero may be passed to get a default size. The function works in two phases:

1. Mark all the objects using livemarks on the objects' NearClass fields.
2. Iterate over the objects, applying the appropriate callback function depending on whether the objects are alive or dead, and clearing the livemarks as objects are processed.

The `depth`, `objList`, and `objListLength` parameters are used to bound the amount of stack used during the marking phase. The depth parameter bounds the number of recursive calls that will be made at any given time. If the depth limit is reached on a particular object, that object is pushed onto the object list, `objList`, so that it and similar objects can be processed later. If the object list overflows, the marking function switches to purely recursive marking until there is again room on the object list.

3.8 Collection-related Support

```
[must use]
void getLocksThenGC(java_int wordsReq, bool_t fullCollection,
                    java_int stackSize, bool_t hasHeapLck)
```

This function acquires all registered locks, and the heap lock if `hasHeapLck` is `FALSE`, stops the other threads in a consistent state, then invokes `specificGC()` to perform a collection, seeking `wordsReq` free memory. If necessary, it also expands the heap by invoking `specificExpandHeap()`. The specific GC routines should not do this. If `fullCollection` is `TRUE`, it does a complete collection of the heap. `stackSize` can be used to specify the number of bytes to allocate for the gc thread's stack, or zero may be passed to get a default size.

```
extern uintptr_t  maxHeapWords
```

This variable keeps track of the maximum number of words the heap is allowed to consume. It is set in `initHeap()`.

```
void setRootCategory(char *catName, ExecEnv *someEE)
```

For collecting statistics. Call this function to register which category of roots is currently being reported by the `processStrongRoots()` and `processWeakRoots()` functions. The category remains in effect until the next call. Usually used only by generational collectors (to categorize younger and older incoming pointers) and the memory system itself.

3.9 Fast-allocation Support

Support for thread local allocation buffers(LABs). See “Allocation,” page 11, for more information.

```
[may define]
macro int FAST_ALLOCATOR
```

A collector can define this true (1) if your memory system configuration uses a contiguous allocator in the youngest generation and is able to hand over the allocation to the fast contiguous allocator in gcimpl.c. Otherwise define as false (0).

```
[may define]
macro int MAX_LOCAL_BUFFER_SIZE  /* 1000 for example */
macro int MIN_LOCAL_BUFFER_SIZE  /* 10 for example */
macro int BIG_FOR_LOCAL_BUFFER   /* Must be >= MIN_LOCAL_BUFFER_SIZE. */
```

Collector can set these limits on how large and small each LAB can be, and the size of an object that is considered too big to allocate in a LAB. The limits are word-based.

```
macro ifFastAllocator(statements)
```

Conditionally compile *statements* if `FAST_ALLOCATOR` is defined to be true.

```
extern word32  *fastAllocaterLimit
extern word32 **fastAllocaterFreePtr
```

If the allocator in `specificAllocateWords()` uses allocation from a contiguous space, and if `FAST_ALLOCATER` is `TRUE`, then these two variables are used to speed up allocation by caching the area from which a thread allocates. The function `specificAllocateWords()` will then be called only upon running out of space according to these cached pointers. Note that if `fastAllocater-Limit == NULL`, `specificAllocateWords()` will be called for every allocation.

3.10 Locking Support

```
void registerGCMutex(sys_mutex_t *mut, bool_t early, int rank)
void registerGCRecursiveMutex(RecursiveMutex *rmut, bool_t early, int rank)
```

These functions register locks that GC must claim before starting. Locks are given an integer `rank`; the GC will acquire locks in order of ascending rank (where `RANK_LEAF` is maximal). Since all locks in the JVM must have a total order, it is beyond the scope of this document to describe how to choose a rank other than `RANK_LEAF`. Note that while some code holds a `RANK_LEAF`, it may not acquire any other locks (`getLocksThenGC()` is the only exception to this). No two GC locks in the system can have the same rank (except for leaf locks). Locks that are registered with `early`

`TRUE` are released once all threads are stopped; locks registered with `early` `FALSE` are held for the duration of collection.

"Early release" GC locks are acquired before suspending threads and are released after the threads are suspended. "Late release" GC locks are acquired at the same time, but are held for the duration of the collection. A collector needs to use an early release lock if it may try to reacquire a non-recursive lock while collection is in progress, or if the collector wants to grab the lock from all threads, but is willing to let non-JVM threads have access to the lock while collection is in progress (the "malloc lock" is a possible example).

```
void initAndRegisterMutex(sys_mutex_t *mut, const char *name, int rank);
void initAndRegisterRecursiveMutex(RecursiveMutex *rmut,
                                   const char *name,int rank);
```

These functions initialize and register locks generically. This must be done before registering a lock as a GC mutex.

```
bool_t sysMutexLocked(sys_mutex_t *rm)
void   sysMutexLock  (sys_mutex_t *rm)
void   sysMutexUnlock(sys_mutex_t *rm)
```

Do the obvious things with non-recursive mutexes. Note that a `sys_mutex_t` is simply a Solaris mutex.

```
bool_t recursiveMutexLocked(RecursiveMutex *rm)
void   recursiveMutexLock  (RecursiveMutex *rm)
void   recursiveMutexUnlock(RecursiveMutex *rm)
```

Do the obvious things with recursive mutexes.

3.11 Miscellaneous Support

```
bool_t heapInitialized()
```

This function returns `TRUE` if the heap has been initialized. *[must use]*

```
extern volatile unsigned gcMoveCount
```

This counter must be incremented when the collector moves objects. This counter allows other code in the JVM to perform "optimistic reads" of object values—the code can:

1. Check the value of `gcMoveCount`.
2. Dereference an object pointer *while consistent*.
3. Read a field in the object.
4. Compare the current value `gcMoveCount` with the saved value. If `gcMoveCount` has changed, then a GC could have occurred between steps 2 and 3, so go back to step 1.

This is more efficient than temporarily becoming inconsistent. This idiom is used by many idempotent LLNI operations.

`extern int` **`minPercentFreeAfterGC`**

The user needs some way to indicate which is more important in the current environment, time or space. No doubt this will evolve, but here is what we will provide for now. The heap is expanded if a (full) collection results in less than this fraction free. The default value is 40.

`extern bool_t` **`workStoppedConsistently`**

This variable is used to indicate that a thread has forced the VM to suspend all threads at consistent points (so the system is consistent).

`extern int` **`verbosegc`**

When > 0, collector may print diagnostics to `stderr`. Larger values allow more verbose output. Set to the number of times the flag `-verbosegc` appears in the command line.

3.12 Debugging and Verification

```
void sysAssert(int expression)
void sysVerify(int expression)
```

Like the C library `assert()` macro, these macros will print a diagnostic message and terminate the program when the `expression` is zero. `sysAssert()` is only enabled when the system is compiled with the `DEBUG` flag defined. `sysVerify()` is always enabled.

`extern int` **`verifyheap`**

If non-zero, the memory system will call `verifyHeap()` before and after GC. Set by the command line flag `-verifyheap`.

`void` **`verifyHeap`**`(char *prefixMsg)`

This function performs global heap verification. It may be slow, but is very useful for debugging. `prefixMsg` is printed out as part of a message to `stderr` indicating that the heap is being verified. The function terminates the program if verification fails.

`macro int` **`DEAD_MARKER`**

This macro is used to mark words in free areas when in debugging modes. Its current value is 0xDEADBEEF.

`void` **`bashWords`**`(word32 *start, uintptr_t numWords)`

This function stores a dead marker (`DEAD_MARKER`) in range of words. It can be used by a collector in debug mode to overwrite dead objects.

`bool_t` **`untouchedSinceBashed`**`(word32 *deadChunk, uintptr_t numWords)`

This function verifies that specified words contain `DEAD_MARKER`. A collector can use this in debug mode at allocations to ensure that recycled memory wasn't written to accidentally.

3.13 Memory Allocation Support

Underneath any collector are facilities to allocate chunks of memory from the operating system. The collectors implementing the GC interface may make use of services provided by two different sets of routines: the MemoryArea and the sys_api routines. The first set provides routines for mapping and unmapping areas of memory and represents an abstraction of the second. The latter set provides the lowest-level services.

3.13.1 MemoryArea

A memory area is an aligned, contiguous region `[start, start + byteSize)` of memory. `start` and `byteSize` are multiples of the OS page size. A memory area is further divided into subregions that are "committed" or "uncommitted." Uncommitted regions consume fewer resources (such as swap space) than committed regions. Here is the type of a memory area (defined in `gc/memarea.h`):

```
typedef struct {
    char     *start;
    uintptr_t byteSize;
} MemoryArea;
```

The main idea behind memory areas is to abstract away from the more complicated interfaces provided by operating system libraries supporting functions like `mmap()`.

`MemoryArea*` **`MemoryArea_create`**`(uintptr_t byteSize, int align)`

This function creates and returns a new memory area with the given `byteSize`. The exact size of the area is obtained by rounding up `byteSize` to be a multiple of the OS page size. The entire new memory area is uncommitted. On success, the function returns a non-NULL result. The area is guaranteed to be aligned on an `align` byte boundary.

`void` **`MemoryArea_destroy`**`(MemoryArea* ma)`

This function deallocates all the memory represented by `ma`, including `ma` itself.

`bool_t` **`MemoryArea_resize`**`(MemoryArea* ma, uintptr_t newByteSize)`

This function resizes `ma` so that at least `newByteSize` bytes are available. Let "alignedNewByteSize" be the smallest multiple of the OS page size at least as large as `newByteSize`. If the operation completes successfully, it returns `TRUE` and sets the extent of `ma` to `[ma->start, ma->start + alignedNewByteSize)`. There are three cases, depending on whether "alignedNewByteSize" is smaller, equal, or greater than the current `ma->byteSize`.

- If it is equal, the call returns `TRUE` and takes no action.
- If the new size is smaller, releases resources associated with freed memory, and returns `TRUE`. The committed/uncommitted regions remaining within the extent of `ma` are otherwise unaffected.
- If the requested size is greater than the new size, successfully added memory forms an uncommitted region. The previously committed/uncommitted regions are otherwise unaffected.

If it impossible to grow the memory area (e.g., because the memory at the high end of the current region is unavailable), returns `FALSE`.

```
bool_t MemoryArea_commitRegion(MemoryArea* ma, char *start,
                               uintptr_t byteSize)
```

This function requires that the `start` and `byteSize` are multiples of the OS page size, and that the region `[start, start+byteSize)` is a subregion of `ma`. It returns `TRUE` if it can successfully ensure that the subregion is committed. Otherwise, it returns `FALSE`. Committed memory is memory that has been allocated swap space.

```
void MemoryArea_decommitRegion(MemoryArea* ma, char *start,
                               uintptr_t byteSize)
```

This function requires that the start and byteSize are multiples of the OS page size, and that the region [start, start+byteSizee) is a subregion of ma. It attempts to decommit the subregion. On some systems, this may have no effect.

```
char *MemoryArea_limit(MemoryArea *ma)
```

This function returns the `char *` one past the end of the region controlled by `ma`. This address is a multiple of the OS page size.

3.13.2 sys_api

This is the layer that defines the lowest-level memory-related functions. There are functions for byte-level allocation and page-level allocation.

```
void* sysMalloc(size_t size)
void* sysCalloc(size_t nelem, size_t elsize)
void* sysRealloc(void *ptr, size_t size)
void  sysFree(void *ptr)
```

These functions are equivalent to the standard C storage allocation functions, but could be implemented differently.

```
extern size_t sysPageSize
```

This variable contains the size of an OS page.

```
void *  sysMapMem(size_t requestedSize)
```

This function attempts to allocate a region of address space of size at least `requestedSize`, which must be a multiple of the OS page size. The allocated memory is uncommitted; that is, it has no backing swap space. If successful, returns address of mapped region, else returns 0.

```
int     sysUnmapMem(void * requestedAddr, size_t requestedSize)
```

This function attempts to deallocate (unmap) the block of memory starting at `requestedAddr` whose size is `requestedSize`. If successful, returns 0, else returns -1.

```
void *  sysCommitMem(void * requestedAddr, size_t requestedSize)
```

This function attempts to commit (allocate swap space for) a region of memory starting at `requestedAddr` of size at least `requestedSize`, both of which must be a multiple of the OS page size. If successful, returns `requestedAddr`, else returns 0.

```
int     sysUncommitMem(void * requestedAddr, size_t requestedSize)
```

This function attempts to uncommit (deallocate swap space for) a region of memory starting at `requestedAddr` whose size is `requestedSize`, both of which must be a multiple of the OS page size. If successful, returns 0, else returns -1.

4. Open Issues and Future Directions

As described in the Introduction, the memory system interface will change (due in part to reader feedback). Here is a list of known problems and future enhancements.

- Currently, the JIT compiler has direct knowledge of the write barriers that it must generate. In fact, it doesn't support read barriers yet. Each collector has to implement a barrier-generating function using undocumented routines in the JIT compiler. There is a need to design and document a more flexible system.

- We have implemented only one concurrent collector—a “Baker-style” concurrent copying collector. The GC interface may or may not accommodate other concurrent collectors. In addition, we haven't yet implemented support for weak references in our concurrent collector. We need to design a flexible, concurrent system for dealing with weak references.

- We would like to add support for parallel GC—several processors working on GC at the same time.

- We may add generational support to the “abstract collector” interface. We currently have a flexible generational framework that is built on top of the “abstract collector” interface. We would like to be able to handle “multi-area” allocation and collection for cases like heaps-by-size (large object areas), heaps-by-type, as well as heaps-by-age (generational collection). For our own internal use, we should also document the existing generational framework.

- We have recently been experimenting with using compare-and-swap in place (or in addition to) LABs in the youngest generation. This choice also affects whether we use the fast allocator. We should document these choices and their consequences.

- 64-bit support has not been fully implemented. In this document, a “word” can be counted on to be 32 bits on a 32-bit system. It is not clear what it means on 64-bit system.

5. More Information and Feedback

5.1 References and Further Reading

1. Gosling, J., Joy, B., Steel, G., *The Java^{TM} Language Specification*. Addison-Wesley, 1996.
   `http://java.sun.com/docs/books/jls/`
2. *Java Native Interface Specification*. Sun Microsystems, 1997.
   `http://java.sun.com/products/jdk/1.2/docs/guide/jni/`
3. *Java^{TM} Platform 1.2 API Specification*. Sun Microsystems, 1998.
   `http://java.sun.com/products/jdk/1.2/docs/api/`
4. Jones, R., Lins, R., *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, 1996.
5. Lindholm, T., Yellin, F., *The Java^{TM} Virtual Machine Specification*. Addison-Wesley, 1996.
   `http://java.sun.com/docs/books/vmspec/`
6. Pawlan, M., *Reference Objects and Garbage Collection*. Java Developer Connection technical article, Sun Microsystems, 1998.
   `http://developer.java.sun.com/developer/technicalArticles/`
7. Wilson, P., *Uniprocessor garbage collection techniques.* Technical report, University of Texas, January 1994. Expanded version of the IWMM92 paper.
   `ftp://ftp.cs.utexas.edu/pub/garbage/bigsurv.ps`.
8. Jones, R., *Richard Jones' Garbage Collection Page*.
   `http://stork.ukc.ac.uk/computer_science/Html/Jones/gc.html`.
9. Chase, D., *GC FAQ*.
   `http://www.iecc.com/gclist/GC-faq.html`

Finally, this document (*The GC Interface in the EVM*) will be online
`http://www.sunlabs.com/technical-reports`.

5.2 Credits

The design of the memory system interface is a product of the Java Topics Group at Sun Microsystems Laboratories at Burlington, MA, which includes Ole Agesen, Corky Cartwright (Visiting Professor), Dave Detlefs, Christine Flood, Alex Garthwaite, Steve Heller, Tony Printezis (intern), Guy Steele, and Derek White.

Thanks to the Java Topics Group, Glenn Skinner, Mario Wolczko, Bernd Mathiske, and Eliot Moss for reviewing this document.

5.3 Feedback

Comments on the memory system design should be sent to `java-topics@east.sun.com`.
Comments on this document can be sent to `derek.white@east.sun.com`.

6. About the Authors

Derek White is a Staff Engineer at Sun Microsystems Laboratories in Burlington, Massachusetts. He is a member of the Java Topics Group, which has developed a JVM as an infrastructure for developing soft real time and scalable garbage collection for the Java language. The group has also studied and optimized JVM performance as a whole. Other interests include performance and heap analysis tools, and incremental development environments. Prior to joining Sun, Derek worked for eight years at Apple Computer, Inc., where he was responsible for the Object Pascal compiler, represented Apple on the ANSI Pascal committee, and worked on the Dylan programming language runtime and development environment.

Alex Garthwaite recently joined Sun Microsystems Laboratories in Burlington, Massachusetts, where he works with the Java Topics Group. His research interests include programming language implementation and runtime system design, automatic memory management, and synchronization techniques.

Alex is also a graduate student at the University of Pennsylvania working with Scott Nettles. He is currently working on a proposal for his doctoral thesis which investigates how runtime services in a Java VM may be safely implemented in Java.