MMTk: The Memory Management Toolkit

Steve Blackburn
Robin Garner
Daniel Frampton

September 28, 2006

The Hitch Hiker's Guide to the Galaxy is an indispensable companion to all those who are keen to make sense of life in an infinitely complex and confusing Universe, for though it cannot hope to be useful or informative on all matters, it does at least make the reassuring claim, that where it is inaccurate it is at least definitely inaccurate. In cases of major discrepancy it's always reality that's got it wrong.

Douglas Adams, The Restaurant at the End of the Universe

Contents

Chapter 1

Introduction

MMTk is a toolkit for writing high-performance memory managers. It currently provides the memory management subsystems of the JikesRVM Java virtual machine and the JNode operating system, and has been ported to the Rotor C# environment.

This manual is intended to be a comprehensive reference to MMTk for programmers who want to do memory management research, or who want to use MMTk as the memory manager for a language runtime or virtual machine. This is not a tutorial on memory management, and if you are new to garbage collection Jones and Lins' excellent book [1] could be considered required background reading. Neither is this manual intended to be a substitute for reading the code, rather, it attempts to bridge the gap between the two.

1.1 Design considerations

MMTk is carefully designed for high performance. Some of the techniques used are obvious in the design, and some are more subtle.

The design principles that MMTk uses are

1. **Make the frequent case fast.**

   And conversely, don't worry too much about the performance of infrequently used code. This principle underlies all of the other techniques in this list. Code that is not used frequently should be designed with elegance and maintainability as its first goal.

2. **Provide fast thread-local access to shared structures.**

   All of the data structures that MMTk uses frequently minimise locking overhead using this design principle. This is evident in the structure of allocators, queues, plans etc.

[1] 'Garbage Collection: Algorithms for Automatic Dynamic Memory Management', Richard Jones and Rafael Lins, John Wiley and Sons, 1996

3. **Avoid virtual dispatch on the fast path**

   When a method of an object whose type is not known exactly is called, Java calls the method indirectly by indexing into the classes virtual method table, which is not fast. One alternative is to use static methods, but this comes at the cost of some design flexibility, so it is preferrable to use instance methods, but code carefully so that the compiler maintains precise type information at the critical call sites. This is discussed in more detail later.

4. **Careful use of directed inlining**

   The InlinePragma and NoInlinePragma idiom allows Java VMs that support it to perform directed inlining. While inlining can improve performance dramatically, especially in the presence of exact type information, too much inlining can reduce performance.

5. **Rely on compile-time constant folding**

   Particularly when combined with inlining, this is a powerful technique for generating fast compact code while maintaining a strict separation of concerns in a multilayered software design.

Chapter 2

A Simple MMTk Tutorial

The goal of this chapter is to provide a basic introduction to MMTk. It is a walk-through of the process of building an incrementally more advanced collector. The intention is that by getting some practical experience building a simple collector, the remainder of the guide will make more sense.

It is estimated that blindly following the steps of the tutorial should take around half a day. The amount of time to really understand MMTk is highly variable, depending both on knowledge of memory management techniques and programming experience.

2.1 Preliminaries

Getting MMTk and Jikes RVM and Eclipse working.

- Download Jikes RVM version 2.3.7 or later (see the Jikes RVM home page and userguide).
- Make sure you can build and run a BaseBaseNoGC configuration (see the Jikes RVM userguide).
- Open MMTk in Eclipse (see Eclipse documentation for further information)
    1. Start Eclipse. If prompted provide a workspace directory somewhere in your home directory (not RVM_ROOT or its descendants).
    2. Select File → Import...
    3. Choose 'Existing Project into Workspace'
    4. Browse to the directory where your MMTk tree sits (`$RVM_ROOT/MMTk`) and select OK.
    5. Select Finish.
    6. Switch to the Java perspective.

2.2 Adding the Tutorial collector

The goal is to create a tutorial collector. In order to achieve this, we will use the NoGC collector in MMTk as a template.

2.2.1 Add a new Jikes RVM configuration.

At the time of writing, the non-MMTk parts of JikesRVM can't be edited in Eclipse[1], so the following steps need to be done at the command line.

For the Tutorial GC, we use the NoGC configuration as a template.

1. Copy the GC configuration file
   ```
   # cd $RVM_ROOT/config/build/gc
   # cp NoGC Tutorial
   ```

2. Edit Tutorial, changing the line:
   ```
   export RVM_WITH_JMTK_PLAN="org.mmtk.plan.nogc.NoGC"
   ```
   to
   ```
   export RVM_WITH_JMTK_PLAN="org.mmtk.plan.tutorial.Tutorial"
   ```

3. Copy the build configuration file.
   ```
   # cd $RVM_ROOT/config/build
   # cp BaseBaseNoGC BaseBaseTutorial
   ```

4. Edit BaseBaseTutorial, changing the line:
   ```
   .  $1/gc/NoGC
   ```
   to
   ```
   .  $1/gc/Tutorial
   ```

2.2.2 Create the Tutorial package

Clone the NoGC plan using Eclipse.

1. Copy the `org.mmtk.plan.nogc` package, renaming it to `org.mmtk.plan.tutorial`. (Highlight the `org.mmtk.plan.nogc` package, and select Edit→Copy followed by Edit→Paste.)

2. Rename each of the classes within the new tutorial package, replacing the substring 'NoGC' with 'Tutorial'. (Use Eclipse's rename refactoring (Refector→Rename...) with its default options (update references) to do this automatically, one class at a time). If you look inside any of the renamed classes you should find that they are properly renamed, with new class names, constructors etc. replacing 'NoGC' with 'Tutorial' throughout.

[1] To be more precise, it can be edited in eclipse, but not as a Java project

Make sure you can build and run the new BaseBaseTutorial configuration. It should behave *exactly* the same as the BaseBaseNoGC configuration (it should fail with an out of memory error as soon as it has exhausted the maximum heap size specified on the command line with -Xms). To run the SPECjvm98 jess benchmark, for example:

1. `# cd` to a directory containing SPECjvm98
2. `# rvm -Xmx25M -Xms25M -X:gc:verbose=3 -X:gc:ignoreSystemGC=true SpecApplication _202_jess`

The above will set the minimum (`-Xms25M`) and maximum (`-Xmx25M`) heap size to 25MB in which to run the benchmark, and will produce verbose (`-X:gc:verbose=3`) GC messages. In order to run for longer, we tell MMTk to ignore calls to `System.gc()` (`-X:gc:ignoreSystemGC=true`). The rest of the command line arguments are specific to SPECjvm98. For a full list of command line arguments run `# rvm -X:gc`, and for default values, run `# rvm -X:gc:printOptions`.

2.3 Mark-Sweep

Change the Tutorial plan to use a MarkSweep garbage collector. It currently uses a bump pointer allocator and does not perform collection.

2.3.1 Change allocation policy

Use a free list allocator instead of a bump-pointer. A MarkSweepSpace is a region of virtual memory that uses a segregated free list to allocate memory. A MarkSweepLocal is a thread-local object that provides fast unsynchronized allocation from a MarkSweepSpace.

1. In `TutorialConstraints`, update GC header requirements:
   - `gcHeaderBits()` should return `MarkSweepSpace.LOCAL_GC_BITS_REQUIRED`
   - `gcHeaderWords()` should return `MarkSweepSpace.GC_HEADER_WORDS_REQUIRED`
   - Add the appropriate import statement.
2. In `Tutorial`, replace the `ImmortalSpace` with a `MarkSweepSpace`:
   - change the appropriate import statement
   - change the type of `defSpace` and change the constructor appropriately.
   - rename `defSpace` to `msSpace` (right-click, Refactor→Rename...)
   - rename `DEF` to `MARK_SWEEP` (right-click, Refactor→Rename...)
   - at this point `TutorialLocal` will be broken (for a moment...)

3. In `TutorialLocal`, replace the `ImmortalLocal` (a bump pointer) with a `MarkSweepLocal` (a free-list allocator):
   - change the appropriate import statement
   - change the type of `def` and change the constructor appropriately.
   - rename `def` to `ms` (right-click, Refactor→Rename...)
   - add an extra argument to `alloc()`: `ms.alloc(bytes, align, offset, false);` because the free list allocator needs to know whether the allocation is occuring during GC (since alloc is never called during GC, we just pass the value false).
4. Fix `postAlloc()` to initialize the mark-sweep header:
   - add the following clause:

```
if (allocator == Tutorial.ALLOC_DEFAULT)
  Tutorial.msSpace.initializeHeader(ref);}
```

With these changes, Tutorial should now work, just as it did before, only exercising a free list (mark-sweep) allocator rather than a bump pointer (immortal) allocator. Create a BaseBaseTutorial build, and test your system to ensure it performs just as it did before. You may notice that its memory is exhausted slightly earlier because the free list allocator is slightly less efficient in space utilization than the bump pointer allocator.

2.3.2 Mark-sweep collection

The next change required is to perform mark-and-sweep collection whenever the heap is exhausted. The `poll()` method of a plan is called at appropriate intervals by other MMTk components to ask the plan whether a collection is required.

1. In `Tutorial.poll()`, first insert a guard to ensure collections are not triggered at the wrong times:

```
if (getCollectionsInitiated() > 0 || !isInitialized() ||
    space == metaDataSpace)
        return false;
```

2. Then in `Tutorial.poll()`, trigger a collection when the heap is exhausted:
   - Remove the `Assert` statement.
   - Add the following:

```
if (getPagesReserved() > getTotalPages()) {
   Collection.triggerCollection(Collection.RESOURCE_GC_TRIGGER);
   return true;
}
```

- Add the appropriate import statement.

3. Next, the plan needs to know how to perform a garbage collection. Collections are performed in phases, coordinated by data structures defined in `StopTheWorld`, and have global and thread-local components.

   In `Tutorial`, add global collection phases:

   - Make `Tutorial` extend `StopTheWorld` rather than `Plan`.
   - Remove the `Assert` call in `Tutorial.collectionPhase()`
   - Using the commented template in `Tutorial.collectionPhase()`, set the following within the clause for `phaseId == PREPARE`

     ```
     super.collectionPhase(phaseId);
     trace.prepare();
     msSpace.prepare();
     return;
     ```

     This code prepares the mark-sweep space for collection (flipping the mark state, among other things.
   - Within the clause for `phaseId == RELEASE`

     ```
     trace.release();
     msSpace.release();
     super.collectionPhase(phaseId);
     return;
     ```

     This releases the spaces from collection mode, and prepares to allow the mutator to run again.
   - Ensure that the default fall through (to `StopTheWorld` is uncommented.

4. In `TutorialLocal`, add local collection phases:

   - Make `TutorialLocal` extend `StopTheWorldLocal` rather than `PlanLocal`.
   - Remove the `collect()` method in `TutorialLocal`
   - Remove the `Assert` call in `TutorialLocal.collectionPhase()`.
   - Using the commented template in `TutorialLocal.collectionPhase()` (being careful to replace `NoGC` with `Tutorial`), set the following within the clause for `phaseId == Tutorial.PREPARE`

     ```
     super.collectionPhase(phaseId, participating, primary);
     trace.prepare();
     ms.prepare();
     return;
     ```

   - Within the clause for `phaseId == Tutorial.START_CLOSURE`

```
trace.startTrace()
return;
```

This calls the method that traces the heap from the root set.

- Within the clause for `phaseId == Tutorial.COMPLETE_CLOSURE`

```
trace.completeTrace()
return;
```

This completes a trace, picking up any objects that may have come alive by having finalizers, for example.

- Within the clause for `phaseId == Tutorial.RELEASE`

```
trace.release();
ms.release();
super.collectionPhase(phaseId, participating, primary);
return;
```

- Ensure that the default fall through (to `StopTheWorldLocal` is uncommented.

With these changes, Tutorial should now work correctly as a mark-sweep collector, allocating with a free list allocator and collecting the heap each time it is exhausted. Create a BaseBaseTutorial build, and test your system to ensure it performs correctly. You should find that it correctly performs collections (this will only be visible if you have set the `-X:gc:verbose` flag appropriately).

2.4 Hybrid Copying/Mark-Sweep

Extend the Tutorial plan to create a "copy-MS" collector, which allocates into a copying nursery and at collection time, copies nursery survivors into a mark-sweep space. This plan does not require a write barrier (it is not strictly generational, as it will collect the whole heap each time the heap is full). Later we will extended it with a write barrier, allowing the nursery to be collected in isolation. Such a collector would be a generational mark-sweep collector, similar to GenMS. *Explanation of nursery, nursery survivors.*

2.4.1 Add a copying nursery

- In `TutorialConstraints`, override the `movesObjects()` method to return true, reflecting that we are now building a copying collector:

```
public boolean movesObjects() {
  return true;
}
```

- In `Tutorial`, add a nursery space
  1. Create a new space, `nurserySpace`, of type `CopySpace`, and make it and `msSpace` each share 0.3 of the available heap (change from 0.6). The new copy space will initially be 'from-space', so provide false as the final (additional) argument to the `CopySpace` constructor. In the constructor argument change the string value to something like `"nursery"`.
  2. add the appropriate import statement
  3. Add a new space descriptor, `NURSERY` initialized with `nurserySpace.getDescriptor()`
  4. Add `nurserySpace` to the prepare and release phases of `collectionPhase()`, passing the flag true to the prepare phase, indicating that the nursery is treated as 'from-space' during the collection.
  5. Fix accounting to account for this new space:
     (a) Add nurserySpace to the equation in `getPagesUsed()`,
     (b) Add a method to override `getCopyReserve()` which returns `nurserySpace.reservedPages()`,
     (c) add a method to override `getPagesAvail()`, returning `(getTotalPages() - getPagesUsed())>` which allows for a copy reserve on the assumption that future allocation will go to the nursery and in the worst case will need to be copied.
  6. In `TutorialLocal`, add a nursery allocator. Add an instance of `CopyLocal` calling it `nursery`. The constructor argument would be `Tutorial.nurserySpace`.
  7. Change `alloc()` to allocate into nursery... return `nursery.alloc(bytes, align, offset);`
- Remove mark-sweep initialization in `postAlloc()`, since no special post allocation initialization is required now.
- Override `allocCopy()` with a new method which allocates into ms when the allocator is `ALLOC_DEFAULT`, passing `Tutorial.msSpace.inMSCollection()` as the final argument to `ms.alloc()`. Calls to `allocCopy()` with other allocator values should fall through to `super.allocCopy()`. Override `postCopy()` with a new method which does the following (the following is currently necessary, but can be considered to be a design bug, which we hope to fix soon):

```
Tutorial.msSpace.writeMarkBit(ref);
MarkSweepLocal.liveObject(ref);
```

- Add `nursery.reset()` to the release phase of `collectionPhase()`. In `TutorialLocal.getSpaceFromAllocator()` add the statment

```
if (a == nursery) return Tutorial.nurserySpace;
```

- In `TutorialLocal.getAllocatorFromSpace()` add the statement

  ```
  if(space == Tutorial.nurserySpace) return nursery;
  ```

- In `TutorialTraceLocal`, add tracing mechanisms to accommodate the new nursery.

  1. Add a clause to `isLive()` which calls `Tutorial.nurserySpace.isLive(object)` if the object is in the nursery space.
  2. Add a clause to `traceObject()`, which calls `Tutorial.nurserySpace.traceObject(this,` if the object is in the nursery space.
  3. Create a method that overrides `precopyObject()`, which returns null for null objects, calls `Tutorial.nurserySpace.traceObject(this, object)` if the object is in the nursery space, and returns the object otherwise.
  4. Create a method that overrides `willNotMove()`, which returns Space.isInSpace(Tutorial.NURSERY, object);!
  5. Create a method which overrides `getAllocator()` which returns `Tutorial.ALLOC_DEFAULT`.

With these changes, Tutorial should now work correctly as a copy-MS collector, allocating with a bump pointer, promoting into a mark-sweep space, and collecting the heap each time it is exhausted. Create a BaseBaseTutorial build, and test your system to ensure it performs correctly. You should find that it correctly performs collections (this will only be visible if you have set the `-X:gc:verbose flag` appropriately). By reading the verbose GC output you should see data being collected from and allocated to the nursery and mark sweep spaces.

2.5 Generational Mark-Sweep

watch this space ...

Chapter 3

Plans

3.1 Overview

A `Plan` is the highest-level entity in the MMTk hierarchy. Each of the collection/memory management schemes that MMTk provides is defined by a single instance of a `Plan`. The role of a `Plan` is to compose lower-level objects in the MMTk hierarchy into a coherent scheme that provides

- The layout of virtual memory, by creating a set of `Spaces`
- Memory allocation
- Garbage collection
- Statistics (since MMTk is a research-oriented toolkit, reporting what is going on in the memory manager is an important goal)

In earlier releases of MMTk, a `Plan` was a single class that provided all the above functions, but currently a `Plan` is represented by a package with 3 or more classes. The parent package for all of the MMTk plans is `org.mmtk.plan`, and the abstract superclasses that provide much of the common functionality resides here. Individual plans are held in deeper levels of the package hierarchy under this parent package.

3.1.1 Global and local data

MMTk is designed for multi-processor environments, and needs to consider issues such as synchronization and cache coherence. MMTk achieves high performance by ensuring that the most frequent operations are performed on dedicated thread-local data structures that require no synchronization, and infrequently operating on global data structures, but amortizing the cost across the faster more frequent case.

One example of this is a bump-pointer allocator, that underlies the spaces used in many of MMTk's copying collectors. The copying memory region as

a whole is managed by a `CopySpace` object, with synchronized methods that allocate memory in large chunks. Each processor has a `CopyLocal` object (an allocator), which essentially caches a chunk of memory for fast thread-local access, periodically calling back to CopySpace to allocate another chunk.

This design pattern is realised by two of the three required classes for a `Plan`, which are the global plan, a subclass of the `Plan` class, and the local plan which is a subclass of the `PlanLocal` class.

3.2 Spaces

The core of a plan is a collection of regions of virtual memory, each managed under an allocation/collection policy, and orchestrated to form a memory management plan for a runtime environment. Some of the spaces MMTk provides currently are

- Immortal space. Bump-pointer allocation and no collection.
- Mark-sweep space. A free-list allocator managed under a mark-sweep collection policy with lazy sweeping.
- Large object space. Baker's treadmill space.
- Copy space. A bump-pointer allocator with copying collection. A single copy space can be used as a nursery in conjunction with a suitable mature space, or a pair can be used for a semi-space collector.
- VM space. This is a space where no allocation takes place, but MMTk will happily scan objects. This is used to denote the virtual memory region where the virtual machine or runtime lives.

  MMTk assumes that its host virtual machine will have pointers into the heap, and possibly there will be heap objects that point to data structures in the host virtual machine. The host virtual machine takes care of initializing this space.
- Raw memory space. MMTk creates its own data structures in raw memory so that it can avoid circularity issues.

Spaces themselves are described in detail in Chapter 4.

The base class Plan defines 4 spaces: a VM space, a small immortal space, a raw memory space for collector metadata, and a large object space. Most subclasses will define one or more new spaces, and must override methods in the Plan classes to handle operations on these new objects. Most of the time this can be done by identifying whether the object in question is in a space defined at this level, and dispatching to the appropriate method of the `Space`, and delegating to the superclass if not.

The actual layout of virtual memory is dictated by the constructors that create the Spaces, and the order in which they are executed. Each constructor implements a slightly different set of methods for specifying how it is to consume virtual memory. The most straightforward is a space which occupies a certain percentage of virtual memory, and is allocated at the lowest available addresses. Another option, useful when creating nursery spaces, is to request contiguous memory at the high end of virtual memory[1].

3.3 Barriers

A barrier is a code sequence that is used to perform additional work when the VM reads or writes to the heap. MMTk currently supports *write barriers* on pointer stores, but there are patches available to extend this to *read barriers* on pointer loads. Some researchers have also implemented barriers on non-pointer loads and stores, but these are not widely used. This section has been written assuming that read barrier support is included, but note that this is an optional patch and not yet supported in a released version of JikesRVM/MMTk.

Barriers, particularly read barriers can be particularly difficult to implement correctly, and can be a nightmare to debug, so a carefully structured approach is necessary.

3.3.1 Write barriers

Write barriers are essential for generational collectors, so it is no surprise that these are well developed and stable features of MMTk. Write barriers are enabled by the `requiresWriteBarrier` method of the `Constraints` class of a plan. The write barrier itself is the method `writeBarrier` in the `PlanLocal` class.

There are two forms of write barrier in MMTk, one for single-field store operations, and a second for array copies (since the barrier could make use of various optimizations in the array copy case). MMTk's write barriers are *substituting* barriers, ie they must actually effect the store.

For a single field barrier, a no-op write barrier is as follows:

```
public void writeBarrier(ObjectReference src,
    Address slot, ObjectReference tgt, Offset metaDataA,
    int metaDataB, int mode) {
  Barriers.performWriteInBarrier(src,slot,newTgt,
                                 metaDataA,metaDataB,mode);
}
```

One obvious requirement of a write barrier is that it must not trigger itself when it performs the pointer write. This effectively means that all writes

[1] This allows write-barriers to identify nursery objects with a single comparison

to heap objects in barriers must be done using the `store()` methods of `org.vmmagic` classes, which are required not to trigger a barrier.

One unfortunate effect of a substituting write barrier is that optimizing compilers can lose some type information. At a high level, the compiler generates code to store a typed value into a field, and this value is converted to a magic `ObjectReference` type before being stored. The solution to this (for JikesRVM and hopefully future optimizing compilers) is to provide placeholder parameters to the write barrier, where the compiler can store some metadata, and to perform the pointer write using the method shown above, which allows the host VM to recover the type information it has passed in to the barrier.

3.3.2 Read barriers

Read barriers are more complex and unfortunately their implementation details are somewhat dictated by limitations of JikesRVM (or rather, of the read-barrier maintainers' understanding of JikesRVM).

Read barriers are enabled by the `Constraints` method `needsReadBarrier`. The read barrier is implemented in three methods of a `PlanLocal`, `readBarrier`, `preReadBarrier` and `postReadBarrier`. The code for readBarrier invokes the pre- and post- barriers and performs the read (ie it is a substituting barrier). Unfortunately, substituting barriers have not been made to run adequately reliably with JikesRVM's optimizing compiler, but non-substituting barriers (ie pre- and post-barriers) work well. The current JikesRVM implementation uses the substituting read barrier for baseline compiled code, and the non-substituting barriers for opt-compiled code, and other compilers are free to use either method. For this reason, it is best if the substituting read barrier code be left alone (if at all possible), and the barrier implementation be confined to pre- and post-barriers. It is probable that passing type information through the read barrier in a similar fashion to the write barrier will solve the problem.

The obvious limitation on code running inside a read barrier is that it must not trigger a read barrier itself. The simplest way to ensure this is to only use the load and store methods of magic types to access the heap.

Performance is even more important for a read barrier, as pointer reads are significantly more frequent than pointer writes.

3.4 Accounting

Another responsibility of a `Plan` is to account for the space it uses. In particular copying collectors must ensure that their copy reserve is accounted for when determining how much space is being used.

Space usage is reported to the rest of MMTk through 4 static methods, all of which return an Extent.

totalMemory() Reports the total amount of memory available. This will be somewhere between the values of the `-Xms` and `-Xmx` command line arguments.

freeMemory() Reports the amount of unused memory, which may include memory held in reserve for copying.

usedMemory() The amount of memory currently allocated.

reservedMemory() The amount of memory used, plus memory reserved for copying.

The internal calculation of space usage is done in pages, and is implemented by the following methods:

getTotalPages() Returns the current size of the heap, in pages.

getPagesAvail() Pages available for allocation, after used and reserved memory is accounted for.

getPagesReserved() Pages currently reserved, defined as pages in use plus copy reserve.

getPagesUsed() Pages currently in use. Each plan needs to override this method to account for pages used by spaces defined at their level of the hierarchy. The usual formula is to add the super-class's contribution to that of spaces defined in the current plan instance.

getCopyReserve() Pages that are not actually used, but are nonetheless unavailable for allocation. For a copying collector, this is the copy reserve, although non-copying collectors may wish to reserve space for other purposes. Plans are required to override this method if they require space to be reserved.

The accounting system is used primarily by the `poll()` method, as a contributor to the decision of when to perform a garbage collection.

3.5 Allocation

The local instance of a plan provides methods that the virtual machine or runtime uses to allocate objects. This is one of the three most performance critical operations that an MMTk plan provides, and is intended to be inlined into the mutator code at each allocation site, so great care must be taken.

MMTk collectors provide allocation through four methods:

alloc() Initial allocation of objects.

allocCopy() Allocate an object during a copying garbage collection phase.

postAlloc() Post-allocation gc-specific initialization. This is to be called after the mutator's initialization of the object.

postCopy() Post-copy initialization of an object.

The `alloc()` and `allocCopy()` methods both allocate space from a mutator-specified allocator, which should be statically resolvable at compile time. The expected pattern for these methods is

```
Address alloc(int bytes, int align, int offset, int allocator)
  throws InlinePragma {
  if (allocator == Plan.ALLOC_DEFAULT) {
    return <allocator>.alloc(bytes, align, offset, false);
  }
  return super.alloc(bytes, align, offset, allocator);
}
```

ie, the method is inlined (into the mutator code), we dispatch to the `alloc()` method of the local allocator for spaces defined at this level of hierarchy (using a statically resolvable comparison), and delegate upwards for all other spaces.

The `allocCopy()` method takes an additional parameter (the object being copied) and needs to inform the allocator that a GC is in progress (by passing `true` as the last parameter, but should otherwise be identical.

`postAlloc()` and `postCopy()`

3.6 Collection

Garbage collection is initiated by calling the collect() method of `PlanLocal()`. It is the responsibility of the virtual machine to ensure that this method is called simultaneously on each processor in a parallel system, as one of the first things MMTk does in a collection is to perform a rendezvous, which will hang if all processors are not participating.

The exact codepath for initiating a collection actually seems rather convoluted at first sight, because it is in fact MMTk that decides when to collect, not the VM. This is

There are three main components to a collection:

- The `poll()` method. Allocators call this periodically to allow the plan to decide when to do a collection.
- The collection phases, implemented by the `collectionPhase` method of both local and global plan classes.
- Tracing the heap, implemented in the `Trace` and `TraceLocal` classes of a plan.

3.6.1 Initiating a collection

Garbage collection is initiated by one of two means: the mutator can explicitly request a GC (eg in Java with the `System.gc()` method), or the memory manager can decide that the heap is too full to satisfy an allocation request, and it needs to first reclaim memory by doing a garbage collection.

As discussed above, memory is allocated to the mutator via the `alloc()` method. This in turn requests memory from an allocator (associated with a policy), and periodically the allocator calls the `poll()` method of a plan. This call is done on the slow-path of the allocator, according to the poll frequency specified when a given space is created.

The `poll()` method evaluates the current state of the heap according to a variety of criteria, and if it decides a collection is required, it calls out to the host VM to request a collection (`Collection.triggerCollection()`). This allows the VM to prepare itself for collection, quiesce the other threads and call `PlanLocal.collect()` for each virtual CPU.

Once collection is complete, `poll()` returns `true` to the allocator, which can then perform any post-collection recovery it needs to and satisfy the allocation request.

A user-initiated collection is much simpler, with the mutator initiating the process somewhere inside the call to `Collection.triggerCollection()`.

3.6.2 Collection phases

A garbage collection generally proceeds in a structured way through several different phases. For example, a full-heap copying collector will enumerate the roots, perform a transitive closure over the root-reachable objects, process reference types and objects with finalizers (if supported by the host VM/language), possibly perform the transitive closure again, and then tidy up and complete. Between each of the phases a parallel collector must synchronize. There is also housekeeping work such as gathering statistics, and global state changes such as flipping from/ to space or changing a mark state bit.

Collections in MMTk are performed by a method called `collectionPhase()` defined for both local and global plan instances. This method takes an integer parameter, phaseId, which identifies the phase of garbage collection begin performed. These phases are defined at the layer of the plan class hierarchy immediately below `Plan`, by defining a collection of `Phase` objects.

Each `Phase` represents a step in the garbage collection process performed in parallel across all collector threads. A phase can have thread-local and/or global components, performed in either order. Threads can also have associated timer objects, either implicitly created for the phase or created externally so that multiple phases can accumulate time to a single timer. There are two types of Phase: simple and complex. A complex phase is

a list of phases, which are executed in order, while a simple phase is one implemented by the collectionPhase method. A garbage collection is defined to be a complex collection phase, which is executed by the plan's `collect()` method.

`StopTheWorld.java` and its local counterpart are the root of the plan hierarchy for most of MMTk's collectors. This layer defines collection phases for a classic stop-the-world collector. A typical collection phase is implemented like this:

```
public void collectionPhase(int id) throws InlinePragma {
  ...
  if (id == PREPARE) {
    super.collectionPhase(id);
    // Do the necessary work
    return;
  }
  ...
  super.collectionPhase(id);
}
```

Each layer in the plan hierarchy should contribute code to collect the resources defined at that level, and then delegate up the class hierarchy. The order in which collectors delegate and perform their own work is important and can lead to subtle problems, so some care is required. If in doubt, look at the order the existing collectors do things in, and copy that. The `InlinePragma` on the `collectionPhase` method ensures that precise type information is passed to method calls wherever possible, and helps to improve the performance of the tracing code.

3.6.3 Tracing the heap

At some point, all garbage collectors need to discover which objects are live by enumerating the pointers in a known live object. Tracing collectors go on to perform a transitive closure operation over the set of live objects, and reference counting objects use this information to perform decrements when an object is collected. This process is encapsulated in MMTk plans inside a `Trace` object (with local and global components).

The process is best understood in the context of a full-heap tracing collector such as the Mark-Sweep collector. The core of the mark sweep trace in MMTk is:

- Enumerate the roots, pushing the locations of the root pointers into a queue.
- Iterate through the root queue, marking the objects pointed to as reachable, and pushing the locations of currently untraced objects onto the work queue.

- Iterate over the work queue, marking objects, enumerating their pointers and enqueueing reachable objects for scanning. The iteration terminates when the queue is empty. This iterative scanning process is the most performance critical operation in most collectors.

The MMTk collectors mostly implement parallel tracing, using queue structures with fast thread-local and slower shared data pools.

The global instance of the tracing class usually holds just the shared components of the tracing queues, and none of the existing collectors need to extend the base class `Trace.java`. Each shared queue in `Trace` has an unsynchronized component in the `TraceLocal` class.

Tracing the heap must also support various flavours of weak references and finalization. For most collectors, it usually only makes sense to ask whether an object is live or not in the time between tracing the heap and allowing the mutator to resume. The `Trace` class provides several methods that allow reference types to query the status of objects, and potentially resurrect them (eg for finalization).

The key methods provided by the `TraceLocal` class are:

addRootLocation() Add a location to the root set. Locating the root set is a VM-dependent operation, and MMTk passes the `TraceLocal` object to a method the VM provides, and the VM calls this method to add roots to the queue. See the `ROOTS` phase of `StopTheWorld`'s `collectionPhase` method.

addInteriorRootLocation() As for root locations, this allows for roots that are in fact interior pointers.

startTrace() Perform the initial trace of the root set.

completeTrace() Perform the transitive closure over the heap.

enqueue() Enqueue an object for tracing.

traceObject() Trace an object. The actual tracing operation depends on the collection policy of the space in which the object lives, so this method generally finds the space that the object lives in and calls the space's trace method. In order to accommodate copying collectors, this method returns the new address of the object.

traceObjectLocation() Given the address of a pointer (a 'slot'), trace the object it points to, and install the resulting pointer.

Chapter 4

Policies

A policy in MMTk implements a region of virtual memory with an associated allocation and collection mechanism. As usual with MMTk structures, a policy consists of a synchronized global component and an unsynchronized local component. The separation of concerns between the global and local components is more pronounced than with the Plan hierarchy, and global policies are known as `Spaces`, while local policies are `Allocators`.

4.1 Spaces

A `Space` manages a range of virtual memory[1] according to a given allocation and collection policy.

The root of the class hierarchy is the abstract `Space` class. The public interface provided by this class is

Constructors MMTk provides several ways to specify the layout of virtual memory, and these are encapsulated in the constructors of a `Space`. The most commonly used constructors lay out virtual memory sequentially from a defined base address. The constructors allow the size of a space to be specified as either a fixed size (used for spaces such as the VM space and the metadata space) or a percentage of available virtual memory.

The other layout option is to require one space to occupy either the highest or lowest range of addresses. This provides a very efficient test of whether an object occupies this space, and is suitable for spaces such as the nursery in a generational collector, where object membership must be tested in a write barrier.

Basic geometry information Which describe the layout of a space: its lowest and highest addresses etc.

[1] Currently restricted to be a contiguous space.

Accounting Numbers of pages committed and reserved *explain.*

Tracing An abstract `traceObject()` method, plus `prepare()` and `release()`.

Liveness Abstract method `isLive()`, and `isReachable)()`

Coarse grained allocation A `Space` provides synchronized allocation in units of pages.

Closely associated with a `Space` is the `SpaceDescriptor` class, which the encoding of the 'vital statistics' of a space into an int. When used in a plan to create a `static final int` descriptor for a space, it allows fast tests for space membership to be created by a sufficiently good compiler. For example, in the MarkSweep plan, we create the space descriptor MS as follows

```
Space ms = new MarkSweepSpace( ... );
static final int MS = ms.getDescriptor();
```

and can subsequently write

```
if (Space.isInSpace(MS,object)) {
  ...
}
```

with reasonable confidence that the `isInSpace()` method will be expanded by the compiler to code that is a moderately optimal membership test.

4.2 Allocators

Chapter 5

Utilities

Chapter 6

Portability

Chapter 7

Java Extensions

It is not possible to do low-level systems-style programming in Java 1.4, so MMTk requires its host Java compilers to support a modest set of extensions to the language. These extensions come in the form of a set of Java classes which are implemented as 'unboxed' words, and a set of interfaces and exceptions that direct the code generation of the underlying compiler.

7.1 Unboxed types

One basic essential operation for a memory manager is to manipulate objects in memory without regard to their type, and to perform address arithmetic and other calculations, as well as to access the metadata areas of objects. The `org.vmmagic.unboxed` package defines objects such as `Address`, with methods like `load`, `add` and `store`, which access memory directly. The compiler is required to implement an `Address` as a single machine word, and is expected to implement its methods with inlined optimal instruction sequences.

The objects provided by `org.vmmagic.unboxed` are:

Word

Address

Offset

Extent

ObjectReference

7.2 Pragmas

Appendix A

VM Interface Specification

MMTk's interface to the host VM is specified as a package, `org.mmtk.vm`, containing several classes. MMTk provides stub implementations of these classes that it compiles against; the VM must provide the real implementations at run time.

In the descriptions given below, the method signatures of the interface methods are given. Since this is an interface, all methods are declared `public`, but in the interests of saving space, the `public` modifier has been elided from the definition given.

A.1 The active plan

MMTk by definition is a toolkit that provides several memory management schemes from which its host VM can choose. For performance reasons, the VM should define a `final` sub-class of whichever `Plan` it intends to use, and it must create one global instance and one local instance per processor. The `ActivePlan` class allows MMTk to locate these instances.

```
static final Plan global()
static final PlanLocal local()
static final PlanLocal local(int id)
```

These methods return the active global and local plan instances.

```
static final PlanConstraints constraints()
```

Returns the current constraints object instance.

```
static final int localCount()
```

The number of registered PlanLocal instances (should be equal to the number of CPUs).

```
static final int registerLocal(PlanLocal local)
```

Register a new PlanLocal instance. Returns a unique identifier which can be used to retrieve the instance later (calling `local(int)`).

A.2 Assertions

While MMTk makes heavy use of assertions, it does not use the Java `assert` statement because it aims to interact nicely with various host runtimes. Providing access to assertions in this way allows the runtime to exit gracefully in ways that might not be possible if the Java compiler were to use its own assertion checking mechanism.

```
static final boolean VERIFY_ASSERTIONS;
```

Whether assertions are being checked. The code idiom for checking assertions is

```
if( Assert.VERIFY_ASSERTIONS )
  Assert._assert(...);
```

and since `VERIFY_ASSERTIONS` is declared final, we rely on the compiler's dead code elimination to remove the assertion completely when we are compiling for performance.

There are three ways that MMTk can ask the VM to fail on error—this is probably due for rationalization.

```
static void error(String str)
```

Requests that the VM fails with an error.

```
static void fail(String message)
```

Requests that the VM fails with a stack trace and an error.

```
static void exit(int rc)
```

Requests that the VM fails with an error.

```
static void _assert(boolean cond)
static void _assert(boolean cond, String s)
```

Assert that a condition is true, and fail if it is not, optionally with a message. It is a checked error to call this if `VERIFY_ASSERTIONS` is `false`.

```
static final void dumpStack()
```

Print a stack dump, but don't exit. Useful for some forms of verbose debugging output.

```
static void failWithOutOfMemoryError()
```

Throw an out of memory exception, giving the VM a chance to prepare.

```
static boolean runningVM()
```

Are we running, or in some kind of initialization/build phase.

A.3 Barrier support

For a Java in Java implementation, read and write barriers require special support, especially if they are substituting barriers. The class `Barriers` provides the necessary support.

```
static void setArrayNoBarrier(char[] dst, int index,
                                        char value)
```

Sets an element of a char array without invoking any write barrier. This method is called by the Log class, as it will be used during garbage collection and needs to manipulate character arrays without causing a write barrier operation.

```
static void performWriteInBarrier(ObjectReference ref,
  Address slot, ObjectReference target, Offset offset,
  int locationMetadata, int mode)
```

Perform the actual write of the write barrier. The parameters specify the object being written to, the address of the pointer field being updated, the new value of the field, and the mode (PUTFIELD, PUTSTATIC, AASTORE).

The `offset` and `locationMetadata` parameters allow the optimizing compiler to preserve type information across the write barrier - the write barrier is invoked with these parameters, and MMTk simply passes them back in this call. This is currently JikesRVM specific.

```
static ObjectReference performWriteInBarrierAtomic(
  ObjectReference ref, Address slot, ObjectReference target,
  Offset offset, int locationMetadata, int mode)
```

Atomically write a reference field of an object or array and return the old value of the reference field.

```
static char getArrayNoBarrier(char[] src, int index)
static byte getArrayNoBarrier(byte[] src, int index)
static int getArrayNoBarrier(int[] src, int index)
static Object getArrayNoBarrier(Object[] src, int index)
static byte[] getArrayNoBarrier(byte[][] src, int index)
```

Gets an element of an array without invoking any read barrier, performing a bounds check or allowing a thread switch.

A.4 Collection

The `Collection` class allows MMTk and the VM to perform garbage collections.

Either party can trigger a collection by calling the `triggerCollection` method. The VM is required to prepare itself for collection, and then schedule a thread on each running processor which calls the MMTk plan's `collect` method.

```
static final int UNKNOWN_GC_TRIGGER = 0;
```

An unknown GC trigger reason. Signals a logic bug.

```
static final int EXTERNAL_GC_TRIGGER = 1;
```

Externally triggered garbage collection (eg call to System.gc())

```
static final int RESOURCE_GC_TRIGGER = 2;
```

Resource triggered garbage collection. For example, an allocation request would take the number of pages in use beyond the number available.

```
static final int INTERNAL_GC_TRIGGER = 3;
```

Internally triggered garbage collection. For example, the memory manager attempting another collection after the first failed to free space.

```
static final int TRIGGER_REASONS = 4;
```

The number of garbage collection trigger reasons.

```
static final double OUT_OF_MEMORY_THRESHOLD = 0.98;
```

The percentage threshold for throwing an OutOfMemoryError. If, after a garbage collection, the amount of memory used as a percentage of the available heap memory exceeds this percentage the memory manager will throw an OutOfMemoryError.

```
static final void triggerCollection(int why)
```

Triggers a collection, specifying one of the above reasons.

```
static final void triggerCollectionNow(int why)
```

Triggers a collection without allowing for a thread switch. This is needed for the Merlin lifetime analysis used by the GCTrace plan.

```
static final void triggerAsyncCollection()
```

There are situations where MMTk may decide that a GC is required, but for various reasons it is impossible to collect immediately. It sets a flag, and at a later stage an asynchronous collection is initiated via this method.

```
static final boolean noThreadsInGC()
```

Determine whether a collection cycle has fully completed (this is used to ensure a GC is not in the process of completing, to avoid, for example, an async GC being triggered on the switch from GC to mutator thread before all GC threads have switched.

```
private static final void checkForExhaustion(int why, boolean async)
```

Check for memory exhaustion, possibly throwing an out of memory exception and/or triggering another GC.

```
static boolean isNonParticipating(PlanLocal plan)
```

CPUs that are blocked in JNI calls cannot participate in GC, but their resources still need to be managed through the GC. This method tests for non-participating plans.

```
static void prepareNonParticipating(PlanLocal p)
```

This method allows the VM to perform its own preparation on a non-participating CPU.

```
static void prepareParticipating (PlanLocal p)
```

Set a collector thread's so that a gc will occur when it next executes.

```
static int rendezvous(int where)
```

Rendezvous with all other processors, returning the rank (that is, the order this processor arrived at the barrier).

```
static void scheduleFinalizerThread () {
```

Schedule the finalizerThread, if there are objects to be finalized and the finalizerThread is on its queue (ie. currently idle). Should be called at the end of GC after moveToFinalizable has been called, and before mutators are allowed to run.

A.5 Locking

Implemented in the class `Lock`. Simple, fair locks with deadlock detection.

```
Lock(String str)
```

Locks are given a name when they are constructed, which helps in debugging and/or error output.

```
void acquire()
```

Tries to acquire a lock and spin-waits until it is acquired.

```
void check (int w)
```

Sanity-check a lock—used for debugging.

```
void release()
```

Releases the lock by incrementing the serving counter.

A.6 Memory

This is how the collector interfaces with the underlying operating system to acquire and release raw memory.

```
static Address HEAP_START()
static Address HEAP_END()
static Address AVAILABLE_START()
static Address AVAILABLE_END()
```

These methods allow MMTk to inquire about the layout of virtual memory. `HEAP_START()` and `HEAP_END()` define the bounds of the virtual address region where MMTk will encounter objects. This can include areas that are managed by the VM. `AVAILABLE_START()` and `AVAILABLE_END()` define a subrange of the heap where MMTk is free to manage as it likes.

```
static ImmortalSpace getVMSpace()
```

Return the space associated with/reserved for the VM.

```
static void globalPrepareVMSpace()
static void localPrepareVMSpace()
static void localReleaseVMSpace()
static void globalReleaseVMSpace()
```

These methods allow the VM to do global and thread-local work before and after collection of the VM's space.

```
static void setHeapRange(int id, Address start, Address end)
```

Inform the VM the range of addresses associated with a portion of the heap.

```
static int mmap(Address start, int size)
static boolean mprotect(Address start, int size)
static boolean munprotect(Address start, int size)
```

These methods interact with the underlying OS to map and set protection on memory regions.

```
static void zero(Address start, Extent len)
static void zeroPages(Address start, int len)
```

Use the VM or underlying OS to zero memory regions. `zeroPages` is assumed to be more efficient for large regions of memory, and expects start and len to describe a region that starts and finishes on a page boundary.

```
static void dumpMemory(Address start, int beforeBytes,
                       int afterBytes)
```

Logs the contents of an address and the surrounding memory to the error output, for debugging purposes.

The following two methods provide support for processors with weak memory models.

```
static void sync()
static void isync()
```

`sync()` is a memory barrier for data writes. All writes performed before this point are guaranteed to be visible on all processors after this method completes.

`isync()` is a memory barrier for the instruction stream. All instructions currently being executed are allowed to run to completion, and all speculative fetches and execution is discarded. Code motion past an `isync()` is prohibited.

A.7 Object model

This interface class allows MMTk to access object metadata, and abstract over many of the fine details of object metadata implementation. Objects may (for instance) have headers embedded in the object, or in side data structures, and Object References could potentially be implemented as Handles. By using these operations to access object metadata, MMTk preserves implementation independance.

```
static ObjectReference copy(ObjectReference from, int allocator)
```

Copy an object using a plan's allocCopy to get space and install a forwarding pointer. On entry, `from` must have been reserved for copying by the caller (to prevent another GC thread from attempting to copy the same object). This method calls the plan's `getStatusForCopy()` method to establish a new status word for the copied object and `postCopy()` to allow the plan to perform any post copy actions.

Delegating this to the VM allows it to optimize the copying process for arrays, and also to react appropriately if objects vital to it are moved (for example, moving a chunk of code requires a memory barrier).

```
static int getSizeWhenCopied(ObjectReference object)
```

Return the size required to copy an object. Implementations of Java address-based hashing, for example, can require an extra header word when a hashed object is copied.

```
static int getCurrentSize(ObjectReference object)
```

Return the size used by an object, counting fields and metadata.

```
static ObjectReference getNextObject(ObjectReference object)
```

Return the next object in the heap under contiguous allocation. This allows algorithms such as a Cheney scan to 'walk the heap'.

```
static ObjectReference getObjectFromStartAddress(Address start)
```

Return an object reference from knowledge of the low order word. It is common for a pointer to an object to point to an actual field in the object, rather than the start of the header. This method allows the VM to convert between the two without requiring MMTk to know the details.

```
static Address objectStartRef(ObjectReference object)
```

Returns the lowest address of the storage associated with an object.

```
static Address refToAddress(ObjectReference object)
```

Returns an address guaranteed to be inside the storage assocatied with an object.

```
static byte [] getTypeDescriptor(ObjectReference ref)
```

Get the type descriptor for an object. A sequence of bytes we can print out in a debugging message to identify the type of an object.

```
static int getArrayLength(ObjectReference object)
```

Get the length of an array object, in elements.

```
static boolean isAcyclic(ObjectReference typeRef)
```

Checks if a reference of the given type in another object is inherently acyclic. This allows MMTk to optimize cycle detection in its reference counting collectors. Returning `false` for all objects is correct, but inefficient, however returning `true` for an object that could participate in a cycle could lead to uncollectable garbage.

A.7.1 Object layout

During a heap trace, MMTk makes a call to the VM for each object it encounters to get information about the object, such as where its pointer fields are located. This is communicated by passing an `MMType` object.

```
static MMType getObjectType(ObjectReference object)
```

An `MMType` object abstracts over the characteristics of an object that MMTk needs to know about in order to scan it. Generally, an `MMType` object will be created by the classloader when a class is loaded.

From the point of view of the VM, only the constructor is public.

```
final class MMType {
  MMType(boolean isDelegated, boolean isReferenceArray,
         boolean isAcyclic, int allocator, int [] offsets)
}
```

If `isDelegated` is true, then instead of using the `MMType` object to scan the object, the object will be passed to `Scanning.scanObject` for the VM to process. This allows for object layouts that are too complex to be efficiently encoded in an `MMType`.

A.7.2 GC metadata

Most GC algorithms require some amount of per-object metadata, for mark bits, 'being forwarded' bits, forwarding words. The current MMTk interface expects the VM to reserve several bits for MMTk metadata as the low-order bits of a metadata word. It also expects the forwarding pointer for copied objects to be written to the (higher order) bits of that word.

```
static boolean testAvailableBit(ObjectReference object, int idx)
static void setAvailableBit(ObjectReference object, int idx,
                            boolean flag)
```

Non-atomically test and set a bit available for memory manager use in an object.

```
static Word prepareAvailableBits(ObjectReference object)
static boolean attemptAvailableBits(ObjectReference object,
                                    Word oldVal, Word newVal)
```

This pair of methods implement atomic modification of the GC bits for an object. These two operations are commonly implemented as either load-locked/store conditional, load and compare-and-swap (CAS) or other architecture-specific mechanism.

```
static Word readAvailableBitsWord(ObjectReference object)
static void writeAvailableBitsWord(ObjectReference object,
                                   Word val)
```

Reads and writes to the word containing the bits available for memory manager metadata.

```
static Offset GC_HEADER_OFFSET()
```

A method in the `Constraints` object can request additional words be made available in an object header. This method gets the offset of the memory management header from the object reference address. The object model / memory manager interface will at some point be improved so that the memory manager does not need to know this.

A.8 Command-line options

MMTk provides a large set of parameters that can be used at run-time to control its operation, for troubleshooting or performance tuning purposes. The `Options` class allows MMTk to interface with VMs that may have their own idea of how certain aspects of GC are configured on the command line.

```
static String getKey(String name)
```

MMTk passes the VM its name for one of its parameters, and the VM returns the name that MMTk can expect to see on the command line. To just use MMTk's names, pass the same string back.

```
static void fail(Option o, String message)
```

Failure during option processing. This must never return.

```
static void warn(Option o, String message)
```

Warning during option processing.

A.9 Reference types

The ReferenceGlue class manages SoftReferences, WeakReferences, and PhantomReferences. When a `java/lang/ref/Reference` object is created, its address is added to a list of pending reference objects of the appropriate type.

An address is used so the reference will not stay alive during gc if it isn't in use elsewhere the mutator. During gc, the various lists are processed in the proper order to determine if any Reference objects are ready to be enqueued or whether referents that have died should be kept alive until the Reference is explicitly cleared. The `ReferenceProcessor` class drives this processing and uses this class to scan the lists of pending reference objects.

Elsewhere, there is a distinguished Finalizer thread. At the end of gc, if needed and if any Reference queue is not empty, the finalizer thread is scheduled to be run when gc is completed. This thread calls Reference.enqueue() to make the actual notifcation to the user program that the object state has changed.

This is currently somewhat Java specific, and will probably need generalizing for other languages.

```
static final int SOFT_SEMANTICS;
static final int WEAK_SEMANTICS;
static final int PHANTOM_SEMANTICS;
```

Identifiers for the semantics that the reference processor knows about, defined in the `ReferenceProcessor` class.

```
static final boolean REFERENCES_ARE_OBJECTS;
```

Whether references are implemented as heap objects (rather than in a table, for example).

```
static void scanReferences(int semantics, boolean nursery)
```

Scan through the set of references with the specified semantics. If 'nursery' is true, only reference objects identified since the last GC are scanned—a performance hint for generational collectors.

```
static final boolean enqueueReference(Address addr,
                                      boolean onlyOnce)
```

Put this Reference object on its ReferenceQueue (if it has one) when its referent is no longer sufficiently reachable. The definition of "reachable" is defined by the semantics of the particular subclass of Reference.

```
static ObjectReference getReferent(Address addr)
```

Get the referent from a reference. For Java the reference is a Reference object.

```
static void setReferent(Address addr, ObjectReference referent)
```

Set the referent in a reference. For Java the reference is a Reference object.

```
static int countWaitingReferences(int semantics)
```

Return the number of references being managed with a given semantics.

A.10 Scanning support

The key operations in tracing the heap are enumerating roots and scanning objects, and the interface for this is in the `Scanning` class.

```
static void preCopyGCInstances(TraceLocal trace)
```

Pre-copy all potentially movable instances used in the course of GC. This includes the thread objects representing the GC threads themselves. It is crucial that these instances are forwarded *prior* to the GC proper. Since these instances *are not* enqueued for scanning, it is important that when roots are computed the same instances are explicitly scanned and included in the set of roots. The existence of this method allows the actions of calculating roots and forwarding GC instances to be decoupled.

```
static void computeAllRoots(TraceLocal trace)
```

Computes all roots. This method establishes all roots for collection and places them in the root values, root locations and interior root locations queues. This method should not have side effects (such as copying or forwarding of objects). There are a number of important preconditions:

- All objects used in the course of GC (such as the GC thread objects) need to be 'pre-copied' prior to calling this method.
- The `threadCounter` must be reset so that load balancing parallel GC can share the work of scanning threads.

For each root identified, `computeAllRoots` calls either the `addRootLocation` or `addInteriorRootLocation` method of the `trace` object.

```
static void resetThreadCounter()
```

Prepares for using the `computeAllRoots` method. The thread counter allows multiple GC threads to co-operatively iterate through the thread data structure (if load balancing parallel GC threads were not important, the thread counter could simply be replaced by a for loop).

A.10.1 Delegated scanning

Scanning an object is a performance critical operation for a garbage collector, so as far as possible MMTk performs the operation in one of its own classes. In order to provide a uniform interface to disparate object models, the `MMType` class (Section A.7.1) is used to communicate the details of object layout between MMTk and the host VM.

Some objects in a virtual machine are sufficiently complex that encoding the description into a variant of an `MMType` object would be difficult, so MMTk allows the VM to request that object scanning be delegated back to it, and the following three methods provide the interface for this mechanism.

```
static void scanObject(TraceLocal trace,
                       ObjectReference object)
```

This method provides for delegated object scanning. When tracing an object, the scanObject method must call

```
trace.traceObjectLocation(Address slot)
```

on each pointer in the object.

```
static void precopyChildren(TraceLocal trace,
                            ObjectReference object)
```

This method provides for delegated pre-copying. When precopying an object's children, the precopyChildren method must call

```
trace.precopyObjectLocation(Address slot)
```

on each pointer in the object.

```
static void enumeratePointers(ObjectReference object,
                              Enumerator e)
```

Some collectors need to trace the heap (or subgraphs thereof) in several different ways, using subclasses of the abstract `Enumerator` type. One example is the recursive decrement operation of a reference counting collector. This method allows delegated enumeration, and must call

```
e.enumeratePointerLocation(Address slot)
```

on each pointer field in the object.

A.11 Statistics

MMTk provides detailed statistics on many aspects of its operation. The `Statistics` class provides the necessary VM support for this.

```
static final int getCollectionCount()
```

The number of collections that have occured.

```
static long cycles()
```

The current time, in a VM-specific unit.

```
static double cyclesToMillis(long c)
static double cyclesToSecs(long c)
static long millisToCycles(double t)
static long secsToCycles(double t)
```

These methods convert between the opaque 'cycles' and more meaningful units.

A.12 String handling

Certain parts of MMTk, such as message logging and options parsing, require string handling. For obvious reasons, MMTk can't use the standard Java library functions, so it requires the VM to implement these methods.

```
static int parseInt(String value)
static float parseFloat(String value)
```

Primitive parsing facilities.

```
static void write(char [] c, int len)
```

Debugging messages in MMTk are handled via the `Log` class, and this method is required to support it.

```
static void writeThreadId(char [] c, int len)
```

Logs a thread identifier and a message.

```
static int copyStringToChars(String src, char [] dst,
                             int dstBegin, int dstEnd)
```

Copies characters from the string into the character array. Thread switching is disabled during this method's execution.

A.13 GC Tracing support

These facilities are required to support the Merlin lifetime analysis used by MMTk's `GCTrace` plan. If you don't intend to use GCTrace, this class can be regarded as optional.

```
static final boolean gcEnabled()

static final Offset adjustSlotOffset(boolean isScalar,
                                     ObjectReference src,
                                     Address slot)

static final Address skipOwnFramesAndDump(ObjectReference typeRef)

static void updateDeathTime(Object obj)

static void setDeathTime(ObjectReference ref, Word time_)

static void setLink(ObjectReference ref, ObjectReference link)

static void updateTime(Word time_)

static Word getOID(ObjectReference ref)

static Word getDeathTime(ObjectReference ref)
```

```
static ObjectReference getLink(ObjectReference ref)

static Address getBootImageLink()

static Word getOID()

static void setOID(Word oid)

static final int getHeaderSize()

static final int getHeaderEndOffset()
```

A.14 Constants

The `VMConstants` class defines certain characteristics of the surrounding environment. These constants are defined as methods to prevent the bytecode compiler from doing any constant folding, but are expected to inline down to a compile-time constant by the time the VM is built.

```
static final byte LOG_BYTES_IN_ADDRESS()
static final byte LOG_BYTES_IN_WORD()
static final byte LOG_BYTES_IN_PAGE()
```

The log base two of the size of an address, a word and an OS page respectively.

```
static final byte LOG_MIN_ALIGNMENT()
```

The log base two of the minimum allocation alignment.

```
static final byte MAX_ALIGNMENT_SHIFT()
```

The log base two of `(MAX_ALIGNMENT/MIN_ALIGNMENT)`. Allows MMTk to calculate the maximum allowable alignment.

```
static final int MAX_BYTES_PADDING()
```

The maximum number of bytes of padding to prepend to an object.

Appendix B

Exported interface

While the services that MMTk requires from its host VM is well defined through the `org.mmtk.vm` package, the interface in the opposite direction is less clear. The majority of the VM's requirements should be satisfiable by the public methods and constants in the `Plan` class, although some of these are only designed to be called from within MMTk and most VMs will also require types and methods defined in the Utility package. The definition of this interface is still a work in progress.

Part of the difficulty in specifiying this interface is that MMTk provides a relatively general purpose set of functionality, out of which each VM is intended to build its own custom interface. The model we advocate is that the host VM define exactly what services it wants from MMTk and implement that requirementan interface package.

In order to illustrate both the model and the types of services that MMTk can rpovide the host VM, the JikesRVM interface is described here.

B.1 JikesRVM's MM Interface

JikesRVM uses a package containing several classes to provide its interface to JikesRVM.

B.1.1 MM Interface

As discussed in Chapter 3, MMTk communicates its requirements to the virtual machine through a `PlanConstraints` object. JikesRVM's MM interface converts these requirements into constants in the interface.

```
public static final boolean NEEDS_WRITE_BARRIER
public static final boolean NEEDS_PUTSTATIC_WRITE_BARRIER
public static final boolean NEEDS_TIB_STORE_WRITE_BARRIER
```

These three constants tell the VM whether MMTk requires write barriers, and if so, whether it requires certain specific types of write barrier.

JikesRVM enumerates the static fields as roots, and so never requires a `PUTSTATIC` write barrier.

```
public static final boolean MOVES_OBJECTS
```

This tells the VM whether it can expect objects to move, triggering for example tri-state hash value processing.

```
public static final boolean MOVES_TIBS
```

A TIB (Type Information Block) is what each object points to to obtain per-type information. MMTk doesn't know anything about TIBs per se, so this information is internal to JikesRVM.

```
public static final boolean GENERATE_GC_TRACE
```

Advises the VM whether GC Tracing is happening.

JikesRVM's initial boot image is created using Java reflection (see the JikesRVM home page and relevant publications for details), and so initialization occurs at build time, when the boot image is written, and again at run time. MMTk does not require any explicit initialization at build time, but the design takes into account that static initialization is done at this time.

```
public static final void init()
```

Initialization that occurs at *build* time. The value of statics as at the completion of this routine will be reflected in the boot image. Any objects referenced by those statics will be transitively included in the boot image.

```
public static final void boot(VM_BootRecord theBootRecord)
```

Initialization that occurs at *boot* time (runtime initialization). This is only executed by one processor (the primordial thread). Several of MMTk's classes require initialization at this time.

```
public static void postBoot()
```

Perform postBoot operations such as dealing with command line options (this is called as soon as options have been parsed, which is necessarily after the basic allocator boot). MMTk has a single post boot method (in `Plan`) which must be called at this time, and which performs initialization tasks that require knowledge of the command-line options.

```
public static void fullyBootedVM()
```

Notifies MMTk that the host VM is now fully booted.

```
public static void processCommandLineArg(String arg)
```

Tells MMTk about one of the gc-specific command line arguments.

If `REQUIRES_WRITE_BARRIER` is set, JikesRVM compiles bytecodes that write to the heap as calls to these methods, which in turn call the appropriate method on `PlanLocal`.

```
public static void putfieldWriteBarrier(Object ref, Offset offset,
      Object value,int locationMetadata)

public static void putstaticWriteBarrier(Offset offset, Object value)

public static void arrayStoreWriteBarrier(Object ref, int index,
                                          Object value)
public static boolean arrayCopyWriteBarrier(Object src,
  Offset srcOffset, Object tgt, Offset tgtOffset, int bytes)
```

The putField and arrayStore write barriers both call the single-field form of MMTk's write barrier (as would the putstatic barrier if it were used), with a context parameter to identify the operation that caused it.

```
public static void modifyCheck(Object object)
```

Checks that if a garbage collection is in progress then the given object is not movable. If it is movable error messages are logged and the system exits. This is used purely for sanity checking.

```
public static final int getCollectionCount()
```

Find out how many GCs have occurred.

```
public static final Extent freeMemory()
public static final Extent totalMemory()
public static final Extent maxMemory()
public static Extent getMaxHeapSize()
public static Extent getInitialHeapSize()
```

These methods allow the VM to access basic memory statistics.

```
public static final void gc()
```

External call to force a garbage collection.

```
public static void dumpRef(ObjectReference ref)
```

Logs information about the target of a reference to the error output.

```
public static boolean validRef(ObjectReference ref)
```

Checks if a reference is valid.

```
public static boolean addressInVM(Address address)
```

Checks if an address refers to an in-use area of memory.

```
public static boolean objectInVM(ObjectReference object)
```

Checks if a reference refers to an object in an in-use area of memory.

MMTk identifies the different regions it manages by an allocator id (an int), and provides a set of constants that the interface can use to choose among them.

```
public static int getDefaultAllocator()
```

Returns the default allocator.

```
public static int pickAllocator(VM_Type type, VM_Method method)
```

Chooses an appropriate allocator for a given method and object type. This method is called by the compiler(s) when a method is compiled, not when a method is executed.

```
public static Object allocateScalar(int size,
  Object [] tib, int allocator, int align, int offset)

public static Object allocateArray(int numElements, int logElementSize,
  int headerSize, Object [] tib, int allocator, int align, int offset)

public static Address allocateSpace(SelectedPlanLocal plan,
  int bytes, int align, int offset, int allocator,
  ObjectReference from)
```

These methods allocate scalars, arrays, and space for copied objects.

```
public static final Offset alignAllocation(
  Offset initialOffset, int align, int offset)
```

Align an allocation using some modulo arithmetic to guarantee the following property:

```
(region + offset) % alignment == 0
```

```
public static VM_CodeArray allocateCode(int numInstrs, boolean isHot) {
public static byte[] newStack(int bytes, boolean immortal)
public static Object[] newTIB (int n)
public static VM_CompiledMethod[] newContiguousCompiledMethodArray(int n
public static VM_DynamicLibrary[] newContiguousDynamicLibraryArray(int n
```

Various specialized allocation routines.

```
public static void addFinalizer(Object object)
public static Object getFinalizedObject()
public static void addSoftReference(SoftReference obj)
public static void addWeakReference(WeakReference obj)
public static void addPhantomReference(PhantomReference obj)
```

Finalizers and weak references.

```
public static void emergencyGrowHeap(int growSize)
```

Increase heap size for an emergency, such as processing an out of memory exception.

```
public static void notifyClassResolved(VM_Type vmType)
```

A new type has been resolved by the VM. Create a new MM type to reflect the VM type, and associate the MM type with the VM type.

```
public static void harnessBegin()
public static void harnessEnd()
```

Generic hooks to allow benchmarks to be harnessed. A plan may use this to perform certain actions prior to the commencement of a benchmark, such as a full heap collection, turning on instrumentation, etc.

```
public static boolean mightBeTIB(ObjectReference obj)
```

Check if object might be a TIB.

```
public static final boolean gcInProgress()
```

Returns true if GC is in progress.

```
public static void startGCspyServer()
```

Start the GCspy server