MMTk Tutorial

Steve Blackburn (Steve.Blackburn@anu.edu.au)
Perry Cheng (perryche@us.ibm.com)

Tutorial Expectations

- Audience
  - GC Researchers
  - VM implementors looking for a memory mangement system
- Takeaway
  - An understanding of what MMTk is
    - Flexible with high performance
    - GC research infrastructure allowing fair comparisons
  - How to build/extend a garbage collector in MMTk
- Format
  - Interactive
  - Keep in mind the varying levels of expertise in audience

Outline

- Part 0: A review of GC   (~10 minutes)
- Part 1: MMTk Overview (~10 minutes)
- Part 2: Structure of MMTk (~30 minutes)
- BREAK (15 minutes)
- Part 3: Demo: Writing a Collector (~1 hour)
- Q&A (15 minutes)

What is Garbage Collection (GC)?

- Automatic Memory Management
  - Minimal programmer interface
    - allocate
    - ~~deallocate~~
  - Optional application-level interface
    - Heap size
    - pause time
    - GC hints, …
  - Avoid error-prone manual memory management
    - Dangling pointers and resource leakage
    - But not all memory leaks

- Increasingly popular because of runtime safety
  - Java, C#, Perl, Python, LISP, ML, Haskell, …
  - Even C/C++ (smart pointers, conservative collectors)

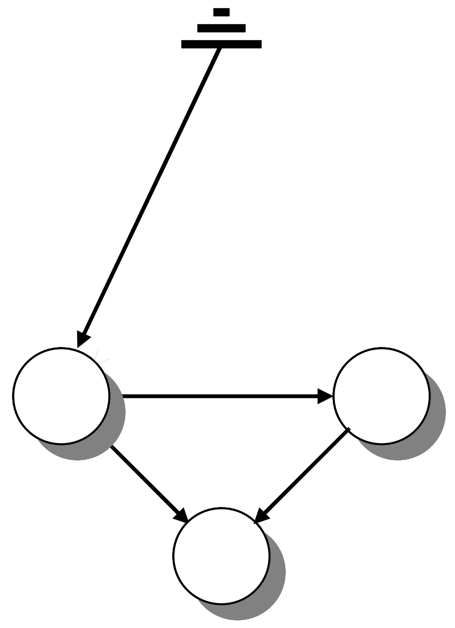
How does GC work?

- Approximate liveness by reachability
  - Liveness → Reachability
  - Unreachable → Dead (garbage)
  - Both deadness and unreachability are stable properties

- GC reclaims the space of unreachable objects

Identifying Garbage

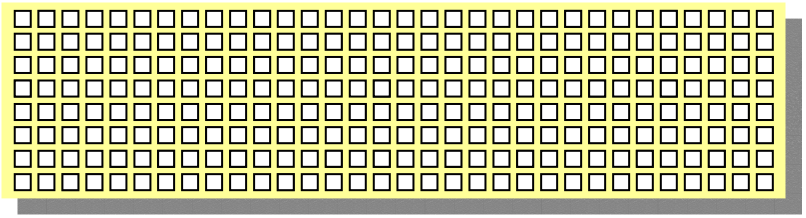
- Normal program execution
  - Allocate objects
  - Mutate edges
- GC triggers when space exhausted
  - Start at the “roots”
    - Registers (Locals)
    - Stacks (Locals)
    - Globals (Statics)
  - Compute transitive closure
  - Unreached objects are dead
- Program resumes



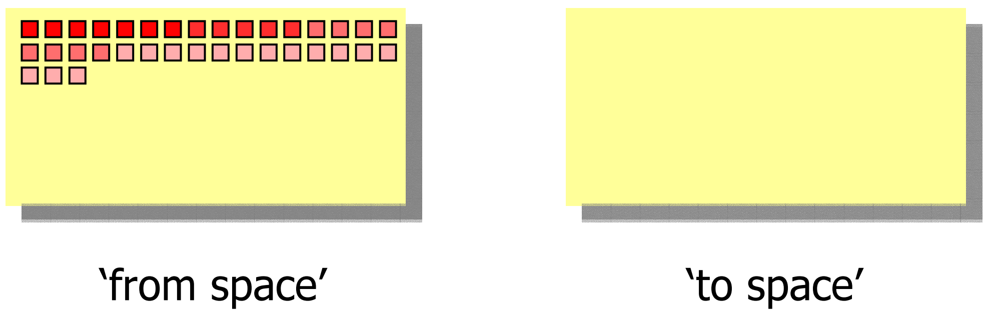
Space Management

- Two broad approaches:
  - Copying
    - Bump allocation & en masse reclamation
      - Fast allocation & reclaim
      - Space overhead, copy cost
  - Non-copying
    - Free-list allocation & reclamation
      - Space efficiency
      - Fragmentation

Non-copying GC



Copying Garbage Collection



Outline

- Part 0: A review of GC (~10 minutes)
- Part 1: MMTk Overview (~10 minutes)
- Part 2: Structure of MMTk (~30 minutes)
- BREAK (15 minutes)
- Part 3: Demo: Writing a Collector (~1 hour)
- Q&A (15 minutes)

Part 2: MMTk
(Memory Management Toolkit)

- Design Goals
  - –Composable
  - –Performance
  - –Portable
  - –Extensible
  - –Flexibility
- Authors: Steve Blackburn, Perry Cheng, Kathryn McKinley
- Support: David Grove

What is GC research about?

- Lower time overhead
- Lower/Better space usage
- (Predictably) Lower pause times
- Scalability / Distributed
- Better cache locality for application
- Better integration with host system

- Almost completely quantitative (performance)

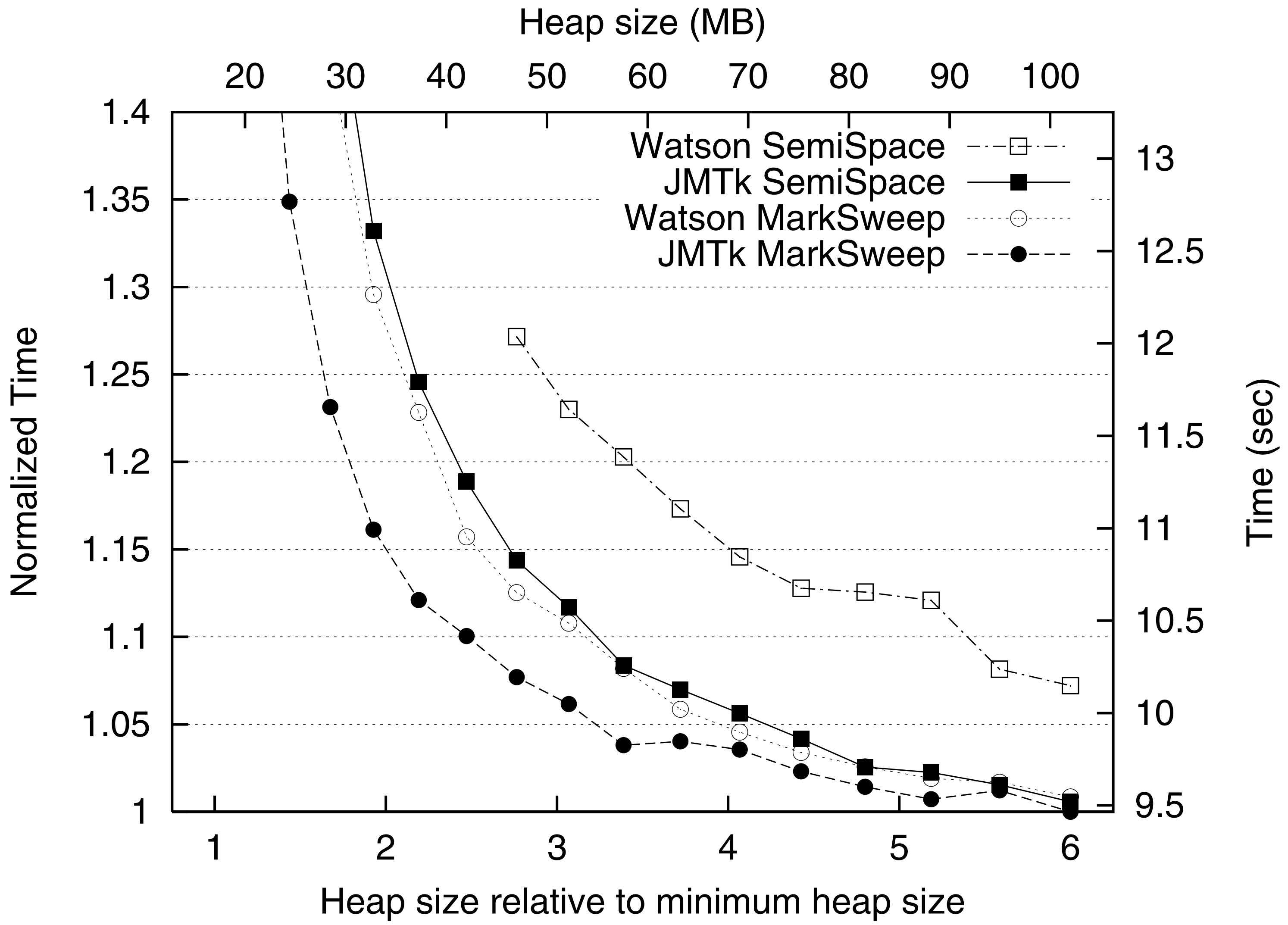
How many types of GC are there?

- Caveat: MMTk is an evolving system
- Many different (similar) GCs
  - Tracing, Copying, Ref-Count
  - Parallelism (SMPs)
  - Incremental/Concurrent/Real-time
  - Performance enhancing (Generational)
  - Portability
  - Specialized to environments (Conservative GC)

What makes a good GC research infrastructure?

- Credible VM
  - So that the GC does not seem absurdly good.
- Modular design
  - Ease of development
- Competitive Performance
  - Ensure reasonable GC code quality
- Uniform Code Quality
  - Quality check/Code Review
  - Allows fair algorithmic comparison

Competitive Performance



Development Context

- Jikes RVM
  - See http://www-124.ibm.com/developerworks/oss/jikesrvm/
  - Open source high-performance VM for Java
  - Adaptive JIT (runtime optimizing compiler)
  - Java-in-Java
  - Monolithic Collectors
- Pitfalls
  - Expressivity
  - Performance
  - Circularity

Avoiding the Pitfalls of Java as a Systems Language

- Expressivity
  - Add low-level types and unsafe operations
  - Machine addresses, direct load/stores
  - Annotate per-method level atomicity
- Performance
  - Use source-level pragmas to control inlining
    - Within collector
    - Collector into application
- Circularity
  - Who collects the collector’s garbage?
  - Avoid allocation
  - Use Immortal space (at least non-moving)
    - E.g. Collector’s stack and data structures
  - Employ pre-copying

Misc

- VMInterface and MMInterface for portability
- What else does MMTk run on?
  - Ongoing Rotor and Haskell interface
- Other functionality
  - GC Stats
  - GCSpy
  - Merlin

Outline

- Part 0: A review of GC (~10 minutes)
- Part 1: MMTk Overview (~10 minutes)
- Part 2: Structure of MMTk (~30 minutes)
- BREAK (15 minutes)
- Part 3: Demo: Writing a Collector (~1 hour)
- Q&A (15 minutes)

Compositionality: High-Level Structure

- Mechanisms
  - 55 collector-neutral, highly-tuned components
- Policies
  - 5 GC sub-components that describe how a region of memory is maintained
- Plans
  - 8 GC algorithms including all the canonical algorithms and some recent ones

Plans

- A plan is a composition of policies.
- Only one type of plan ever exists at runtime.
- One plan instance per kernel thread.
- That plan inherits from a specific plan to define a policy.

Policy, Space, Allocators

- An MMTk space is a region of memory (not necessarily contiguous) that is governed to the same policy and collected at the same time
- An MMTk policy is an allocation and collection strategy
- An MMTk allocator is an instance of an allocation mechanism
  - Typically, there are many concurrently active allocation points within a space.
  - Examples: Free list and bump pointer

Coding Issues

- Write clean code
- Write correct code
- Instrument your code with timers
- Identify performance issues
- Believe in your compiler writing colleagues
- Focus optimization efforts very carefully

Magic Types

- Magic types the fact that we have access to the compiler compiler to extend Java with new unboxed types (in other words, extend the existing primitive types)
- Magic types also implement magic operations, such as loading and storing to memory
- Magic types give us some degree of type safety (better than "int“ or "void*")
- Magic types allow us to abstract over VM implementation details
  - Width of word and address is abstracted over
  - Implementation of object references is abstracted over

Pragmas

- Like magic types, pragmas use the compiler to extend Java so that we can provide hints to the compiler.
  - Inline and NoInline pragmas make inlining requests (performance)
  - Uninterruptible allows us to be sure that a GC can never be triggered in some region (correctness)
  - Pragmas can be scoped w.r.t the whole class, a method, or (potentially) even a code block (by abusing the try-catch idom)

- When should I use pragmas?
  - Use uninteruptible whenever writing code that must not be interrupted by GC (or other threads).
  - Use inline sparingly (premature optimization is the root of all evil).
  - The opt compiler does a pretty good job of getting it right, generally.

Space Accounting

- MMTk accounts virtual and physical memory usage
  - virtual memory is consumed when spaces reserve regions of virtual memory
  - physical memory is tracked at a page granularity to reflect the number of pages of physical memory actually in use at any moment in time. This is done by a PageResouce associated with each Space

Local and Global Scope

- MMTk is designed to efficiently support concurrency
- The broad strategy is synchronized access to coarse grained global resources with unsynchronsized access to locally owned chunks of the global resource.
- "local" refers to fast unsychronized, per-thread activity
- "global" refers to heavy weight, coarse-grained global activity
- Each plan instance corresponds to one kernel thread

Fast Path vs. Slow Path

- We split performance critical activity into frequently executed, low overhead code (fastpath), and rarely executed code that may be somehwat more complex or heavyweight (slowpath)
- The fast path typically makes no checks, except whether the slow path should be taken.
- The slow path can make somewhat more complex checks and implement complex policy choices (because it is rarely executed, so the cost is heavily amortized)

Polling

- Poll is a key policy mechanism to determine whether a GC (or other triggers) is required.
- On certain slow path executions, the allocator will call poll and possibly trigger a GC
- The implementation of poll can be quite complex.
  - (E.g., see RefCount)
- Poll frequency is specified on a per-space basis
- Poll frequency is 128KB by default.

Why alloc() and postAlloc()?

- MMTk is responsible only for allocating raw (zeroed) space via alloc().
- Object initialization is performed by the VM.
- postAlloc() initializes GC metadata for the allocated object once it has been initialized.

Prepare and Commit

- Prepare and commit are major phases of each garbage collection, with the transitive closure of object tracing in between the two.
- During prepare() spaces (global) and allocators (local) are initialized for a pending collection. For example, semispaces might be flipped and the allocator readied for alloction into thenew to-space.
- During release() spaces (global) and allocators (local) are cleaned up following a collection. For example, semispaces are reclaimed and free lists might be reconstructed.

Object Tracing

- Object tracing refers to the transitive closure operation.
- The treatment for each object depends on the space in which it resides.
- Generally implemented by establishing the space and calling trace on the space
- Typically it involves scanning each object for references.

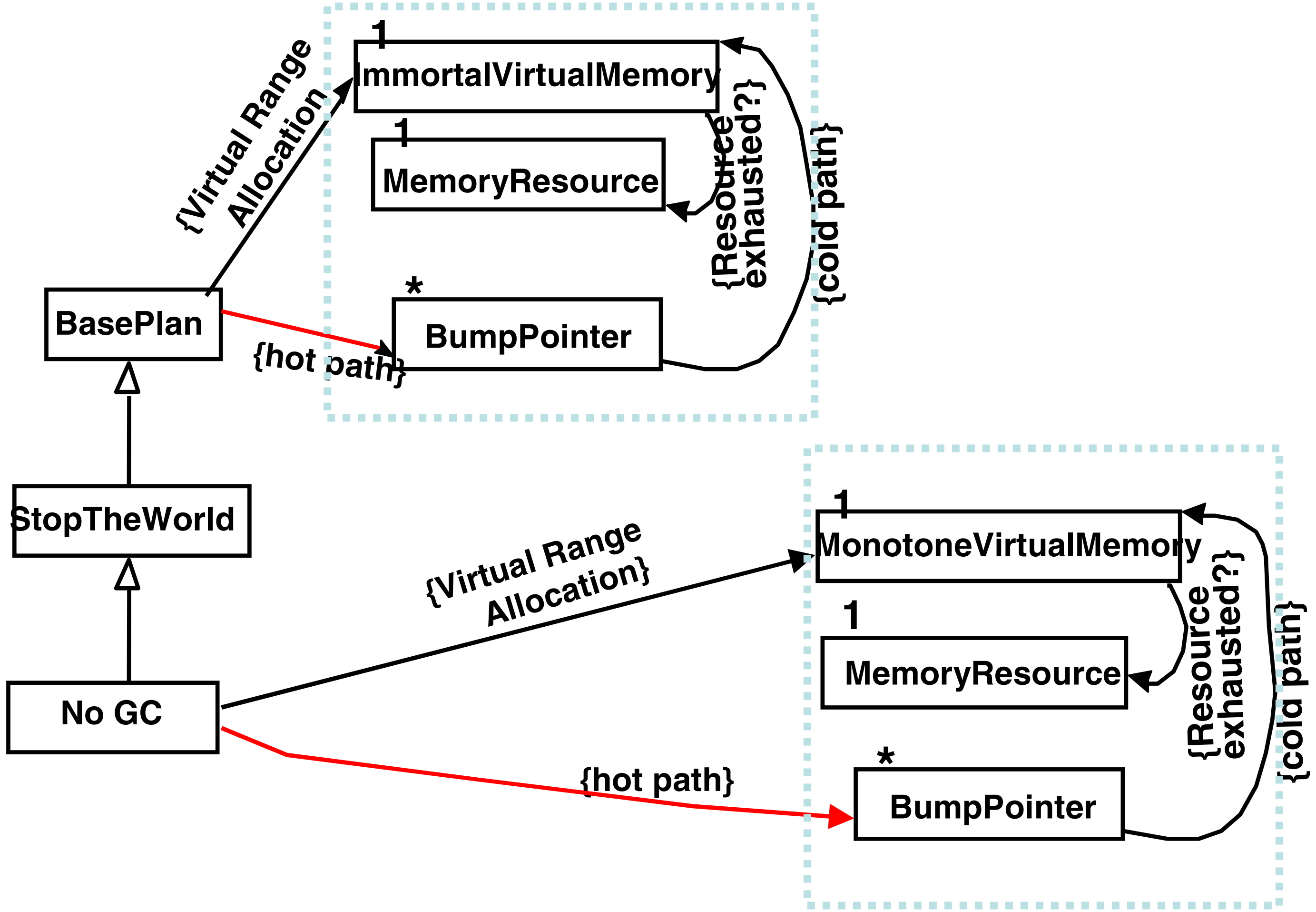
Special Built-in Spaces

- Boot-image
  - VM + JIT + GC + …
- Immortal (non-moving)
  - TIBs, stacks, and GC data structures
- Code Space (Coming soon)
  - Hot and Cold

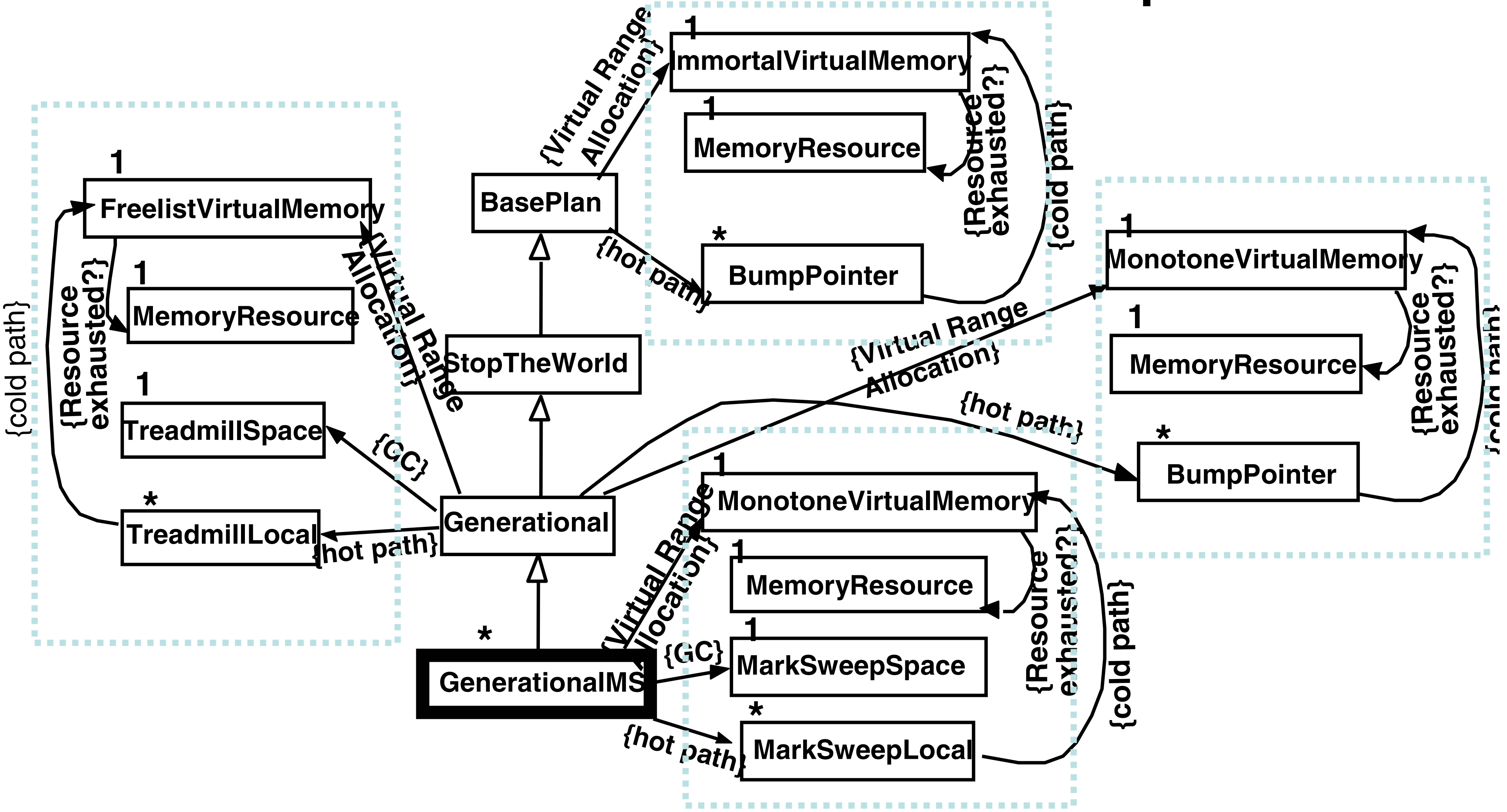
Outline

- Part 0: A review of GC (~10 minutes)
- Part 1: MMTk Overview (~10 minutes)
- Part 2: Structure of MMTk (~30 minutes)
- BREAK (15 minutes)
- Part 3: Demo: Writing a Collector (~1 hour)
- Q&A (15 minutes)

No GC Diagram



Generational Mark-Sweep



```
/*
 * (C) Copyright Department of Computer Science,
 * Australian National University. 2002
 */
package org.mmtk.plan;

import org.mmtk.policy.ImmortalSpace;
import org.mmtk.policy.Space;
import org.mmtk.utility.alloc.AllocAdvice;
import org.mmtk.utility.alloc.Allocator;
import org.mmtk.utility.alloc.BumpPointer;
import org.mmtk.utility.CallSite;
import org.mmtk.utility.heap.*;
import org.mmtk.utility.scan.MMType;
import org.mmtk.vm.Assert;
import org.mmtk.vm.Memory;
import org.mmtk.vm.ObjectModel;

import org.vmmagic.unboxed.*;
import org.vmmagic.pragma.*;

/**
 * This class implements a simple allocator without a collector.
 *
 * $Id: NoGC.java,v 1.5 2004/10/18 11:13:46 steveb-oss Exp $
 *
 * @author <a href="http://cs.anu.edu.au/~Steve.Blackburn">Steve Blackburn</a>
 * @version $Revision: 1.5 $
 * @date $Date: 2004/10/18 11:13:46 $
 */
public class NoGC extends StopTheWorldGC implements Uninterruptible {

  /****************************************************************************
   *
   * Class variables
   */
  public static final boolean MOVES_OBJECTS = false;
  public static final int GC_HEADER_BITS_REQUIRED = 0;
  public static final int GC_HEADER_BYTES_REQUIRED = 0;

  // Allocators
  public static final int ALLOCATORS = BASE_ALLOCATORS;

  // spaces
  private static ImmortalSpace defaultSpace = new ImmortalSpace("default", DEFAULT
_POLL_FREQUENCY, (float) 0.6);
  private static final int DS = defaultSpace.getDescriptor();

  /****************************************************************************
   *
   * Instance variables
   */

  // allocators
  private BumpPointer def;

  /****************************************************************************
   *
   * Initialization
   */

  /**
   * Class initializer.  This is executed <i>prior</i> to bootstrap
   * (i.e. at "build" time).  This is where key <i>global</i>
   * instances are allocated.  These instances will be incorporated
   * into the boot image by the build process.
   */
  static {}
```

```
  /**
   * Constructor
   */
  public NoGC() {
    def = new BumpPointer(defaultSpace);
  }

  /**
   * The boot method is called early in the boot process before any
   * allocation.
   */
  public static final void boot()
    throws InterruptiblePragma {
    StopTheWorldGC.boot();
  }


  /****************************************************************************
   *
   * Allocation
   */

  /**
   * Allocate space (for an object)
   *
   * @param bytes The size of the space to be allocated (in bytes)
   * @param align The requested alignment
   * @param offset The alignment offset
   * @param allocator The allocator number to be used for this allocation
   * @return The address of the first byte of the allocated region
   */
  public final Address alloc(int bytes, int align, int offset, int allocator)
    throws InlinePragma {
    switch (allocator) {
    case      ALLOC_LOS:  // no los, so use default allocator
    case  ALLOC_DEFAULT:  return def.alloc(bytes, align, offset);
    case ALLOC_IMMORTAL:  return immortal.alloc(bytes, align, offset);
    default:
      if (Assert.VERIFY_ASSERTIONS) Assert.fail("No such allocator");
      return Address.zero();
    }
  }

  /**
   * Perform post-allocation actions.  For many allocators none are
   * required.
   *
   * @param ref The newly allocated object
   * @param typeRef the type reference for the instance being created
   * @param bytes The size of the space to be allocated (in bytes)
   * @param allocator The allocator number to be used for this allocation
   */
  public final void postAlloc(ObjectReference ref, ObjectReference typeRef,
                              int bytes, int allocator) throws InlinePragma {
    switch (allocator) {
    case      ALLOC_LOS: // no los, so use default allocator
    case  ALLOC_DEFAULT: return;
    case ALLOC_IMMORTAL: return;
    default:
      if (Assert.VERIFY_ASSERTIONS) Assert.fail("No such allocator");
    }
  }

  /**
   * Allocate space for copying an object (this method <i>does not</i>
   * copy the object, it only allocates space)
   *
   * @param original A reference to the original object
   * @param bytes The size of the space to be allocated (in bytes)
   * @param align The requested alignment.
   * @param offset The alignment offset.
   * @return The address of the first byte of the allocated region
   */
  public final Address allocCopy(ObjectReference original, int bytes,
                                  int align, int offset)
    throws InlinePragma {
    Assert.fail("no allocCopy in noGC");
    //    return Address.zero();   // Trips some intel opt compiler bug...
    return Address.max();
  }

  /**
   * Perform any post-copy actions.  In this case nothing is required.
   *
   * @param ref The newly allocated object
   * @param typeRef The type reference for the instance being created
   * @param tib The TIB of the newly allocated object
   * @param bytes The size of the space to be allocated (in bytes)
   */
  public final void postCopy(ObjectReference ref, ObjectReference typeRef,
                             int bytes) {
    Assert.fail("no postCopy in noGC");
  }

  /**
   * Return the space into which an allocator is allocating.  This
   * particular method will match against those spaces defined at this
   * level of the class hierarchy.  Subclasses must deal with spaces
   * they define and refer to superclasses appropriately.  This exists
   * to support {@link BasePlan#getOwnAllocator(Allocator)}.
   *
   * @see BasePlan#getOwnAllocator(Allocator)
   * @param a An allocator
   * @return The space into which <code>a</code> is allocating, or
   * <code>null</code> if there is no space associated with
   * <code>a</code>.
   */
  protected final Space getSpaceFromAllocator(Allocator a) {
    if (a == def) return defaultSpace;
    return super.getSpaceFromAllocator(a);
  }

  /**
   * Return the allocator instance associated with a space
   * <code>space</code>, for this plan instance.  This exists
   * to support {@link BasePlan#getOwnAllocator(Allocator)}.
   *
   * @see BasePlan#getOwnAllocator(Allocator)
   * @param space The space for which the allocator instance is desired.
   * @return The allocator instance associated with this plan instance
   * which is allocating into <code>space</code>, or <code>null</code>
   * if no appropriate allocator can be established.
   */
  protected final Allocator getAllocatorFromSpace(Space space) {
    if (space == defaultSpace) return def;
    return super.getAllocatorFromSpace(space);
  }

  /**
   * Give the compiler/runtime statically generated alloction advice
   * which will be passed to the allocation routine at runtime.
   *
   * @param type The type id of the type being allocated
   * @param bytes The size (in bytes) required for this object
   * @param callsite Information identifying the point in the code
   * where this allocation is taking place.
   * @param hint A hint from the compiler as to which allocator this
   * site should use.
```

```
   * @return Allocation advice to be passed to the allocation routine
   * at runtime
   */
  public final AllocAdvice getAllocAdvice(MMType type, int bytes,
                                          CallSite callsite,
                                          AllocAdvice hint) {
    return null;
  }

  /**
   * Return the initial header value for a newly allocated LOS
   * instance.
   *
   * @param bytes The size of the newly created instance in bytes.
   * @return The inital header value for the new instance.
   */
  public static final Word getInitialHeaderValue(int bytes)
    throws InlinePragma {
    if (Assert.VERIFY_ASSERTIONS) Assert._assert(false);
    return Word.zero();
  }

  /**
   * This method is called periodically by the allocation subsystem
   * (by default, each time a page is consumed), and provides the
   * collector with an opportunity to collect.<p>
   *
   * We trigger a collection whenever an allocation request is made
   * that would take the number of pages in use (committed for use)
   * beyond the number of pages available.  Collections are triggered
   * through the runtime, and ultimately call the
   * <code>collect()</code> method of this class or its superclass.<p>
   *
   * This method is clearly interruptible since it can lead to a GC.
   * However, the caller is typically uninterruptible and this fiat allows
   * the interruptibility check to work.  The caveat is that the caller
   * of this method must code as though the method is interruptible.
   * In practice, this means that, after this call, processor-specific
   * values must be reloaded.
   *
   * @see org.mmtk.policy.Space#acquire(int)
   * @param mustCollect if <code>true</code> then a collection is
   * required and must be triggered.  Otherwise a collection is only
   * triggered if we deem it necessary.
   * @param space the space that triggered the polling (i.e. the space
   * into which an allocation is about to occur).
   * @return This method always returns false because this plan will
   * never trigger a GC.
   */
  public final boolean poll(boolean mustCollect, Space space)
    throws LogicallyUninterruptiblePragma {
    if (getPagesReserved() > getTotalPages()) Assert.error("Out of memory");
    return false;
  }

  /**
   * Perform operations with <i>global</i> scope in preparation for a
   * collection.  This is called by <code>StopTheWorld</code>, which will
   * ensure that <i>only one thread</i> executes this.<p>
   *
   * In this case, it means flipping semi-spaces, resetting the
   * semi-space memory resource, and preparing each of the collectors.
   */
  protected final void globalPrepare() {
    Assert.fail("\nGC Triggered in NoGC Plan.  Have you set -X:gc:ignoreSystemGC=true?");
  }

  /**
   * Perform operations with <i>thread-local</i> scope in preparation
   * for a collection.  This is called by <code>StopTheWorld</code>, which
   * will ensure that <i>all threads</i> execute this.<p>
   *
   * In this case, it means resetting the semi-space and large object
   * space allocators.
   */
  protected final void threadLocalPrepare(int count) {
    if (Assert.VERIFY_ASSERTIONS) Assert._assert(false);
  }

  /**
   * Perform operations with <i>thread-local</i> scope to clean up at
   * the end of a collection.  This is called by
   * <code>StopTheWorld</code>, which will ensure that <i>all threads</i>
   * execute this.<p>
   *
   * In this case, it means releasing the large object space (which
   * triggers the sweep phase of the mark-sweep collector used by the
   * LOS).
   */
  protected final void threadLocalRelease(int count) {
    if (Assert.VERIFY_ASSERTIONS) Assert._assert(false);
  }

  /**
   * Perform operations with <i>global</i> scope to clean up at the
   * end of a collection.  This is called by <code>StopTheWorld</code>,
   * which will ensure that <i>only one</i> thread executes this.<p>
   *
   * In this case, it means releasing each of the spaces and checking
   * whether the GC made progress.
   */
  protected final void globalRelease() {
    if (Assert.VERIFY_ASSERTIONS) Assert._assert(false);
  }


  /****************************************************************************
   *
   * Object processing and tracing
   */

  /**
   * Trace a reference during GC.  This involves determining which
   * collection policy applies and calling the appropriate
   * <code>trace</code> method.
   *
   * @param obj The object reference to be traced.  This is <i>NOT</i> an
   * interior pointer.
   * @return The possibly moved reference.
   */
  public static final ObjectReference traceObject(ObjectReference obj)
    throws InlinePragma {
    if (Assert.VERIFY_ASSERTIONS) Assert._assert(false);
    return obj;
  }

  /**
   * Trace a reference during GC.  This involves determining which
   * collection policy applies and calling the appropriate
   * <code>trace</code> method.
   *
   * @param obj The object reference to be traced.  This is <i>NOT</i>
   * an interior pointer.
   * @param root True if this reference to <code>obj</code> was held
   * in a root.
   * @return The possibly moved reference.
   */
  public static final ObjectReference traceObject(ObjectReference obj,
```

```
                                                    boolean root) {
    if (Assert.VERIFY_ASSERTIONS) Assert._assert(false);
    return ObjectReference.nullReference();
  }


  /**
   * Return true if <code>obj</code> is a live object.
   *
   * @param object The object in question
   * @return True if <code>obj</code> is a live object.
   */
  public static final boolean isLive(ObjectReference object) {
    if (object.isNull()) return false;
    return true;
  }


  public static boolean willNotMove(ObjectReference obj) {
    return true;
  }

  /****************************************************************************
   *
   * Space management
   */

  /**
   * Return the number of pages reserved for use given the pending
   * allocation.  This <i>includes</i> space reserved for copying.
   *
   * @return The number of pages reserved given the pending
   * allocation, including space reserved for copying.
   */
  protected static final int getPagesReserved() {
    int pages = defaultSpace.reservedPages();
    pages += immortalSpace.reservedPages();
    pages += metaDataSpace.reservedPages();
    return pages;
  }

  /**
   * Return the number of pages reserved for use given the pending
   * allocation.  This is <i>exclusive of</i> space reserved for
   * copying.
   *
   * @return The number of pages reserved given the pending
   * allocation, excluding space reserved for copying.
   */
  protected static final int getPagesUsed() {
    int pages = defaultSpace.reservedPages();
    pages += immortalSpace.reservedPages();
    pages += metaDataSpace.reservedPages();
    return pages;
  }

  /**
   * Return the number of pages available for allocation, <i>assuming
   * all future allocation is to the semi-space</i>.
   *
   * @return The number of pages available for allocation, <i>assuming
   * all future allocation is to the semi-space</i>.
   */
  protected static final int getPagesAvail() {
    return (getTotalPages() - defaultSpace.reservedPages()
            - immortalSpace.reservedPages());
  }
  /****************************************************************************
   *
   * Miscellaneous
   */

  /**
   * Show the status of each of the allocators.
   */
  public final void show() {
    def.show();
    immortal.show();
  }
}
```

```
/*
 * (C) Copyright Department of Computer Science,
 *     Australian National University. 2002
 */
package org.mmtk.policy;

import org.mmtk.utility.heap.*;
import org.mmtk.vm.Assert;
import org.mmtk.vm.Constants;
import org.mmtk.vm.ObjectModel;
import org.mmtk.vm.Plan;

import org.vmmagic.unboxed.*;
import org.vmmagic.pragma.*;

/**
 * This class implements tracing functionality for a simple copying
 * space.  Since no state needs to be held globally or locally, all
 * methods are static.
 *
 * $Id: CopySpace.java,v 1.20 2004/10/18 11:13:46 steveb-oss Exp $
 *
 * @author Perry Cheng
 * @author <a href="http://cs.anu.edu.au/~Steve.Blackburn">Steve Blackburn</a>
 * @author David Bacon
 * @author Steve Fink
 * @author Dave Grove
 *
 * @version $Revision: 1.20 $
 * @date $Date: 2004/10/18 11:13:46 $
 */
public final class CopySpace extends Space
  implements Constants, Uninterruptible {

  /****************************************************************************
   *
   * Class variables
   */
  public static final int LOCAL_GC_BITS_REQUIRED = 2;
  public static final int GLOBAL_GC_BITS_REQUIRED = 0;
  public static final int GC_HEADER_BYTES_REQUIRED = 0;

  private static final Word GC_MARK_BIT_MASK = Word.one();
  private static final Word GC_FORWARDED        = Word.one().lsh(1);  // ...10
  private static final Word GC_BEING_FORWARDED  = Word.one().lsh(2).sub(Word.one
());  // ...11
  private static final Word GC_FORWARDING_MASK  = GC_FORWARDED.or(GC_BEING_FORWA
RDED);

  /****************************************************************************
   *
   * Instance variables
   */
  private boolean fromSpace = true;

  /****************************************************************************
   *
   * Initialization
   */

  /**
   * The caller specifies the region of virtual memory to be used for
   * this space.  If this region conflicts with an existing space,
   * then the constructor will fail.
   *
   * @param name The name of this space (used when printing error messages etc)
   * @param pageBudget The number of pages this space may consume
   * before consulting the plan
   * @param start The start address of the space in virtual memory
```

```
   * @param bytes The size of the space in virtual memory, in bytes
   * @param fromSpace The does this instance start life as from-space
   * (or to-space)?
   */
  public CopySpace(String name, int pageBudget, Address start, Extent bytes,
                   boolean fromSpace) {
    super(name, true, false, start, bytes);
    this.fromSpace = fromSpace;
    pr = new MonotonePageResource(pageBudget, this, start, extent);
  }

  /**
   * Construct a space of a given number of megabytes in size.<p>
   *
   * The caller specifies the amount virtual memory to be used for
   * this space <i>in megabytes</i>.  If there is insufficient address
   * space, then the constructor will fail.
   *
   * @param name The name of this space (used when printing error messages etc)
   * @param pageBudget The number of pages this space may consume
   * before consulting the plan
   * @param mb The size of the space in virtual memory, in megabytes (MB)
   * @param fromSpace The does this instance start life as from-space
   * (or to-space)?
   */
  public CopySpace(String name, int pageBudget, int mb, boolean fromSpace) {
    super(name, true, false, mb);
    this.fromSpace = fromSpace;
    pr = new MonotonePageResource(pageBudget, this, start, extent);
  }

  /**
   * Construct a space that consumes a given fraction of the available
   * virtual memory.<p>
   *
   * The caller specifies the amount virtual memory to be used for
   * this space <i>as a fraction of the total available</i>.  If there
   * is insufficient address space, then the constructor will fail.
   *
   * @param name The name of this space (used when printing error messages etc)
   * @param pageBudget The number of pages this space may consume
   * before consulting the plan
   * @param frac The size of the space in virtual memory, as a
   * fraction of all available virtual memory
   * @param fromSpace The does this instance start life as from-space
   * (or to-space)?
   */
  public CopySpace(String name, int pageBudget, float frac,
                   boolean fromSpace) {
    super(name, true, false, frac);
    this.fromSpace = fromSpace;
    pr = new MonotonePageResource(pageBudget, this, start, extent);
  }

  /**
   * Construct a space that consumes a given number of megabytes of
   * virtual memory, at either the top or bottom of the available
   * virtual memory.
   *
   * The caller specifies the amount virtual memory to be used for
   * this space <i>in megabytes</i>, and whether it should be at the
   * top or bottom of the available virtual memory.  If the request
   * clashes with existing virtual memory allocations, then the
   * constructor will fail.
   *
   * @param name The name of this space (used when printing error messages etc)
   * @param pageBudget The number of pages this space may consume
   * before consulting the plan
   * @param mb The size of the space in virtual memory, in megabytes (MB)
```

```java
   * @param top Should this space be at the top (or bottom) of the
   * available virtual memory.
   * @param fromSpace The does this instance start life as from-space
   * (or to-space)?
   */
  public CopySpace(String name, int pageBudget, int mb, boolean top,
                   boolean fromSpace) {
    super(name, true, false, mb, top);
    this.fromSpace = fromSpace;
    pr = new MonotonePageResource(pageBudget, this, start, extent);
  }

  /**
   * Construct a space that consumes a given fraction of the available
   * virtual memory, at either the top or bottom of the available
   * virtual memory.
   *
   * The caller specifies the amount virtual memory to be used for
   * this space <i>as a fraction of the total available</i>, and
   * whether it should be at the top or bottom of the available
   * virtual memory.  If the request clashes with existing virtual
   * memory allocations, then the constructor will fail.
   *
   * @param name The name of this space (used when printing error messages etc)
   * @param pageBudget The number of pages this space may consume
   * before consulting the plan
   * @param frac The size of the space in virtual memory, as a
   * fraction of all available virtual memory
   * @param top Should this space be at the top (or bottom) of the
   * available virtual memory.
   * @param fromSpace The does this instance start life as from-space
   * (or to-space)?
   */
  public CopySpace(String name, int pageBudget, float frac, boolean top,
                   boolean fromSpace) {
    super(name, true, false, frac, top);
    this.fromSpace = fromSpace;
    pr = new MonotonePageResource(pageBudget, this, start, extent);
  }

  public void prepare(boolean fromSpace) { this.fromSpace = fromSpace; }
  public void release() { ((MonotonePageResource) pr).reset(); }

  /**
   * Release an allocated page or pages.  In this case we do nothing
   * because we only release pages enmasse.
   *
   * @param start The address of the start of the page or pages
   */
  public final void release(Address start) throws InlinePragma {
    Assert._assert(false);  // this policy only releases pages enmasse
  }

  /**
   * Trace an object under a copying collection policy.
   * If the object is already copied, the copy is returned.
   * Otherwise, a copy is created and returned.
   * In either case, the object will be marked on return.
   *
   * @param object The object to be traced.
   * @return The forwarded object.
   */
  public final ObjectReference traceObject(ObjectReference object)
    throws InlinePragma {
    if (fromSpace)
    return forwardObject(object, true);
    else
      return object;
  }
```

```java
  /**
   * Mark an object as having been traversed.
   *
   * @param object The object to be marked
   * @param markState The sense of the mark bit (flips from 0 to 1)
   */
  public static void markObject(ObjectReference object, Word markState)
    throws InlinePragma {
    if (testAndMark(object, markState)) Plan.enqueue(object);
  }

  /**
   * Forward an object.
   *
   * @param object The object to be forwarded.
   * @return The forwarded object.
   */
  public static ObjectReference forwardObject(ObjectReference object)
    throws InlinePragma {
    return forwardObject(object, false);
  }

  /**
   * Forward an object and enqueue it for scanning
   *
   * @param object The object to be forwarded.
   * @return The forwarded object.
   */
  public static ObjectReference forwardAndScanObject(ObjectReference object)
    throws InlinePragma {
    return forwardObject(object, true);
  }

  /**
   * Forward an object.  If the object has not already been forwarded,
   * then conditionally enqueue it for scanning.
   *
   * @param object The object to be forwarded.
   * @param scan If <code>true</code>, then enqueue the object for
   * scanning if the object was previously unforwarded.
   * @return The forwarded object.
   */
  private static ObjectReference forwardObject(ObjectReference object,
                                               boolean scan)
    throws InlinePragma {
    Word forwardingPtr = attemptToForward(object);

    // Somebody else got to it first.
    //
    if (stateIsForwardedOrBeingForwarded(forwardingPtr)) {
      while (stateIsBeingForwarded(forwardingPtr))
        forwardingPtr = getForwardingWord(object);
      ObjectReference newObject = forwardingPtr.and(GC_FORWARDING_MASK.not()).to
Address().toObjectReference();
      return newObject;
    }

    // We are the designated copier
    //
    ObjectReference newObject = ObjectModel.copy(object);
    setForwardingPointer(object, newObject);
    if (scan) {
      Plan.enqueue(newObject);       // Scan it later
    } else {
      Plan.enqueueForwardedUnscannedObject(newObject);
    }
    return newObject;
  }
  public final boolean isLive(ObjectReference object) {
    return isForwarded(object);
  }


  /****************************************************************************
   *
   * Header manipulation
   */

  /**
   * Clear the GC portion of the header for an object.
   *
   * @param object the object ref to the storage to be initialized
   */
  public static void clearGCBits(ObjectReference object) throws InlinePragma {
    Word header = ObjectModel.readAvailableBitsWord(object);
    ObjectModel.writeAvailableBitsWord(object, header.and(GC_FORWARDING_MASK.not
()));
  }

  /**
   * Has an object been forwarded?
   *
   * @param object The object to be checked
   * @return True if the object has been forwarded
   */
  public static boolean isForwarded(ObjectReference object)
    throws InlinePragma {
    return stateIsForwarded(getForwardingWord(object));
  }

  /**
   * Has an object been forwarded or being forwarded?
   *
   * @param object The object to be checked
   * @return True if the object has been forwarded or is being forwarded
   */
  public static boolean isForwardedOrBeingForwarded(ObjectReference object)
    throws InlinePragma {
    return stateIsForwardedOrBeingForwarded(getForwardingWord(object));
  }

  /**
   * Non-atomic read of forwarding pointer word
   *
   * @param object The object whose forwarding word is to be read
   * @return The forwarding word stored in <code>object</code>'s
   * header.
   */
  private static Word getForwardingWord(ObjectReference object)
    throws InlinePragma {
    return ObjectModel.readAvailableBitsWord(object);
  }

 /**
   * Non-atomic read of forwarding pointer
   *
   * @param object The object whose forwarding pointer is to be read
   * @return The forwarding pointer stored in <code>object</code>'s
   * header.
   */
  public static ObjectReference getForwardingPointer(ObjectReference object)
    throws InlinePragma {
    return getForwardingWord(object).and(GC_FORWARDING_MASK.not()).toAddress().t
oObjectReference();
  }
```

```
  /**
   * Used to mark boot image objects during a parallel scan of objects
   * during GC Returns true if marking was done.
   *
   * @param object The object to be marked
   * @param value The value to store in the mark bit
   */
  private static boolean testAndMark(ObjectReference object, Word value)
    throws InlinePragma {
    Word oldValue;
    do {
      oldValue = ObjectModel.prepareAvailableBits(object);
      Word markBit = oldValue.and(GC_MARK_BIT_MASK);
      if (markBit.EQ(value)) return false;
    } while (!ObjectModel.attemptAvailableBits(object, oldValue,
                                                oldValue.xor(GC_MARK_BIT_MASK)))
;
    return true;
  }

  /**
   * Either return the forwarding pointer if the object is already
   * forwarded (or being forwarded) or write the bit pattern that
   * indicates that the object is being forwarded
   *
   * @param object The object to be forwarded
   * @return The forwarding pointer for the object if it has already
   * been forwarded.
   */
  private static Word attemptToForward(ObjectReference object)
    throws InlinePragma {
    Word oldValue;
    do {
      oldValue = ObjectModel.prepareAvailableBits(object);
      if (oldValue.and(GC_FORWARDING_MASK).EQ(GC_FORWARDED)) return oldValue;
    } while (!ObjectModel.attemptAvailableBits(object, oldValue,
                                                oldValue.or(GC_BEING_FORWARDED))
);
    return oldValue;
  }

  /**
   * Is the state of the forwarding word being forwarded?
   *
   * @param fword A forwarding word.
   * @return True if the forwarding word's state is being forwarded.
   */
  private static boolean stateIsBeingForwarded(Word fword)
    throws InlinePragma {
    return  fword.and(GC_FORWARDING_MASK).EQ(GC_BEING_FORWARDED);
  }

  /**
   * Is the state of the forwarding word forwarded?
   *
   * @param fword A forwarding word.
   * @return True if the forwarding word's state is forwarded.
   */
  private static boolean stateIsForwarded(Word fword)
    throws InlinePragma {
    return  fword.and(GC_FORWARDING_MASK).EQ(GC_FORWARDED);
  }

  /**
   * Is the state of the forwarding word forwarded or being forwarded?
   *
   * @param fword A forwarding word.
   * @return True if the forwarding word's state is forwarded or being
   * forwarded.
   */
  public static boolean stateIsForwardedOrBeingForwarded(Word fword)
    throws InlinePragma {
    return !(fword.and(GC_FORWARDED).isZero());
  }

  /**
   * Non-atomic write of forwarding pointer word (assumption, thread
   * doing the set has done attempt to forward and owns the right to
   * copy the object)
   *
   * @param object The object whose forwarding pointer is to be set
   * @param ptr The forwarding pointer to be stored in the object's
   * forwarding word
   */
  private static void setForwardingPointer(ObjectReference object,
                                           ObjectReference ptr)
    throws InlinePragma {
    ObjectModel.writeAvailableBitsWord(object, ptr.toAddress().toWord().or(GC_FO
RWARDED));
  }
}
```

```
/*
 * (C) Copyright Department of Computer Science,
 * Australian National University. 2002
 */

package org.mmtk.utility.alloc;

import org.mmtk.policy.Space;
import org.mmtk.utility.*;
import org.mmtk.utility.heap.*;
import org.mmtk.vm.Constants;

import org.vmmagic.unboxed.*;
import org.vmmagic.pragma.*;

import org.mmtk.vm.gcspy.AbstractDriver;

/**
 * This class implements a simple bump pointer allocator.  The
 * allocator operates in <code>BLOCK</code> sized units.  Intra-block
 * allocation is fast, requiring only a load, addition comparison and
 * store.  If a block boundary is encountered the allocator will
 * request more memory (virtual and actual).
 *
 * @author <a href="http://cs.anu.edu.au/~Steve.Blackburn">Steve Blackburn</a>
 * @version $Revision: 1.25 $
 * @date $Date: 2004/10/06 11:24:42 $
 */
public final class BumpPointer extends Allocator
  implements Constants, Uninterruptible {
  public final static String Id = "$Id: BumpPointer.java,v 1.25 2004/10/06 11:24:42 steveb–oss Exp
$";

  /**
   * Constructor
   *
   * @param vmr The virtual memory resource from which this bump
   * pointer will acquire virtual memory.
   * @param mr The memory resource from which this bump pointer will
   * acquire memory.
   */
  public BumpPointer(Space space) {
    this.space = space;
    reset();
  }

  public void reset () {
    cursor = Address.zero();
    limit = Address.zero();
  }

  /**
   * Re-associate this bump pointer with a different virtual memory
   * resource.  Reset the bump pointer so that it will use this virtual
   * memory resource on the next call to <code>alloc</code>.
   *
   * @param vmr The virtual memory resouce with which this bump
   * pointer is to be associated.
   */
  public void rebind(Space space) {
    reset();
    this.space = space;
  }

  /**
   * Allocate space for a new object.  This is frequently executed code and
   * the coding is deliberaetly sensitive to the optimizing compiler.
   * After changing this, always check the IR/MC that is generated.
   *
```

```
   * @param bytes The number of bytes allocated
   * @param align The requested alignment
   * @param offset The offset from the alignment
   * @return The address of the first byte of the allocated region
   */
  final public Address alloc(int bytes, int align, int offset)
    throws InlinePragma {
    Address oldCursor = alignAllocation(cursor, align, offset);
    Address newCursor = oldCursor.add(bytes);
      if (newCursor.GT(limit))
      return allocSlow(bytes, align, offset);
    cursor = newCursor;
    //    Log.write("a["); Log.write(oldCursor); Log.writeln("]");
    return oldCursor;
  }

  final protected Address allocSlowOnce(int bytes, int align, int offset,
                                        boolean inGC) {
    Extent chunkSize = Word.fromIntZeroExtend(bytes).add(CHUNK_MASK).and(CHUNK_M
ASK.not()).toExtent();
    Address start;
    start = space.acquire(Conversions.bytesToPages(chunkSize));
    if (start.isZero())
      return start;

    // check for (dis)contiguity with previous chunk
    if (limit.NE(start)) cursor = start;
    limit = start.add(chunkSize);
    return alloc(bytes, align, offset);
  }

  public void show() {
    Log.write("cursor = "); Log.write(cursor);
    Log.write(" limit = "); Log.writeln(limit);
  }

  /**
   * Gather data for GCSpy
   * @param event The GCSpy event
   * @param driver the GCSpy driver for this space
   */
  public void gcspyGatherData(int event, AbstractDriver driver) {
    //    vmResource.gcspyGatherData(event, driver);
  }

  /****************************************************************************
   *
   * Instance variables
   */
  private Address cursor;
  private Address limit;
  private Space space;

  /****************************************************************************
   *
   * Final class variables (aka constants)
   *
   * Must ensure the bump pointer will go through slow path on (first)
   * alloc of initial value
   */
  private static final int LOG_CHUNK_SIZE = LOG_BYTES_IN_PAGE + 3;
  private static final Word CHUNK_MASK = Word.one().lsh(LOG_CHUNK_SIZE).sub(Word
.one());
}
```

```
/*
 * (C) Copyright Department of Computer Science,
 * Australian National University. 2002
 */
package org.mmtk.plan;

import org.mmtk.policy.ImmortalSpace;
import org.mmtk.policy.LargeObjectSpace;
import org.mmtk.policy.LargeObjectLocal;
import org.mmtk.policy.RawPageSpace;
import org.mmtk.policy.Space;
import org.mmtk.utility.alloc.Allocator;
import org.mmtk.utility.alloc.BumpPointer;
import org.mmtk.utility.Conversions;
import org.mmtk.utility.heap.*;
import org.mmtk.utility.Log;
import org.mmtk.utility.Options;
import org.mmtk.utility.deque.*;
import org.mmtk.utility.statistics.*;
import org.mmtk.utility.TraceGenerator;
import org.mmtk.vm.Assert;
import org.mmtk.vm.Collection;
import org.mmtk.vm.Constants;
import org.mmtk.vm.Memory;
import org.mmtk.vm.ObjectModel;
import org.mmtk.vm.Plan;

import org.vmmagic.pragma.*;
import org.vmmagic.unboxed.*;

/**
 * This abstract class implments the core functionality for all memory
 * management schemes.  All JMTk plans should inherit from this
 * class.<p>
 *
 * All plans make a clear distinction between <i>global</i> and
 * <i>thread-local</i> activities.  Global activities must be
 * synchronized, whereas no synchronization is required for
 * thread-local activities.  Instances of Plan map 1:1 to "kernel
 * threads" (aka CPUs or in Jikes RVM, VM_Processors).  Thus instance
 * methods allow fast, unsychronized access to Plan utilities such as
 * allocation and collection.  Each instance rests on static resources
 * (such as memory and virtual memory resources) which are "global"
 * and therefore "static" members of Plan.  This mapping of threads to
 * instances is crucial to understanding the correctness and
 * performance proprties of this plan.
 *
 * @author Perry Cheng
 * @author <a href="http://cs.anu.edu.au/~Steve.Blackburn">Steve Blackburn</a>
 * @version $Revision: 1.103 $
 * @date $Date: 2004/10/18 11:13:45 $
 */
public abstract class BasePlan
  implements Constants, Uninterruptible {
  public final static String Id = "$Id: BasePlan.java,v 1.103 2004/10/18 11:13:45 steveb-oss Exp $
";

  /****************************************************************************
   *
   * Class variables
   */
  public static final boolean NEEDS_WRITE_BARRIER = false;
  public static final boolean NEEDS_PUTSTATIC_WRITE_BARRIER = false;
  public static final boolean NEEDS_TIB_STORE_WRITE_BARRIER = false;
  public static final boolean SUPPORTS_PARALLEL_GC = true;
  public static final boolean MOVES_TIBS = false;
  public static final boolean STEAL_NURSERY_GC_HEADER = false;
  public static final boolean GENERATE_GC_TRACE = false;
```

```
  private static final int MAX_PLANS = 100;
  protected static Plan [] plans = new Plan[MAX_PLANS];
  protected static int planCount = 0;        // Number of plan instances in exis
tence

  // GC state and control variables
  public static final int NOT_IN_GC = 0;   // this must be zero for C code
  public static final int GC_PREPARE = 1;  // before setup and obtaining root
  public static final int GC_PROPER = 2;
  protected static boolean initialized = false;
  protected static boolean awaitingCollection = false;
  protected static int collectionsInitiated = 0;
  private static int gcStatus = NOT_IN_GC; // shared variable
  protected static int exceptionReserve = 0;
  public static final int DEFAULT_POLL_FREQUENCY = (128<<10)>>LOG_BYTES_IN_PAGE;

  // Spaces
  protected static final int IMMORTAL_MB = 32;
  protected static final int META_DATA_MB = 32;
  protected static final float LOS_FRAC = (float) 0.1;
  protected static Space vmSpace = Memory.getVMSpace();
  protected static ImmortalSpace immortalSpace = new ImmortalSpace("immortal", DE
FAULT_POLL_FREQUENCY, META_DATA_MB);
  protected static final int IMMORTAL = immortalSpace.getDescriptor();
  protected static RawPageSpace metaDataSpace = new RawPageSpace("meta", DEFAULT
_POLL_FREQUENCY, META_DATA_MB);
  protected static final int META = metaDataSpace.getDescriptor();
  protected static LargeObjectSpace loSpace = new LargeObjectSpace("los", DEFAULT
_POLL_FREQUENCY, LOS_FRAC);
  public static final int LOS = loSpace.getDescriptor();

  // Allocators
  public static final int ALLOC_DEFAULT = 0;
  public static final int ALLOC_IMMORTAL = 1;
  public static final int ALLOC_LOS = 2;
  public static final int ALLOC_GCSPY = 3;
  public static final int ALLOC_HOT_CODE = ALLOC_DEFAULT;
  public static final int ALLOC_COLD_CODE = ALLOC_DEFAULT;
  public static final int BASE_ALLOCATORS = 4;

  // Statistics
  protected static boolean insideHarness = false;
  public static Timer totalTime;
  public static SizeCounter mark;
  public static SizeCounter cons;

  // Miscellaneous constants
  protected static final int META_DATA_POLL_FREQUENCY = DEFAULT_POLL_FREQUENCY;
  protected static final int LOS_SIZE_THRESHOLD = 8 * 1024;
  public    static final int NON_PARTICIPANT = 0;
  protected static final boolean GATHER_WRITE_BARRIER_STATS = false;

  public static final int DEFAULT_MIN_NURSERY = (256*1024)>>LOG_BYTES_IN_PAGE;
  public static final int DEFAULT_MAX_NURSERY = MAX_INT;


  /****************************************************************************
   *
   * Instance variables
   */
  private int id = 0;                     // Zero-based id of plan instance
  public BumpPointer immortal;
  protected LargeObjectLocal los;
  Log log;

  /****************************************************************************
   *
   * Initialization
   */
  /**
   * Class initializer.  This is executed <i>prior</i> to bootstrap
   * (i.e. at "build" time).  This is where key <i>global</i>
   * instances are allocated.  These instances will be incorporated
   * into the boot image by the build process.
   */
  static {
    totalTime = new Timer("time");
    if (Stats.GATHER_MARK_CONS_STATS) {
      mark = new SizeCounter("mark", true, true);
      cons = new SizeCounter("cons", true, true);
    }

  }

  /**
   * Constructor
   */
  BasePlan() {
    id = planCount++;
    plans[id] = (Plan) this;
    immortal = new BumpPointer(immortalSpace);
    los = new LargeObjectLocal(loSpace);
    log = new Log();
  }

  /**
   * The boot method is called early in the boot process before any
   * allocation.
   */
  public static void boot() throws InterruptiblePragma {
    if (Plan.GENERATE_GC_TRACE) TraceGenerator.boot(Memory.HEAP_START());
  }

  /**
   * The boot method is called by the runtime immediately after
   * command-line arguments are available.  Note that allocation must
   * be supported prior to this point because the runtime
   * infrastructure may require allocation in order to parse the
   * command line arguments.  For this reason all plans should operate
   * gracefully on the default minimum heap size until the point that
   * boot is called.
   */
  public static void postBoot() {
    if (Options.verbose > 2) Space.printVMMap();
    if (Options.verbose > 0) Stats.startAll();
  }

  public static void fullyBooted() {
    initialized = true;
    exceptionReserve = (int) (getTotalPages() * (1 - Collection.OUT_OF_MEMORY_TH
RESHOLD));
  }

  /****************************************************************************
   *
   * Allocation
   */

  /**
   * Run-time check of the allocator to use for a given allocation
   *
   * At the moment this method assumes that allocators will use the simple
   * (worst) method of aligning to determine if the object is a large object
   * to ensure that no objects are larger than other allocators can handle.
   *
   * @param bytes The number of bytes to be allocated
   * @param align The requested alignment.
```

```
   * @param allocator The allocator statically assigned to this allocation
   * @return The allocator dyncamically assigned to this allocation
   */
  public static int checkAllocator(int bytes, int align, int allocator)
    throws InlinePragma {
    if (allocator == ALLOC_DEFAULT &&
        Allocator.getMaximumAlignedSize(bytes, align) > LOS_SIZE_THRESHOLD)
      return ALLOC_LOS;
    else
      return allocator;
  }

  /**
   * Given an allocator, <code>a</code>, determine the space into
   * which <code>a</code> is allocating and then return an allocator
   * (possibly <code>a</code>) associated with <i>this plan
   * instance</i> which is allocating into the same space as
   * <code>a</code>.<p>
   *
   * The need for the method is subtle.  The problem arises because
   * application threads may change their affinity with
   * processors/posix threads, and this may happen during a GC (at the
   * point at which the scheduler performs thread switching associated
   * with the GC). At the end of a GC, the thread that triggered the
   * GC may now be bound to a different processor and thus the
   * allocator instance on its stack may be no longer be valid
   * (i.e. it may pertain to a different plan instance).<p>
   *
   * This method allows the correct allocator instance to be
   * established and associated with the thread (see {@link
   * org.mmtk.utility.alloc.Allocator#allocSlowBody(int, int, int,
   * boolean) Allocator.allocSlowBody()}).
   *
   * @see org.mmtk.utility.alloc.Allocator
   * @see org.mmtk.utility.alloc.Allocator#allocSlowBody(int, int,
   * int, boolean)
   * @param a An allocator instance.
   * @return An allocator instance associated with <i>this plan
   * instance</i> that allocates into the same space as <code>a</code>
   * (this may in fact be <code>a</code>).
   */
  public final Allocator getOwnAllocator(Allocator a) {
    Space space = getSpaceFromAllocatorAnyPlan(a);
    if (space == null)
      Assert.fail("BasePlan.getOwnAllocator could not obtain space");
    return getAllocatorFromSpace(space);
  }

  /**
   * Return the name of the space into which an allocator is
   * allocating.  The allocator, <code>a</code> may be assocaited with
   * any plan instance.
   *
   * @param a An allocator
   * @return The name of the space into which <code>a</code> is
   * allocating, or "<null>" if there is no space associated with
   * <code>a</code>.
   */
  public static String getSpaceNameFromAllocatorAnyPlan(Allocator a) {
    Space space = getSpaceFromAllocatorAnyPlan(a);
    if (space == null)
      return "<null>";
    else
      return space.getName();
  }

  /**
   * Return the space into which an allocator is allocating.  The
   * allocator, <code>a</code> may be assocaited with any plan
   * instance.
   *
   * @param a An allocator
   * @return The space into which <code>a</code> is allocating, or
   * <code>null</code> if there is no space associated with
   * <code>a</code>.
   */
  private static Space getSpaceFromAllocatorAnyPlan(Allocator a) {
    for (int i=0; i<plans.length; i++) {
      Space space = plans[i].getSpaceFromAllocator(a);
      if (space != null)
        return space;
    }
    return null;
  }

  /**
   * Return the space into which an allocator is allocating.  This
   * particular method will match against those spaces defined at this
   * level of the class hierarchy.  Subclasses must deal with spaces
   * they define and refer to superclasses appropriately.
   *
   * @param a An allocator
   * @return The space into which <code>a</code> is allocating, or
   * <code>null</code> if there is no space associated with
   * <code>a</code>.
   */
  protected Space getSpaceFromAllocator(Allocator a) {
    if (a == immortal) return immortalSpace;
    else if (a == los) return loSpace;
    return null;
  }

  /**
   * Return the allocator instance associated with a space
   * <code>space</code>, for this plan instance.
   *
   * @param space The space for which the allocator instance is desired.
   * @return The allocator instance associated with this plan instance
   * which is allocating into <code>space</code>, or <code>null</code>
   * if no appropriate allocator can be established.
   */
  protected Allocator getAllocatorFromSpace(Space space) {
    if (space == immortalSpace) return immortal;
    else if (space == loSpace) return los;
    else if (space == metaDataSpace)
      Assert.fail("BasePlan.getAllocatorFromSpace given meta space");
    else if (space != null)
      Assert.fail("BasePlan.getAllocatorFromSpace given invalid space");
    else
      Assert.fail("BasePlan.getAllocatorFromSpace given null space");
    return null;
  }

  /**
   * Perform any required initialization of the GC portion of the header.
   * Called for objects created at boot time.
   *
   * @param ref the object ref to the storage to be initialized
   * @param typeRef the type reference for the instance being created
   * @param size the number of bytes allocated by the GC system for
   * this object.
   * @param status the initial value of the status word
   * @return The new value of the status word
   */
  public static Word getBootTimeAvailableBits(int ref, ObjectReference typeRef,
                                              int size, Word status)
    throws InlinePragma {
    return status; // nothing to do (no bytes of GC header)
```

```
  }

 /****************************************************************************
   *
   * Object processing and tracing
   */

  /**
   * Add a gray object
   *
   * @param object The object to be enqueued
   */
  public static final void enqueue(ObjectReference object)
    throws InlinePragma {
    Plan.getInstance().values.push(object);
  }

  /**
   * Return true if the object is either forwarded or being forwarded
   *
   * @param object
   * @return True if the object is either forwarded or being forwarded
   */
  public static boolean isForwardedOrBeingForwarded(ObjectReference object)
    throws InlinePragma {
    return false;
  }

  /**
   * Add an unscanned, forwarded object for subseqent processing.
   * This mechanism is necessary for "pre-copying".
   *
   * @param object The object to be enqueued
   */
  public static final void enqueueForwardedUnscannedObject(ObjectReference objec
t)
    throws InlinePragma {
    Plan.getInstance().forwardedObjects.push(object);
  }

  /**
   * Trace a reference during GC.  This involves determining which
   * collection policy applies and calling the appropriate
   * <code>trace</code> method.
   *
   * @param objLoc The location containing the object reference to be
   * traced.  The object reference is <i>NOT</i> an interior pointer.
   * @param root True if <code>objLoc</code> is within a root.
   */
  public static final void traceObjectLocation(Address objLoc, boolean root)
    throws InlinePragma {
    ObjectReference object = objLoc.loadObjectReference();
    ObjectReference newObject = Plan.traceObject(object, root);
    objLoc.store(newObject);
  }

  /**
   * Trace a reference during GC.  This involves determining which
   * collection policy applies and calling the appropriate
   * <code>trace</code> method.  This reference is presumed <i>not</i>
   * to be from a root.
   *
   * @param objLoc The location containing the object reference to be
   * traced.  The object reference is <i>NOT</i> an interior pointer.
   */
  public static final void traceObjectLocation(Address objLoc)
    throws InlinePragma {
    traceObjectLocation(objLoc, false);
  }
  /**
   * Trace a reference during GC.  This involves determining which
   * collection policy applies and calling the appropriate
   * <code>trace</code> method.
   *
   * @param object The object reference to be traced.
   * @param interiorRef The interior reference inside obj that must be traced.
   * @param root True if the reference to <code>obj</code> was held in a root.
   * @return The possibly moved interior reference.
   */
  public static final Address traceInteriorReference(ObjectReference object,
                                                     Address interiorRef,
                                                     boolean root) {
    Offset offset = interiorRef.diff(object.toAddress());
    ObjectReference newObject = Plan.traceObject(object, root);
    if (Assert.VERIFY_ASSERTIONS) {
      if (offset.sLT(Offset.zero()) || offset.sGT(Offset.fromIntSignExtend(1<<24
))) {  // There is probably no object this large
        Log.writeln("ERROR: Suspiciously large delta of interior pointer from object base");
        Log.write("    object base = "); Log.writeln(object);
        Log.write("    interior reference = "); Log.writeln(interiorRef);
        Log.write("    delta = "); Log.writeln(offset);
        Assert._assert(false);
      }
    }
    return newObject.toAddress().add(offset);
  }

  /**
   * A pointer location has been enumerated by ScanObject.  This is
   * the callback method, allowing the plan to perform an action with
   * respect to that location.  By default nothing is done.
   *
   * @param location An address known to contain a pointer.  The
   * location is within the object being scanned by ScanObject.
   */
  public void enumeratePointerLocation(Address location) {}

  /**
   * Return true if an object is known to be immovable.  This method
   * should be refined by subclasses.  At this level we simply make a
   * conservative check whether the object resides in a space that is
   * declared to be immovable.
   *
   * @param object The object whose movability is being tested
   * @return True if the object resides in a space that is known to be
   * immovable.
   */
  public static boolean willNotMove(ObjectReference object) {
    return !Space.isMovable(object);
  }

  /**
   * Forward the object referred to by a given address and update the
   * address if necessary.  This <i>does not</i> enqueue the referent
   * for processing; the referent must be explicitly enqueued if it is
   * to be processed.<p>
   *
   * <i>Non-copying collectors do nothing, copying collectors must
   * override this method.</i>
   *
   * @param location The location whose referent is to be forwarded if
   * necessary.  The location will be updated if the referent is
   * forwarded.
   */
  public static void forwardObjectLocation(Address location) {
    if (Assert.VERIFY_ASSERTIONS) Assert._assert(!Plan.MOVES_OBJECTS);
  }
```

```
  /**
   * If the object in question has been forwarded, return its
   * forwarded value.<p>
   *
   * <i>Non-copying collectors do nothing, copying collectors must
   * override this method.</i>
   *
   * @param object The object which may have been forwarded.
   * @return The forwarded value for <code>object</code>.  <i>In this
   * case return <code>object</code>, copying collectors must override
   * this method.
   */
  public static ObjectReference getForwardedReference(ObjectReference object) {
    if (Assert.VERIFY_ASSERTIONS) Assert._assert(!Plan.MOVES_OBJECTS);
    return object;
  }

  /**
   * Make alive an object that was not otherwise known to be alive.
   * This is used by the ReferenceProcessor, for example.
   *
   * @param object The object which is to be made alive.
   */
  public static void makeAlive(ObjectReference object) {
    Plan.traceObject(object);
  }

  /**
   * An object is unreachable and is about to be added to the
   * finalizable queue.  The collector must ensure the object is not
   * collected (despite being otherwise unreachable), and should
   * return its forwarded address if keeping the object alive involves
   * forwarding.<p>
   *
   * <i>For many collectors these semantics relfect those of
   * <code>traceObject</code>, which is implemented here.  Other
   * collectors must override this method.</i>
   *
   * @param object The object which may have been forwarded.
   * @return The forwarded value for <code>object</code>.  <i>In this
   * case return <code>object</code>, copying collectors must override
   * this method.
   */
  public static ObjectReference retainFinalizable(ObjectReference object) {
    return Plan.traceObject(object);
  }

  /**
   * Return true if an object is ready to move to the finalizable
   * queue, i.e. it has no regular references to it.  This method may
   * (and in some cases is) be overridden by subclasses.
   *
   * @param object The object being queried.
   * @return <code>true</code> if the object has no regular references
   * to it.
   */
  public static boolean isFinalizable(ObjectReference object) {
    return !Plan.isLive(object);
  }

  /****************************************************************************
   *
   * Read and write barriers.  By default do nothing, override if
   * appropriate.
   */

  /**
   * A new reference is about to be created. Take appropriate write
   * barrier actions.<p>
   *
   * <b>By default do nothing, override if appropriate.</b>
   *
   * @param src The object into which the new reference will be stored
   * @param slot The address into which the new reference will be
   * stored.
   * @param tgt The target of the new reference
   * @param metaDataA An int that assists the host VM in creating a store
   * @param metaDataB An int that assists the host VM in creating a store
   * @param mode The context in which the store occured
   */
  public void writeBarrier(ObjectReference src, Address slot,
                           ObjectReference tgt, int metaDataA, int metaDataB,
                           int mode) {
    // Either: write barriers are used and this is overridden, or
    //         write barriers are not used and this is never called
    if (Assert.VERIFY_ASSERTIONS) Assert._assert(false);
  }

  /**
   * A number of references are about to be copied from object
   * <code>src</code> to object <code>dst</code> (as in an array
   * copy).  Thus, <code>dst</code> is the mutated object.  Take
   * appropriate write barrier actions.<p>
   *
   * @param src The source of the values to be copied
   * @param srcOffset The offset of the first source address, in
   * bytes, relative to <code>src</code> (in principle, this could be
   * negative).
   * @param dst The mutated object, i.e. the destination of the copy.
   * @param dstOffset The offset of the first destination address, in
   * bytes relative to <code>tgt</code> (in principle, this could be
   * negative).
   * @param bytes The size of the region being copied, in bytes.
   * @return True if the update was performed by the barrier, false if
   * left to the caller (always false in this case).
   */
  public boolean writeBarrier(ObjectReference src, int srcOffset,
                              ObjectReference dst, int dstOffset,
                              int bytes) {
    // Either: write barriers are used and this is overridden, or
    //         write barriers are not used and this is never called
    if (Assert.VERIFY_ASSERTIONS) Assert._assert(false);
    return false;
  }

  /**
   * Read a reference. Take appropriate read barrier action, and
   * return the value that was read.<p> This is a <b>substituting<b>
   * barrier.  The call to this barrier takes the place of a load.<p>
   *
   * @param src The object being read.
   * @param src The address being read.
   * @param context The context in which the read arose (getfield, for example)
   * @return The reference that was read.
   */
  public final Address readBarrier(ObjectReference src, Address slot,
                                   int context)
    throws InlinePragma {
    // read barrier currently unimplemented
    if (Assert.VERIFY_ASSERTIONS) Assert._assert(false);
    return Address.max();
  }

  /****************************************************************************
   *
   * GC trace generation support methods
   */
```

```
  /**
   * Return true if <code>obj</code> is in a space known to the class and
   * is reachable.
   *
   * <i> For this method to be accurate, collectors must override this method
   * to define results for the spaces they create.</i>
   *
   * @param object The object in question
   * @return True if <code>obj</code> is a reachable object in a space known by
   *         the class; unreachable objects may still be live, however.  False
   *         will be returned if it cannot be determined if the object is
   *         reachable (e.g., resides in a space unknown to the class).
   */
  public boolean isReachable(ObjectReference object) {
    if (object.isNull()) return false;
    if (Space.isImmortal(object)) {
      return ImmortalSpace.isReachable(object);
    }
      if (Assert.VERIFY_ASSERTIONS)
        Assert.fail("BasePlan.isReachable given object from unknown space");
      return false;
    }

  /**
   * Follow a reference during GC.  This involves determining which
   * collection policy applies and getting the final location of the object
   *
   * <i> For this method to be accurate, collectors must override this method
   * to define results for the spaces they create.</i>
   *
   * @param object The object reference to be followed.  This is
   * <i>NOT</i> an interior pointer.
   * @return The possibly moved reference.
   */
  public static ObjectReference followObject(ObjectReference object) {
    if (Assert.VERIFY_ASSERTIONS) Assert._assert(!Plan.MOVES_OBJECTS);
    return ObjectReference.nullReference();
  }

  /****************************************************************************
   *
   * Space management
   */

  /**
   * Return the amount of <i>free memory</i>, in bytes (where free is
   * defined as not in use).  Note that this may overstate the amount
   * of <i>available memory</i>, which must account for unused memory
   * that is held in reserve for copying, and therefore unavailable
   * for allocation.
   *
   * @return The amount of <i>free memory</i>, in bytes (where free is
   * defined as not in use).
   */
  public static Extent freeMemory() throws UninterruptiblePragma {
    return totalMemory().sub(usedMemory());
  }

  /**
   * Return the amount of <i>memory in use</i>, in bytes.  Note that
   * this excludes unused memory that is held in reserve for copying,
   * and therefore unavailable for allocation.
   *
   * @return The amount of <i>memory in use</i>, in bytes.
   */
  public static Extent usedMemory() throws UninterruptiblePragma {
    return Conversions.pagesToBytes(Plan.getPagesUsed());
  }
```

```java
  /**
   * Return the amount of <i>memory in use</i>, in bytes.  Note that
   * this includes unused memory that is held in reserve for copying,
   * and therefore unavailable for allocation.
   *
   * @return The amount of <i>memory in use</i>, in bytes.
   */
  public static Extent reservedMemory() throws UninterruptiblePragma {
    return Conversions.pagesToBytes(Plan.getPagesReserved());
  }

  /**
   * Return the total amount of memory managed to the memory
   * management system, in bytes.
   *
   * @return The total amount of memory managed to the memory
   * management system, in bytes.
   */
  public static Extent totalMemory() throws UninterruptiblePragma {
    return HeapGrowthManager.getCurrentHeapSize();
  }

  /**
   * Return the total amount of memory managed to the memory
   * management system, in pages.
   *
   * @return The total amount of memory managed to the memory
   * management system, in pages.
   */
  public static int getTotalPages() throws UninterruptiblePragma {
    return totalMemory().toWord().rshl(LOG_BYTES_IN_PAGE).toInt();
  }

  /**
   * @return Whether last GC is a full GC.
   */
  public static boolean isLastGCFull () {
    return true;
  }

  /****************************************************************************
   *
   * Collection
   */

  /**
   * Check whether an asynchronous collection is pending.<p>
   *
   * This is decoupled from the poll() mechanism because the
   * triggering of asynchronous collections can trigger write
   * barriers, which can trigger an asynchronous collection.  Thus, if
   * the triggering were tightly coupled with the request to alloc()
   * within the write buffer code, then inifinite regress could
   * result.  There is no race condition in the following code since
   * there is no harm in triggering the collection more than once,
   * thus it is unsynchronized.
   */
  public static void checkForAsyncCollection() {
    if (awaitingCollection && Collection.noThreadsInGC()) {
      awaitingCollection = false;
      Collection.triggerAsyncCollection();
    }
  }

  /**
   * A collection has been initiated.  Increment the collectionInitiated
   * state variable appropriately.
```

```java
   */
  public static void collectionInitiated() throws UninterruptiblePragma {
    collectionsInitiated++;
  }

  /**
   * A collection has fully completed.  Decrement the collectionInitiated
   * state variable appropriately.
   */
  public static void collectionComplete() throws UninterruptiblePragma {
    if (Assert.VERIFY_ASSERTIONS) Assert._assert(collectionsInitiated > 0);
    // FIXME The following will probably break async GC.  A better fix
    // is needed
    collectionsInitiated = 0;
  }

  /**
   * Return true if a collection is in progress.
   *
   * @return True if a collection is in progress.
   */
  public static boolean gcInProgress() {
    return gcStatus != NOT_IN_GC;
  }

  /**
   * Return true if a collection is in progress and past the preparatory stage.
   *
   * @return True if a collection is in progress and past the preparatory stage.
   */
  public static boolean gcInProgressProper () {
    return gcStatus == GC_PROPER;
  }

  /**
   * Return true if a collection is in progress.
   *
   * @return True if a collection is in progress.
   */
  protected static void setGcStatus (int s) {
    Memory.isync();
    gcStatus = s;
    Memory.sync();
  }

  /**
   * A user-triggered GC has been initiated.  By default, do nothing,
   * but this may be overridden.
   */
  public static void userTriggeredGC() throws UninterruptiblePragma {
  }

  /****************************************************************************
   *
   * Miscellaneous
   */

  /**
   * Generic hook 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.  By default do nothing.  Subclasses may
   * override.
   */
  public static void harnessBegin() throws InterruptiblePragma {
    Options.fullHeapSystemGC = true;
    System.gc();
    Options.fullHeapSystemGC = false;
    insideHarness = true;
    Stats.startAll();
  }

  /**
   * Generic hook to allow benchmarks to be harnessed.  A plan may use
   * this to perform certain actions after the completion of a
   * benchmark, such as a full heap collection, turning off
   * instrumentation, etc.  By default do nothing.  Subclasses may
   * override.
   */
  public static void harnessEnd() {
    Stats.stopAll();
    Stats.printStats();
    insideHarness = false;
  }

  /**
   * Return the GC count (the count is incremented at the start of
   * each GC).
   *
   * @return The GC count (the count is incremented at the start of
   * each GC).
   */
  public static int gcCount() {
    return Stats.gcCount();
  }

  /**
   * Return the <code>RawPageAllocator</code> being used.
   *
   * @return The <code>RawPageAllocator</code> being used.
   */
  public static RawPageSpace getMetaDataRPA() {
    return metaDataSpace;
  }

  /**
   * The VM is about to exit.  Perform any clean up operations.
   *
   * @param value The exit value
   */
  public void notifyExit(int value) {
    if (Options.verbose == 1) {
      Log.write("[End ");
      totalTime.printTotalSecs();
      Log.writeln(" s]");
    } else if (Options.verbose == 2) {
      Log.write("[End ");
      totalTime.printTotalMillis();
      Log.writeln(" ms]");
    }
    if (Options.verboseTiming) printDetailedTiming(true);
    planExit(value);
    if (Plan.GENERATE_GC_TRACE)
      TraceGenerator.notifyExit(value);
  }

  protected void printDetailedTiming(boolean totals) {}

  /**
   * The VM is about to exit.  Perform any plan-specific clean up
   * operations.
   *
   * @param value The exit value
   */
  protected void planExit(int value) {}

  /**
```

```
   * Specify if the plan has been fully initialized
   *
   * @return True if the plan has been initialized
   */
  public static boolean initialized() {
    return initialized;
  }

  /****************************************************************************
   *
   * Miscellaneous
   */

  /**
   * Return the <code>Log</code> instance for this plan.
   *
   * @return the <code>Log</code> instance
   */
  public Log getLog() {
    return log;
  }

  /**
   * Start the GCSpy server
   *
   * @param wait Whether to wait
   * @param port The port to talk to the GCSpy client (e.g. visualiser)
   */
  protected static void startGCSpyServer(int port, boolean wait) {}

  /**
   * Prepare GCSpy for a collection
   * Order of operations is guaranteed by StopTheWorld plan
   *    1. globalPrepare()
   *    2. threadLocalPrepare()
   *    3. gcspyPrepare()
   *    4. gcspyPreRelease()
   *    5. threadLocalRelease()
   *    6. gcspyRelease()
   *    7. globalRelease()
   *
   * Typically, zero gcspy's buffers
   */
  protected void gcspyPrepare() {}

  /**
   * Deal with root locations
   *
   */
  protected void gcspyRoots(AddressDeque rootLocations, AddressPairDeque interio
rRootLocations) {}

  /**
   * Before thread-local release
   *
   */
  protected void gcspyPreRelease() {}

  /**
   * After thread-local release
   *
   */
  protected void gcspyPostRelease() {}
}
```

```
/*
 * (C) Copyright Department of Computer Science,
 * Australian National University. 2002
 */
package org.mmtk.plan;

import org.mmtk.policy.RawPageSpace;
import org.mmtk.policy.Space;
import org.mmtk.utility.Conversions;
import org.mmtk.utility.heap.*;
import org.mmtk.utility.Finalizer;
import org.mmtk.utility.Log;
import org.mmtk.utility.Options;
import org.mmtk.utility.deque.*;
import org.mmtk.utility.ReferenceProcessor;
import org.mmtk.utility.scan.Scan;
import org.mmtk.utility.statistics.*;
import org.mmtk.vm.Assert;
import org.mmtk.vm.Constants;
import org.mmtk.vm.Plan;
import org.mmtk.vm.Scanning;
import org.mmtk.vm.Statistics;
import org.mmtk.vm.Collection;

import org.vmmagic.unboxed.*;
import org.vmmagic.pragma.*;

/**
 * This abstract class implments the core functionality for
 * stop-the-world collectors.  Stop-the-world collectors should
 * inherit from this class.<p>
 *
 * All plans make a clear distinction between <i>global</i> and
 * <i>thread-local</i> activities.  Global activities must be
 * synchronized, whereas no synchronization is required for
 * thread-local activities.  Instances of Plan map 1:1 to "kernel
 * threads" (aka CPUs or in Jikes RVM, VM_Processors).  Thus instance
 * methods allow fast, unsychronized access to Plan utilities such as
 * allocation and collection.  Each instance rests on static resources
 * (such as memory and virtual memory resources) which are "global"
 * and therefore "static" members of Plan.  This mapping of threads to
 * instances is crucial to understanding the correctness and
 * performance proprties of this plan.
 *
 * @author Perry Cheng
 * @author <a href="http://cs.anu.edu.au/~Steve.Blackburn">Steve Blackburn</a>
 * @version $Revision: 1.69 $
 * @date $Date: 2004/10/18 11:13:46 $
 */
public abstract class StopTheWorldGC extends BasePlan
  implements Constants, Uninterruptible {
  public final static String Id = "$Id: StopTheWorldGC.java,v 1.69 2004/10/18 11:13:46 steveb-os
s Exp $";

  /****************************************************************************
   *
   * Class variables
   */
  // Global pools for load-balancing queues
  protected static SharedDeque valuePool = new SharedDeque(metaDataSpace, 1);
  protected static SharedDeque remsetPool = new SharedDeque(metaDataSpace, 1);
  protected static SharedDeque forwardPool = new SharedDeque(metaDataSpace, 1);
  protected static SharedDeque rootLocationPool = new SharedDeque(metaDataSpace,
 1);
  protected static SharedDeque interiorRootPool = new SharedDeque(metaDataSpace,
 2);

  // Statistics
  static Timer initTime = new Timer("init", false, true);
```

```
  static Timer rootTime = new Timer("root", false, true);
  static Timer scanTime = new Timer("scan", false, true);
  static Timer finalizeTime = new Timer("finalize", false, true);
  static Timer refTypeTime = new Timer("refType", false, true);
  static Timer finishTime = new Timer("finish", false, true);

  // GC state
  protected static boolean progress = true;  // are we making progress?
  protected static int required; // how many pages must this GC yeild?

  // GC stress test
  private static long lastStressCumulativeCommittedPages = 0;

  /****************************************************************************
   *
   * Instance variables
   */
  protected ObjectReferenceDeque values;  // gray objects
  protected AddressDeque remset;          // remset
  protected ObjectReferenceDeque forwardedObjects; // forwarded, unscanned obj
  protected AddressDeque rootLocations;   // root locs containing white objects
  protected AddressPairDeque interiorRootLocations; // interior root locations

  /****************************************************************************
   *
   * Initialization
   */

  /**
   * Class initializer.  This is executed <i>prior</i> to bootstrap
   * (i.e. at "build" time).  This is where key <i>global</i>
   * instances are allocated.  These instances will be incorporated
   * into the boot image by the build process.
   */
  static {}

  /**
   * Constructor
   */
  StopTheWorldGC() {
    values = new ObjectReferenceDeque("value", valuePool);
    valuePool.newClient();
    remset = new AddressDeque("remset", remsetPool);
    remsetPool.newClient();
    forwardedObjects = new ObjectReferenceDeque("forwarded", forwardPool);
    forwardPool.newClient();
    rootLocations = new AddressDeque("rootLoc", rootLocationPool);
    rootLocationPool.newClient();
    interiorRootLocations = new AddressPairDeque(interiorRootPool);
    interiorRootPool.newClient();
}

  /****************************************************************************
   *
   * Collection
   *
   * Important notes:
   *   . Global actions are executed by only one thread
   *   . Thread-local actions are executed by all threads
   *   . The following order is guaranteed by BasePlan, with each
   *     separated by a synchronization barrier.
   *      1. globalPrepare()
   *      2. threadLocalPrepare()
   *      3. threadLocalRelease()
   *      4. globalRelease()
   */
  abstract protected void globalPrepare();
  abstract protected void threadLocalPrepare(int order);
  abstract protected void threadLocalRelease(int order);
  abstract protected void globalRelease();

  /**
   * Check whether a stress test GC is required
   */
  protected static final boolean stressTestGCRequired()
    throws InlinePragma {
    long pages = Space.cumulativeCommittedPages();
    if (initialized &&
        ((pages ^ lastStressCumulativeCommittedPages) > Options.stressPages)) {
      lastStressCumulativeCommittedPages = pages;
      return true;
    } else
      return false;
  }

  /**
   * Perform a collection.
   *
   * Important notes:
   *   . Global actions are executed by only one thread
   *   . Thread-local actions are executed by all threads
   *   . The following order is guaranteed by BasePlan, with each
   *     separated by a synchronization barrier.:
   *      1. globalPrepare()
   *      2. threadLocalPrepare()
   *      3. threadLocalRelease()
   *      4. globalRelease()
   */
  public void collect() {
    if (Assert.VERIFY_ASSERTIONS) Assert._assert(collectionsInitiated > 0);

    boolean designated = (Collection.rendezvous(4210) == 1);
    boolean timekeeper = Stats.gatheringStats() && designated;
    if (timekeeper) Stats.startGC();
    if (timekeeper) initTime.start();
    prepare();
    if (Plan.WITH_GCSPY) gcspyPrepare();
    if (timekeeper) initTime.stop();

    if (timekeeper) rootTime.start();
    Scanning.computeAllRoots(rootLocations, interiorRootLocations);
    if (Plan.WITH_GCSPY) gcspyRoots(rootLocations, interiorRootLocations);
    if (timekeeper) rootTime.stop();

    // This should actually occur right before preCopyGC but
    // a spurious complaint about setObsolete would occur.
    // The upshot is that objects coped by preCopyGC are not
    // subject to the sanity checking.
    int order = Collection.rendezvous(4900);
    if (order == 1) {
      Scanning.resetThreadCounter();
      setGcStatus(GC_PROPER);
    }
    Collection.rendezvous(4901);

    if (timekeeper) scanTime.start();
    processAllWork();
    if (timekeeper) scanTime.stop();

    if (!Options.noReferenceTypes) {
      if (timekeeper) refTypeTime.start();
      if (designated) ReferenceProcessor.processSoftReferences();
      if (designated) ReferenceProcessor.processWeakReferences();
      if (timekeeper) refTypeTime.stop();
    }

    if (Options.noFinalizer) {
      if (designated) Finalizer.kill();
```

```
    } else {
      if (timekeeper) finalizeTime.start();
      if (designated) Finalizer.moveToFinalizable();
      Collection.rendezvous(4220);
      if (timekeeper) finalizeTime.stop();
    }

    if (!Options.noReferenceTypes) {
      if (timekeeper) refTypeTime.start();
      if (designated) ReferenceProcessor.processPhantomReferences();
      if (timekeeper) refTypeTime.stop();
    }

    if (!Options.noReferenceTypes || !Options.noFinalizer) {
      if (timekeeper) scanTime.start();
      processAllWork();
      if (timekeeper) scanTime.stop();
    }

    if (timekeeper) finishTime.start();
    if (Plan.WITH_GCSPY) gcspyPreRelease();
    release();
    if (Plan.WITH_GCSPY) gcspyPostRelease();
    if (timekeeper) finishTime.stop();
    if (timekeeper) Stats.endGC();
    if (timekeeper) printPostStats();
  }

  /**
   * Prepare for a collection.
   */
  protected final void prepare() {
    long start = Statistics.cycles();
    int order = Collection.rendezvous(4230);
    if (order == 1) {
      setGcStatus(GC_PREPARE);
      baseGlobalPrepare(start);
    }
    Collection.rendezvous(4240);
    if (order == 1)
      for (int i=0; i<planCount; i++) {
        Plan p = plans[i];
        if (Collection.isNonParticipating(p))
          p.baseThreadLocalPrepare(NON_PARTICIPANT);
      }
    baseThreadLocalPrepare(order);
    Collection.rendezvous(4250);
    if (Plan.MOVES_OBJECTS) {
      Scanning.preCopyGCInstances();
      Collection.rendezvous(4260);
      if (order == 1) Scanning.resetThreadCounter();
      Collection.rendezvous(4270);
    }
  }

  /**
   * Perform operations with <i>global</i> scope in preparation for a
   * collection.  This is called by <code>prepare()</code>, which will
   * ensure that <i>only one thread</i> executes this.<p>
   *
   * In this case, it means performing generic operations and calling
   * <code>globalPrepare()</code>, which performs plan-specific
   * operations.
   *
   * @param start The time that this GC started
   */
  private final void baseGlobalPrepare(long start) {
    printPreStats();
    globalPrepare();
  }

  /**
   * Perform operations with <i>thread-local</i> scope in preparation
   * for a collection.  This is called by <code>prepare()</code> which
   * will ensure that <i>all threads</i> execute this.<p>
   *
   * After performing generic operations,
   * <code>threadLocalPrepare()</code> is called to perform
   * subclass-specific operations.
   *
   * @param order A unique ordering placed on the threads by the
   * caller's use of <code>rendezvous</code>.
   */
  public final void baseThreadLocalPrepare(int order) {
    if (order == NON_PARTICIPANT) {
      Collection.prepareNonParticipating((Plan) this);
    }
    else {
      Collection.prepareParticipating((Plan) this);
      Collection.rendezvous(4260);
    }
    if (Options.verbose >= 4) Log.writeln(" Preparing all collector threads for start");
    threadLocalPrepare(order);
  }

  /**
   * Clean up after a collection
   */
  protected final void release() {
    if (Options.verbose >= 4) Log.writeln(" Preparing all collector threads for termination");
    int order = Collection.rendezvous(4270);
    baseThreadLocalRelease(order);
    if (order == 1) {
      int count = 0;
      for (int i=0; i<planCount; i++) {
        Plan p = plans[i];
        if (Collection.isNonParticipating(p)) {
          count++;
          ((StopTheWorldGC) p).baseThreadLocalRelease(NON_PARTICIPANT);
        }
      }
      if (Options.verbose >= 4) {
        Log.write(" There were "); Log.write(count);
        Log.writeln(" non-participating GC threads");
      }
    }
    order = Collection.rendezvous(4280);
    if (order == 1) {
      baseGlobalRelease();
      setGcStatus(NOT_IN_GC);    // GC is in progress until after release!
    }
    Collection.rendezvous(4290);
  }

  /**
   * Perform operations with <i>global</i> scope to clean up after a
   * collection.  This is called by <code>release()</code>, which will
   * ensure that <i>only one thread</i> executes this.<p>
   *
   * In this case, it means performing generic operations and calling
   * <code>globalRelease()</code>, which performs plan-specific
   * operations.
   */
  private final void baseGlobalRelease() {
    globalRelease();
    valuePool.reset();
    remsetPool.reset();
    forwardPool.reset();
```

```
    rootLocationPool.reset();
    interiorRootPool.reset();
  }

  /**
   * Perform operations with <i>thread-local</i> scope to release
   * resources after a collection.  This is called by
   * <code>release()</code> which will ensure that <i>all threads</i>
   * execute this.
   */
  private final void baseThreadLocalRelease(int order) {
    values.reset();
    remset.reset();
    forwardedObjects.reset();
    rootLocations.reset();
    interiorRootLocations.reset();
    threadLocalRelease(order);
  }

  /**
   * Process all GC work.  This method iterates until all work queues
   * are empty.
   */
  private final void processAllWork() throws NoInlinePragma {

    if (Options.verbose >= 4) { Log.prependThreadId(); Log.writeln(" Working on GC
in parallel"); }
    do {
      if (Options.verbose >= 5) { Log.prependThreadId(); Log.writeln("  processing f
orwarded (pre-copied) objects"); }
      while (!forwardedObjects.isEmpty()) {
        ObjectReference object = forwardedObjects.pop();
        scanForwardedObject(object);
      }
      if (Options.verbose >= 5) { Log.prependThreadId(); Log.writeln("  processing r
oot locations"); }
      while (!rootLocations.isEmpty()) {
        Address loc = rootLocations.pop();
        traceObjectLocation(loc, true);
      }
      if (Options.verbose >= 5) { Log.prependThreadId(); Log.writeln("  processing i
nterior root locations"); }
      while (!interiorRootLocations.isEmpty()) {
        ObjectReference obj = interiorRootLocations.pop1().toObjectReference();
        Address interiorLoc = interiorRootLocations.pop2();
        Address interior = interiorLoc.loadAddress();
        Address newInterior = traceInteriorReference(obj, interior, true);
        interiorLoc.store(newInterior);
      }
      if (Options.verbose >= 5) { Log.prependThreadId(); Log.writeln("  processing
gray objects"); }
      while (!values.isEmpty()) {
        ObjectReference v = values.pop();
        Scan.scanObject(v);  // NOT traceObject
      }
      if (Options.verbose >= 5) { Log.prependThreadId(); Log.writeln("  processing r
emset"); }
      while (!remset.isEmpty()) {
        Address loc = remset.pop();
        traceObjectLocation(loc, false);
      }
      flushRememberedSets();
    } while (!(rootLocations.isEmpty() && interiorRootLocations.isEmpty()
              && values.isEmpty() && remset.isEmpty()));

    if (Options.verbose >= 4) { Log.prependThreadId(); Log.writeln("  waiting at barri
er"); }
    Collection.rendezvous(4300);
  }
  /**
   * Flush any remembered sets pertaining to the current collection.
   * Non-generational collectors do nothing.
   */
  protected void flushRememberedSets() {}

  /**
   * Collectors that move objects <b>must</b> override this method.
   * It performs the deferred scanning of objects which are forwarded
   * during bootstrap of each copying collection.  Because of the
   * complexities of the collection bootstrap (such objects are
   * generally themselves gc-critical), the forwarding and scanning of
   * the objects must be dislocated.  It is an error for a non-moving
   * collector to call this method.
   *
   * @param object The forwarded object to be scanned
   */
  protected void scanForwardedObject(ObjectReference object) {
    if (Assert.VERIFY_ASSERTIONS) Assert._assert(!Plan.MOVES_OBJECTS);
  }

  /**
   * Print out plan-specific timing info
   */
  protected void printPlanTimes(boolean totals) {}

  /**
   * Print out statistics at the start of a GC
   */
  private void printPreStats() {
    if ((Options.verbose == 1) || (Options.verbose == 2)) {
      Log.write("[GC "); Log.write(Stats.gcCount());
      if (Options.verbose == 1) {
        Log.write(" Start ");
        totalTime.printTotalSecs();
        Log.write(" s");
      } else {
        Log.write(" Start ");
        totalTime.printTotalMillis();
        Log.write(" ms");
      }
      Log.write("  ");
      Log.write(Conversions.pagesToKBytes(Plan.getPagesUsed()));
      Log.write("KB ");
      Log.flush();
    }
    if (Options.verbose > 2) {
      Log.write("Collection "); Log.write(Stats.gcCount());
      Log.write(":    ");
      printUsedPages();
      Log.write("  Before Collection: ");
      Space.printUsageMB();
      if (Options.verbose >= 4) {
        Log.write("          ");
        Space.printUsagePages();
      }
    }
  }

  /**
   * Print out statistics at the end of a GC
   */
  private final void printPostStats() {
    if ((Options.verbose == 1) || (Options.verbose == 2)) {
      Log.write("-> ");
      Log.write(Conversions.pagesToBytes(Plan.getPagesUsed()).toWord().rshl(10).
toInt());
      Log.write(" KB  ");
```

```
      if (Options.verbose == 1) {
        totalTime.printLast();
        Log.writeln(" ms]");
      } else {
        Log.write("End ");
        totalTime.printTotal();
        Log.writeln(" ms]");
      }
    }
    if (Options.verbose > 2) {
      Log.write("  After Collection: ");
      Space.printUsageMB();
      if (Options.verbose >= 4) {
          Log.write("          ");
          Space.printUsagePages();
      }
      Log.write("          ");
      printUsedPages();
      Log.write("  Collection time: ");
      totalTime.printLast();
      Log.writeln(" seconds");
    }
  }

  private final void printUsedPages() {
      Log.write("reserved = ");
      Log.write(Conversions.pagesToMBytes(Plan.getPagesReserved()));
      Log.write(" MB (");
      Log.write(Plan.getPagesReserved());
      Log.write(" pgs)");
      Log.write("    total = ");
      Log.write(Conversions.pagesToMBytes(getTotalPages()));
      Log.write(" MB (");
      Log.write(getTotalPages());
      Log.write(" pgs)");
      Log.writeln();
  }
}
```