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 management 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

'from space'

'to space'
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

Heap size relative to minimum heap size

Normalized Time

Heap size (MB)

Watson SemiSpace
JMTk SemiSpace
Watson MarkSweep
JMTk MarkSweep
Development Context

• Jikes RVM
  – 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 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 uninterruptible 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 PageResource 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 unsynchronized access to locally owned chunks of the global resource.
- "local" refers to fast unsynchronized, 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 somewhat 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 allocation into the new 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
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.OMap;
import org.vmmagic.unboxed.*;
import org.vmmagic.pragma.*;
/**
 * This class implements a simple allocator without a collector.
 * 
 * @version $Revision: 1.5 $
 * @date $Date: 2004/10/18 11:13:46 $
 */
public class NoGC extends StopTheWorldGC implements Uninterruptible {
    /**
     * 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();
    }
    /**
     * 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();
        }
    }
    /**
     * 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 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 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();
        }
    }
}
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.
 */
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 may need to refer to superclasses appropriately. This exists to support [link BasePlan#getOwnAllocator(Allocator)].
 */
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)].
 */
protected final Allocator getAllocatorFromSpace(Space space) {
    if (space == defaultSpace) return def;
    return super.getAllocatorFromSpace(space);
}

/**
 * Give the compiler/runtime statically generated allocation advice which will 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.
 */
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 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 fact 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.
 */
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</i> one thread 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() {
    //GC Triggered in NoGC Plan. Have you set −X:gc:ignoreSystemGC=true?
}

/**
 * Perform operations with <i>thread−local</i> scope in preparation for a collection.
 */
protected final void threadLocalPrepare() {
    //NoGC Triggered in NoGC Plan. Have you set −X:gc:ignoreSystemGC=true?
}
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
 * L0M).<p/>
 */
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>all thread</i>s execute this.<p>
 * * In this case, it means releasing each of the spaces and checking
 * whether the GC made progress.<p/>
 */
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.<p>
 * * @param obj The object reference to be traced. This is <i>NOT</i>/
 * an interior pointer.<p>
 * @return The possibly moved reference.<p/>
 */
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.<p>
 * * @param obj The object reference to be traced. This is <i>NOT</i>/
 * an interior pointer.<p>
 * @param root True if this reference to <code>obj</code> was held
 * in a root.<p>
 * @return The possibly moved reference.<p/>
 */
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.<p>
 * * @param obj The object in question
 * @return True if <code>obj</code> is a live object.<p/>
 */
public static final boolean isLive(ObjectReference obj) {
  return true;
}
/**
 * Return true if <code>obj</code> is a live object.<p>
 * * @param obj The object in question
 * @return True if <code>obj</code> is a live object.<p/>
 */
public static final boolean willNotMove(ObjectReference obj) {
  return true;
}
****************************************************************************
* Space management
*/
/**
 * Return the number of pages reserved for use given the pending
 * allocation. This <i>includes</i>/i space reserved for copying.<p>
 * * @return The number of pages reserved given the pending
 * allocation, including space reserved for copying.<p/>
 */
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 <i>excludes</i>/i space reserved for
 * copying.<p>
 * * @return The number of pages reserved given the pending
 * allocation, excluding space reserved for copying.<p/>
 */
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>/i.<p>
 * * @return The number of pages available for allocation, <i>assuming
 * all future allocation is to the semi-space</i>/i.<p/>
 */
protected static final int getPagesAvail() {
  return (getTotalPages() - defaultSpace.reservedPages()
    - immortalSpace.reservedPages());
}
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 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;
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_FORWARDED);

/*************************************************************/
/* 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 mb The size of the space in virtual memory in megabytes (MB) *
*/
public CopySpace(String name, int mb, boolean fromSpace) {
super(name, true, false, mb);
this.fromSpace = fromSpace;
pr = new MonotonePageResource(pageBudget, this, start, extent);
}

/*************************************************************/
/* Construct a space of a given number of megabytes in size. */
* The caller specifies the amount virtual memory to be used for * this space &lt;i&gt;in megabytes&lt;/i&gt;. 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 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. */
* The caller specifies the amount virtual memory to be used for * this space &lt;i&gt;as a fraction of the total available&lt;/i&gt;. 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 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);
}
**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)?

```java
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.**

**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)?

```java
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);
}
```

**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.

```java
public final void release(Address start) throws InlinePragma {
    Boolean forwardingPtr = attemptToForward(start);
    if (stateIsForwardedOrBeingForwarded(forwardingPtr)) {
        // Somebody else got to it first.
        while (stateIsBeingForwarded(forwardingPtr))
            forwardingPtr = getForwardingWord(start);
        ObjectReference newObject = ObjectModel.copy(start);
        setForwardingPointer(start, newObject);
        if (scan) {
            Plan.enqueue(newObject);
            // Scan it later
        } else {
            Plan.enqueueForwardedUnscannedObject(newObject);
        }
        return newObject;
    }
    // We are the designated copier
    return newObject;
}
```

```java
public void prepare(Address start) throws InlinePragma {
    this.fromSpace = fromSpace;
}
```

```java
public final ObjectReference traceObject(ObjectReference object) throws InlinePragma {
    if (fromSpace)
        return forwardObject(object, true);
    else
        return object;
}
```
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().toObjectReference();
}

/**
* 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
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_FORWARDED));
  }
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).
 **/
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 resource 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 deliberately 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_MASK.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);
    }
}

Oct 06, 04 21:24  BumpPointer.java  Page 1/2

Oct 06, 04 21:24  BumpPointer.java  Page 2/2
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 existence

// 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 boolean 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 int LOS_SIZE_THRESHOLD = 8 * 1024;
protected static final float LOS_FRAC = (float) 0.1;
protected static final int IMMORTAL = immortalSpace.getDescriptor();
protected static final int META_DATA = metaDataSpace.getDescriptor();
protected 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_GCSPY = 3;
public static final int BASE_ALLOCATORS = 4;

// Statistics
public static final boolean INSIDE_Harness = false;
public static final boolean GENERATE_GC_TRACE = true;
public static final boolean MOVES_TIBS = false;
public static final boolean GENERATE_GC_TRACE = false;

public static final String Id = "$Id: BasePlan.java,v 1.103 2004/10/18 11:13:45 steveb−oss Exp $";

public final static String Id = "$Id: BasePlan.java,v 1.103 2004/10/18 11:13:45 steveb−oss Exp $";
/**
 * Class initializer.  This is executed <i>prior</i> to bootstrap
 * (i.e. at "build" time).  This is where key <i>global</i>-level
 * 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 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_THRESHOLD));
}

/***************************************************************************/

/**
 * Allocation
 */

/***************************************************************************/

/**
 * Run-time check of the allocator to use for a given allocation
 * The <i>Simple</i> 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.
 * The <i>Number</i> method requires the number of bytes to be allocated
 * @param bytes The number of bytes to be allocated
 * @param align The requested alignment.
 */

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</i>
 * instance/<i>/</i> which is allocating into the same space as
 * <code>a</code> in <code>a</code>.
 */

public static 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> in <code>a</code> may be associated with
 * any plan instance.
 */

public final String getSpaceNameFromAllocatorAnyPlan(Allocator a) {
  Space space = getSpaceFromAllocatorAnyPlan(a);
  if (space == null)
    return getAllocatorFromSpace("BasePlan.getOwnAllocator could not obtain space");
  return space.getName();
}

/**
 * Return the space into which an allocator is allocating.  The
 * allocator, <code>a</code> in <code>a</code> may be associated with any plan
 */

Oct 18, 04 21:13 BasePlan.java Page 3/14

Oct 18, 04 21:13 BasePlan.java Page 4/14

Wednesday October 21, 2004 11/20

Printed by Steve Blackburn

org/mmtk/plan/BasePlan.java
package org.mmtk.plan;

import org.mmtk.protection.Allocator;
import org.mmtk.protection.Space;
import org.mmtk.protection.meta.MetaDataSpace;
import org.mmtk.protection.immortal.ImmortalSpace;
import org.mmtk.protection.los.LosSpace;
import org.mmtk.protection.immortal.Allocator;

public class BasePlan {

    protected volatile Allocator immortal;
    protected volatile Allocator los;
    protected volatile Allocator metaDataSpace;

    public static final Word immortalSpace = new Word();
    public static final Word losSpace = new Word();
    public static final Word metaDataSpace = new Word();

    private static Space getSpaceFromAllocator(Allocator a) {
        for (int i=0; i<plans.length; i++) {
            Space space = plans[i].getSpaceFromAllocator(a);
            if (space != null)
                return space;
        }
        return null;
    }

    protected Space getSpaceFromAllocator(Allocator a) {
        if (a == immortal)
            return immortalSpace;
        else if (a == los)
            return losSpace;
        return null;
    }

    protected Allocator getAllocatorFromSpace(Space space) {
        if (space == immortalSpace)
            return immortal;
        else if (space == losSpace)
            return los;
        else if (space == metaDataSpace)
            Assert.fail("BasePlan.getAllocatorFromSpace given meta space");
        else if (space != null)
            Assert.fail("BasePlan.getAllocatorFromSpace given invalid space");
        return null;
    }

    public static final void traceObjectLocation(Address objLoc, boolean root) throws InlinePragma {
        ObjectReference object = objLoc.loadObjectReference();
        ObjectReference newObject = Plan.traceObject(object, root);
        objLoc.store(newObject);
    }

    public static final void traceObjectLocation(Address objLoc) throws InlinePragma {
        traceObjectLocation(objLoc, false);
    }

    public static void enqueue(ObjectReference object) throws InlinePragma {
        Plan.getInstance().values.push(object);
    }

    public static final boolean isForwardedOrBeingForwarded(ObjectReference object) throws InlinePragma {
        return false;
    }

    public static final void enqueueForwardedUnscannedObject(ObjectReference object) throws InlinePragma {
        Plan.getInstance().forwardedObjects.push(object);
    }

    public static final void getBootTimeAvailableBits(int ref, ObjectReference typeRef, int size, Word status) throws InlinePragma {
        return status;
    }

    public static void getSpaceFromAllocatorAnyPlan(Allocator a) {
        Space space = plans[0].getSpaceFromAllocator(a);
        if (space == null)
            return;
        return space;
    }

    // private static Space getSpaceFromAnyPlan(Allocator a) {
    //    for (int i=0; i<plans.length; i++) {
    //        Space space = plans[i].getSpaceFromAllocator(a);
    //        if (space != null)
    //            return space;
    //    }
    //    return null;
    // }

    // protected Space getSpaceFromAnyPlan(Allocator a) {
    //     if (a == immortal)
    //         return immortalSpace;
    //     else if (a == los)
    //         return losSpace;
    //     return null;
    // }

}
/** Trace a reference during GC. This involves determining which   * collection policy applies and calling the appropriate   * <code>trace</code>/</code>method.   *   * @param object The object reference to be traced.   * @param object reference inside obj that must be traced.   * @param root True if the reference to &lt;code&gt;obj&lt;/code&gt; 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))) {
            Log.writeln("ERROR: Suspiciously large delta of interior pointer from object base");
            Log.writeln("object base = "); Log.writeln(object);
            Log.writeln("interior reference = "); Log.writeln(interiorRef);
            Log.writeln("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.&lt;/p&gt;   *   * @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.&lt;/p&gt;   *   * @<i>Non-copying collectors do nothing, copying collectors must   * override this method.&lt;/i&gt;   *   * @param object The object which may have been forwarded.   * @return The forwarded value for &lt;code&gt;object&lt;/code&gt.;   * &lt;i&gt;In this   * case return &lt;code&gt;object&lt;/code&gt;, copying collectors must override   * this method.&lt;/i&gt;   */

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.&lt;/p&gt;   *   * @<i>For many collectors these semantics reflect those of   * &lt;code&gt;traceObject&lt;/code&gt; , which is implemented here. Other   * collectors must override this method.&lt;/i&gt;   *   * @param object The object which may have been forwarded.   * @return The forwarded value for &lt;code&gt;object&lt;/code&gt.;   * &lt;i&gt;In this   * case return &lt;code&gt;object&lt;/code&gt;, copying collectors must override   * this method.&lt;/i&gt;   */

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 &lt;code&gt;true&lt;/code&gt; 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>
* By default do nothing, override if appropriate.</p>
* @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 occurred
*/
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
    // 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>src</code> and <code>dst</code> are the mutated object. Take
 * appropriate write barrier actions.<p>
 * @param src The source of the values to be copied
 * @param dst The offset of the first source address, in
 * bytes, relative to <code>src</code> (in principle, this could be
 * negative).
 * @param dstOffset The offset of the first destination address, in
 * bytes relative to <code>dst</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
    // 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;
}

/**
 * Follow a reference during GC. This involves determining which
 * collection policy applies and getting the final location of the object
 * @return The possibly moved reference.
 */
public static ObjectReference followObject(ObjectReference object) {
    if (Assert.VERIFY_ASSERTIONS) Assert._assert(!Plan.MOVES_OBJECTS);
    return ObjectReference.nullReference();
}

/**
 * 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());
}
/**
 * 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 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;
}

public static void collectionInitiated() throws UninterruptiblePragma {
    collectionsInitiated++;  }

public static boolean gcInProgress() {
    return gcStatus != NOT_IN_GC;
}

public static boolean gcInProgressProper () {
    return gcStatus == GC_PROPER;
}

protected static void setGcStatus (int s) {
    Memory.isync();    gcStatus = s;    Memory.sync();  }

public static void userTriggeredGC() throws UninterruptiblePragma {
}

/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 cleanup 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);
}

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) {}

 public static boolean initialized() {
    return initialized;
}

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

/**
 * 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() {}

protected void gcspyRoots(AddressDeque rootLocations, AddressPairDeque interiorRootLocations) {
    gcspyPreRelease();
    gcspyPostRelease();
}

protected void gcspyPreRelease() {}

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.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.*

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 Exp $";

    /** * Class variables */
    static Timer rootTime = new Timer("root", false, true);
    static Timer scanTime = new Timer("scan", false, true);
    static Timer refTypeTime = new Timer("refType", false, true);
    static Timer finalizeTime = 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;

    public StopTheWorldGC() {
        // 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);

        // Collections
        // Important notes:
        // 1. Global actions are executed by only one thread
        // 2. 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);

    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

    protected ObjectReferenceDeque[] valuePool;
    protected AddressDeque[] remsetPool;
    protected ObjectReferenceDeque[] forwardPool;
    protected AddressDeque[] rootLocationPool;
    protected AddressPairDeque[] interiorRootPool;

    protected boolean progress;
    protected int required;

    // Arrays
    protected AddressDeque[] rootLocations;
    protected AddressPairDeque[] interiorRootLocations;

    // Global pools for load-balancing queues
    protected SharedDeque valuePool;
    protected SharedDeque remsetPool;
    protected SharedDeque forwardPool;
    protected SharedDeque rootLocationPool;
    protected SharedDeque interiorRootPool;

    // Collections
    // Important notes:
    // 1. Global actions are executed by only one thread
    // 2. 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()
private final void globalRelease() {
    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.processSoftReferences();
    if (timekeeper) refTypeTime.stop();
}

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

/**
 * Perform operations with <i>global</i> scope in preparation for a
 * collection. This is called by <code>preparation()</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.<p>
 * @param start The time that this GC started
 */
private final void baseGlobalPrepare(long start) {
}

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

if (timekeeper) globalPrepare();
if (timekeeper) globalRelease();
}

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

if (Options.noFinalizer) {
    if (designated) Finalizer.kill();
}
/**
 * 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.<p>
 * @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();
            gCStatus(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 performs plan-specific
 * operations.<p>
 * private final void baseGlobalRelease() {
 *     globalRelease();
 *     remset.reset();
 *     forwardedObjects.reset();
 *     rootLocations.reset();
 *     interiorRootLocations.reset();
 *     threadLocalRelease(order);
 * }
 */

private final void baseGlobalRelease() {
    globalRelease();
    remset.reset();
    forwardedObjects.reset();
    rootLocations.reset();
    interiorRootLocations.reset();
    threadLocalRelease(order);
    globalRelease();
    remsetPool.reset();
    forwardedObjectPool.reset();
    rootLocationPool.reset();
    interiorRootLocationPool.reset();
    threadLocalRelease(order);
    if (Options.verbose >= 4) Log.writeln("Waiting at barrier");
    Collection.rendezvous(4300);
}

/**
 * Process all GC work. This method iterates until all work queues
 * are empty.
 */

private final void processAllWork() throws NoInlinePragma {
    if (Options.verbose >= 4) {
        Log.reserveThreadId();
        Log.writeln("Working on GC in parallel");
    }
    do {
        if (Options.verbose >= 5) {
            Log.reserveThreadId();
            Log.writeln("processing forwarded(pre-copied) objects");
        }
        while (!forwardedObjects.isEmpty()) {
            ObjectReference object = forwardedObjects.pop();
            scanForwardedObject(object);
        }
        if (Options.verbose >= 5) {
            Log.reserveThreadId();
            Log.writeln("processing root locations");
        }
        while (!rootLocations.isEmpty()) {
            Address loc = rootLocations.pop();
            traceObjectLocation(loc, true);
        }
        if (Options.verbose >= 5) {
            Log.reserveThreadId();
            Log.writeln("processing interior 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.reserveThreadId();
            Log.writeln("processing remaining gray objects");
        }
        while (!values.isEmpty()) {
            ObjectReference v = values.pop();
            Scan.scanObject(v); // NOT traceObject
        }
        if (Options.verbose >= 5) {
            Log.reserveThreadId();
            Log.writeln("remset");
        }
        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.reserveThreadId();
        Log.writeln("Waking at barrier");
    }
    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.pagesToMBytes(Plan.getPagesUsed()));
    Log.writeln();
  }
  if (Options.verbose > 2) {
    Log.write("Collection "); Log.write(Stats.gcCount());
    Log.write(" Before Collection:");
    Space.printUsageMB();
    if (Options.verbose >= 4) {
      Log.writeln("Conversions.pagesToMB(bytes) getTotalPages(): ");
      Log.write("MB");
      Log.writeln();
    }
  }
}

/**
 * Print out statistics at the end of a GC
 */
private final void printPostStats() {
  if (Options.verbose == 1) {
    totalTime.printLast();
    Log.writeln("ms");
  } else {
    Log.write("End");
    totalTime.printTotal();
    Log.writeln("ms");
  }
  if (Options.verbose > 2) {
    Log.write("Collection:");
    Space.printUsagePages();
    if (Options.verbose >= 4) {
      Log.writeln("Conversions.pagesToMB(bytes) getTotalPages(): ");
      Log.write("MB");
      Log.writeln();
    }
  }
}

private final void printUsedPages() {
  Log.write("reserved = ");
  Log.write(Conversions.pagesToBytes(Plan.getPagesReserved()).toWord().rshl(10).toInt());
  Log.writeln("KB");
  if (Options.verbose == 1) {
    Log.writeln("Last:");
    totalTime.printLast();
    Log.writeln("seconds");
  } else {
    Log.write("End");
    totalTime.printTotal();
    Log.writeln("ms");
  }
  if (Options.verbose > 2) {
    Log.write("After Collection: ");
    Space.printUsageMB();
    if (Options.verbose >= 4) {
      Log.writeln("Conversions.pagesToMB(bytes) getTotalPages(): ");
      Log.write("MB");
      Log.writeln();
    }
  }
}

private void printPostStats() {
  if (Options.verbose == 1) {
    totalTime.printLast();
    Log.writeln("ms");
  } else {
    Log.write("End");
    totalTime.printTotal();
    Log.writeln("ms");
  }
  if (Options.verbose > 2) {
    Log.write("Collection: ");
    Space.printUsageMB();
    if (Options.verbose >= 4) {
      Log.writeln("Conversions.pagesToMB(bytes) getTotalPages(): ");
      Log.write("MB");
      Log.writeln();
    }
  }
}