Virtual Machine Design
Lecture 2: Memory management

Antero Taivalsaari
September 2003
Lecture Schedule

- Sep 10: History and overview of VM design
- Sep 17: Memory management
- Sep 24: Interpretation and execution
- Oct 1: Multithreading, synchronization and I/O
- Oct 8: Internals of the Java virtual machine
  (Oct 15: No lecture)
  (Oct 22: No lecture)
- Oct 29: High performance VMs (guest lecture)
- Nov 5: Student presentations begin
Lecture Goals

Memory management is one of the fundamental areas in virtual machine design.

This lecture will present key algorithms and techniques for automatic memory management and garbage collection (GC).
## Essential Components of a (Java) Virtual Machine

<table>
<thead>
<tr>
<th>Class Loader and JAR Reader</th>
<th>Verifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interpreter and Execution Stacks</td>
<td>Compiler (optional)</td>
</tr>
<tr>
<td>Threading System and Thread Scheduler</td>
<td></td>
</tr>
<tr>
<td>Native Interface</td>
<td>Internal Runtime Structures</td>
</tr>
<tr>
<td>Memory System and Garbage Collector</td>
<td></td>
</tr>
</tbody>
</table>
Recap: What is a Virtual Machine?

• A *virtual machine* (VM) is an abstract definition of a computing architecture that is independent of any particular hardware or operating system.

• “Software machine” that runs on top of a “real” hardware platform and operating system.

• Allows the same programs to run “virtually” on any hardware for which a VM is available.
Why Is Memory Management Needed in Virtual Machines?

• Virtual machines are usually “closed” environments.
  – Programs run “inside” the virtual machine.
  – The virtual machine provides the program execution infrastructure, and controls the access to native functions and external resources such as the network, files, etc.

• In order to provide controlled access to resources, the virtual machine needs to take care of memory management as well.

• Memory management inside the virtual machine can be either:
  – manual (explicit), or
  – automatic (implicit).
Basic Memory Management Strategies

1) Static memory management
   - Everything allocated statically.

2) Linear memory management.
   - Memory is allocated and freed in Last-In-First-Out (LIFO) order.

3) Dynamic memory management
   - Memory is allocated dynamically from a large preallocated “heap” of memory.
Static Memory Management

- In static memory management, all memory areas are allocated statically before the execution begins.
  - The Fortran programming language traditionally allocated everything statically.
  - In pure static allocation, even the space for function parameters is allocated statically (no call stack).

- Static allocation is not appropriate for virtual machine implementation:
  - Would prevent recursive function calls.
  - Would prevent multithreading (no re-entrant code)
  - In general: Would prevent any changes to data structures at runtime.
Linear Memory Management

• In linear memory management, memory is allocated as needed in LIFO order.
  – Used, e.g., in the Forth programming language.

• New objects can be allocated freely.

• However, old objects cannot be deleted without deleting all the “younger” objects.

• Linear memory management is only applicable to very simple environments / VMs.
  – Many internal data structures in virtual machines are managed in a stack-like manner, though.
Dynamic Memory Management

• In dynamic memory management, objects can be allocated and deleted freely.
  – Allows the creation and deletion of objects in an arbitrary order.
  – Objects can usually be resized on the fly.

• Most modern virtual machines use some form of dynamic memory management.

• Depending on the implementation, dynamic memory management can be either:
  – Manual: the programmer is responsible for freeing the unused areas explicitly (e.g., malloc/free/realloc in C)
  – Automatic: the virtual machine frees the unused areas implicitly without any programmer intervention.
Automatic Memory Management and Garbage Collection
Automatic Memory Management

• Most modern virtual machines support *automatic dynamic memory management*.

• Automatic dynamic memory management frees the programmer from the responsibility of explicitly managing memory.

• The programmer can allocate memory without having to worry about deallocation.

• The memory system will automatically:
  – reclaim unused memory using a *garbage collector*,
  – expand and shrink data in the heap as necessary,
  – service weak pointers and perform finalization of objects (if necessary).
Benefits of Automatic Memory Management

• Makes programming much easier
  – Removes the problems of explicit deallocation
  – Decreases the risk of memory leaks
  – Simplifies the use of abstract data types
  – Enables proper encapsulation

• Generally: ensures that programs are *pointer-safe*
  – No more dangling pointers

• Automatic memory management improves program reliability and safety dramatically!!
Example

```java
public void foo() {
    MyClass object = new MyClass();
    object.doSomething();
    object = new MyClass();
    // ...
}
```

The previously allocated object instance just became garbage!
Challenges in Automatic Dynamic Memory Management

• How does the memory system know where all the pointers are?

• How does the memory system know when it is safe to delete an object?

• How does the memory system avoid memory fragmentation problems?

• If the memory system needs to move an object, how does the system update all the pointers to that object?
How to Keep Track of Pointers?

• The memory system must be able to know which memory locations contain pointers and which don't.

• Three basic approaches:
  • In some systems, all memory words are tagged with pointer/type information.
  • In some systems, objects have headers that contain pointer information.
  • In some systems, pointer information is kept in separate data structures.
When Is It Safe to Delete An Object?

- Generally, an object can be deleted when there are no more pointers to it.
- All the dependent objects can be deleted as well, if there are no references to them either.
To Compact Memory or Not?

- When objects are deleted, the object heap will contain *holes* unless the heap is *compacted*.

- If a compaction algorithm is used, objects in the heap may move.
  - All pointers to the moved objects must be updated!

- If no compaction is used, the system must be able to manage free memory areas.
  - Often, a *free list* is used to chain together the free areas.
  - Memory allocation will become slower.
  - Fragmentation problems are possible!
Designing A Garbage Collector: Basic Design Choices
Garbage Collection: Basic Design Choices

1) Exact vs. conservative
2) Handle-free vs. handle-based
3) Full vs. partial
4) Cooperative vs. concurrent
5) Single vs. multi-threaded
1) Exact vs. Conservative Garbage Collection

- **Exact (precise) collection**
  - Assumes that all pointers can be identified precisely
    - No accidental memory leaks
    - Enables object migration/copying
    - Enables full heap compaction
    - More complex algorithms to implement

- **Conservative collection**
  - Not all pointers are known; some “guesses” made
    - Occasional memory leaks possible
    - Can cause heap fragmentation
    - Object migration unsafe
    - Simple to implement
2) Handle-free vs. Handle-based Garbage Collectors

- Handle-free collectors
  - Object references are direct
    - Higher performance
    - Harder to relocate objects

- Handle-based collectors
  - Object references are indirect
    - Slower performance
    - Separate handle area needed
    - Easy to relocate objects
3) Full vs. Partial Garbage Collection

- **Full collectors**
  - Perform full garbage collection each time GC is invoked
    - Pause times proportional to heap size
    - Often unacceptable for interactive applications and servers with strict response time requirements
    - Simple to implement; less performance overhead

- **Partial collectors**
  - Heap subdivided into areas or generations that are collected separately
    - Reduces pause times; pause times more evenly distributed
    - Requires a *write barrier*, increasing complexity and slowing down overall performance
4) Cooperative vs. Concurrent Garbage Collection

• Cooperative (stop-the-world) collection
  – Application threads temporarily stopped during GC
    • “Frozen” heap during GC simplifies collection
    • Introduces pauses that may be undesirable
    • Simple to implement; less overall performance overhead

• Concurrent collection
  – Application and the collector may run simultaneously
    • Heap contents may change during collection
    • Application threads might compete with the collector
    • Synchronization and load balancing difficult to implement
5) Single vs. Multi-Threaded Garbage Collectors

• Single-threaded collectors
  – Only one thread performs garbage collection
    • Simple to implement
    • Not scalable to multi-processor machines

• Multi-threaded collectors
  – Multiple threads performing garbage collection
    • Scales better on multi-processor machines
    • Difficult to implement because of synchronization issues, possible race conditions, etc.
    • Not applicable to mobile device VMs (yet!)
Fundamental Garbage Collection Algorithms
Garbage Collection: Fundamental Algorithms

• There are three “classical” categories of garbage collection algorithms:
  – 1) Marking collection
    • For instance, mark-and-sweep, mark-and-compact
  – 2) Copying collection
  – 3) Reference counting collection

• In addition, there are other algorithm categories that are variations of the above:
  – 4) Generational collection
  – 5) Hybrid/adaptive collection
1) Marking Collectors

- Marking collectors typically operate in two phases:
  1) In a *marking* phase, all reachable objects are marked “alive”.
  2) In a *sweep* phase, all dead (unmarked) objects are added to a *free list*, and marks in live objects are cleared.

- Many marking collectors also perform a third phase:
  3) During *compact* phase, all the live objects are packed to the beginning of the heap.

- Marking collectors are not very efficient, but they are simple to implement and have many other desirable properties.
Example: Marking Collection

**Phase 1**: Starting from a set of root objects, mark all reachable objects alive.

**Phase 2**: Add all dead (unmarked) objects to a free list (and optionally compact memory). Clear all mark bits.
2) Copying Collectors

- Copying collectors divide memory into two (or more) semi-spaces.
- During collection, live objects are copied from one semi-space to another, leaving dead objects behind.
- Copying collectors are generally very efficient, since dead objects do not require processing.
- Copying collectors have other nice properties, such as the ability to automatically compact memory and group live objects together.
- However, copying collectors waste memory, since at least one semi-space must be empty.
Example: Cheney Copying Collection
(Communications of the ACM, September 1970)

**Phase 1:** Starting from a set of root objects, copy all reachable (live) objects to a free semi-space.

**Phase 2:** Adjust pointers in the copied objects to point to correct locations, utilizing information in the old space ("pointer swizzling")

During the next garbage collection, the "old" space and "new" space are swapped, and copying is performed again.
3) Reference Counting Collectors

- In reference counting garbage collection, each object keeps track of how many references are pointing to it.
- When the count becomes zero, the object can be reclaimed.
- Reference counting distributes the costs of memory management evenly during execution.
- However, assignment statements become rather expensive.
- Also, cyclic data structures are problematic.
Example: Reference Counting

Each object has a reference count that holds the number of references to that object.

Each time an assignment is performed, reference count is incremented or decremented.

Problem: How to collect cyclic data structures?
4) Generational Collectors

• General observations about garbage collection:
  – Most objects die young!!
  – The more garbage collections an object survives, the more likely it is to stay alive for a long time.

• Modern garbage collectors utilize *object age* in determining the need for garbage collection.
  – Heap is divided into generations based on object age.
  – Objects are “tenured” (promoted) to older generations as they age.
5) Hybrid/Adaptive Collectors

- *Hybrid collectors* use a number of different collection algorithms.
  - For instance:
    - Young objects collected with a fast copying collector.
    - Old objects collected (infrequently) with a compacting collector that maximizes the amount of free available memory.

- *Adaptive collectors* gather statistics/feedback about previous garbage collections, and adapt the behavior of the collector dynamically based on the feedback.
Other Garbage Collection Algorithms

• There are hundreds of garbage collection algorithms.

• For a good overview, read the “bible” of garbage collection:

Fundamental Garbage Collection Algorithms: A Deeper Look
Reference Counting

• Let's take a look at code...
• Let's take a look at code...
• Let's take a look at code...
Performance Considerations
Performance Considerations

- According to studies, garbage collection overhead ranges from 2% to 20%.
  - 10% of total execution time is a good average estimate.
- The number can be larger for extremely dynamic programming languages.
  - For instance, studies in the 1970s/early 1980s indicated that Lisp programs spent 40% of execution time garbage collecting!
  - However, GC algorithms have improved a lot since then.
  - Smalltalk and Self create a huge number short-lived method instances that need to be garbage-collected.
- For Java, the GC overhead is generally much smaller (no more than 5% usually).
Tradeoffs in Garbage Collection

• Best-performing collection algorithms are combinations of simpler algorithms.

• Simple and small basic algorithms have serious space vs. time tradeoffs.
  – For instance, the Cheney copying collector is very efficient, provides fast allocation and packs objects close to each other. However, it requires twice the heap space of a mark-and-sweep collector.

• Naïve mark-and-sweep collectors have serious memory fragmentation problems. Also, these algorithms use recursion which may be a problem in memory-constrained environments.

• Pause times may become excessive with large heaps unless generational collection or reference counting is used.
Avoiding Recursion

- Many garbage collection algorithms are recursive.
  - For instance, the mark-and-sweep collector scans and marks all objects recursively.

- In the worst case, the depth of recursion may be the same as the number of objects in the heap!
  - Recursive algorithms are totally unsuitable for memory-constrained devices.

- Techniques for avoiding recursion:
  - Tail recursion
  - Use an explicit marking stack
  - Pointer reversal techniques (e.g., the Deutsch-Schorr-Waite algorithm)
Heap Compaction Techniques

• 1) Two-finger algorithms
  – Two pointers are used, one to point to the next free location, the other to the next object to be moved. As objects are moved, a forwarding address is left in their old location.
  – Generally applicable only to systems that use fixed-size objects (e.g., Lisp).

• 2) Forwarding address algorithms
  – Forwarding addresses are written into an additional field within each object before the object is moved.
  – These methods are suitable for collecting objects of different sizes.

• 3) Table-based methods
  – A relocation map, usually called a breaktable, is constructed in the heap either before or during object relocation. This table is consulted later to calculate new values for pointers.
  – Best-known algorithm: Haddon-Waite breaktable algorithm; used in Sun's KVM.

• 4) Threaded methods
  – Each object is chained to a list of those objects that originally pointed to it. When the object is moved, the list is traversed to readjust pointer values.

• 5) Semi-space (copying) compaction
  – In copying collectors, compaction occurs as a side-effect to copying.
Real-World Garbage Collectors: Case Studies
Case Study: Garbage Collection in the K Virtual Machine (J2ME)

- KVM uses a simple mark-sweep-(compact) garbage collector.
  - Performs only a mark-and-sweep collection in most cases; free areas kept in a free list.
  - Compaction performed only when the heap is very full and so fragmented that allocation fails.

- Classes and many other data structures kept in a separate “permanent” space.
  - Minimizes the amount of data that need to move during collections.

- No handles; all object (and class) references are direct.
Case Study: Garbage Collection in the HotSpot Virtual Machine (J2SE/EE)

- Three different collectors / heap areas used:

- New generation uses a copying collector.

- Train generation uses a *train algorithm* copying collector.
  - incremental collection of old objects.

- Permanent generation uses a mark-and-sweep-compact collector.
Object Layout in Java

• All Java object instances need to be able to potentially store the following data:
  – Instance variables
    • One slot for each instance variable, two slots for variables of type `long` and `double`.
  – Class pointer
    • `java.lang.Object.getClass()`
  – Monitor reference
    • `synchronized (x) { ... }`
  – Object identity (hashcode)
    • `java.lang.Object.hashCode()`

• Not all this information is needed for every object.
  – > Various optimizations are possible
Example: Object Layout in the K Virtual Machine

GC header word (32 bits):

Object size (24 bits)

GC type (6 bits)

Static bit (unused)

Mark bit

Object reference

GC header

Class pointer

MonitorOrHashCode

Instance variable #1

Instance variable #2

...

Instance variable #n
Object Layout in the HotSpot Virtual Machine

GC header word (32 bits):
- GC status
- Hash value
- Synchronization bits

Instance variable packing is used for data types shorter than four bytes (byte, short, char, ...)

- GC header
- Class pointer
- Instance variable #1
- Instance variable #2
- ...
- Instance variable #n
Garbage Collector Implementation Issues
Updating Garbage Collection Roots

• In order to determine which objects are alive, garbage collection algorithms require information about “root objects”.
  – The root objects serve as the basis for scanning the heap. For instance, in marking or copying collectors, the root objects are usually marked or copied first.

• Root information is typically kept in native pointers or tables outside the actual heap.

• Whenever objects in the heap are moved, the root table must be updated as well!
Object References from Native Code

• When a garbage collection occurs, *native pointers to objects can become invalid!!*

• Example:

  ```c
  ... object* o1 = allocateHeapObject(size1);
  ...
  object* o2 = allocateHeapObject(size2);
  ... 
  ```

• Problem: Second allocation may invalidate *native pointer o1*!

• The virtual machine designer must safeguard all *native pointers inside the virtual machine!!*
Global Roots and Temporary Roots

• In KVM, two mechanisms are provided to safeguard native pointers:

➊ A *global root* mechanism allows the definition of native object pointers that are valid as long as the virtual machine is running.

➋ A *temporary root* mechanism allows the definition of native object pointers that are valid inside a user-defined scope.

• In KVM, these are implemented as C macros.
• Similar mechanisms are needed in all virtual machines implemented in native/C code.
KVM Temporary Roots: Example

```
START_TEMPORARY_ROOTS
    DECLARE_TEMPORARY_ROOT(object*, o1, NULL);
    DECLARE_TEMPORARY_ROOT(object*, o2, NULL);
    ...
    o1 = allocateHeapObject(size1);
    ...
    o2 = allocateHeapObject(size2);
    ...
END_TEMPORARY_ROOTS
```
Write Barriers in Generational GC

- Generational garbage collectors need to keep track of pointers from older to younger generations so that younger generations can be garbage collected without inspecting every object in the older generation(s).

- The set of locations potentially containing pointers to newer objects is often called the **remembered set**.

- At every pointer store, the system must ensure that the updated location is added to the remembered set if the store creates a reference from an older to a newer object.

- This mechanism is usually referred to as a **write barrier**.

- An efficient write barrier implementation is essential to ensure efficient generational garbage collection.

- For a good summary of the problem, refer to:
Comments on Debugging Garbage Collector Implementations

• Garbage collector implementations are very error-prone and tricky to debug.
  – Different programs allocate memory in very different ways; it is difficult to anticipate all situations.
  – Some object pointers are very volatile/dynamic, especially those on the execution stacks of threads; native threading will make things even more complicated.
  – The VM developer or native library developer may forget to safeguard native pointers pointing to heap objects.

• Don't trust your garbage collector unless it has been tested in “excessive collection mode”!
  – Enforce a collection at every allocation.
  – Enforce a collection between every bytecode.
Questions?