Virtual Machines

Lecture on

Virtual Machines

Flexibility vs. Performance

Walter Kriha

Overview

- Virtual Machine Technology (History, Concepts)
- Examples
  - Simple PC Simulator
  - VMWare
  - (Java) VM
- Selected Problems
  - Understand Garbage Collection in the Java Virtual Machine
  - Sandbox Model of security and class loading
  - Bytecode Manipulation and Optimization

Goals

- Understand what virtualization means
- Understand how what a virtual machine is and how it works
- Understand Garbage Collection in the Java Virtual Machine
- Understand the sandbox model of security and how it can be applied in embedded control
- Understand concepts of isolation and parallel processing using several VMs

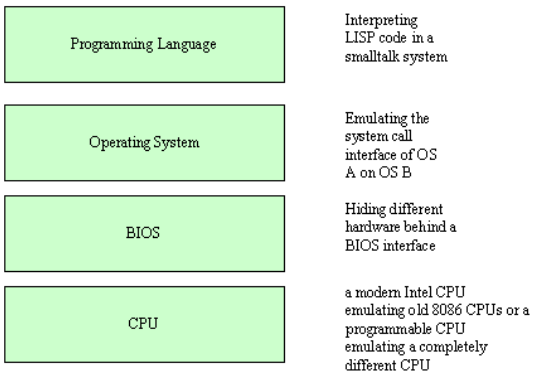
The concept of virtual machines is important in both Java and .Net environments. Just like years ago kernel functions where moved to user processes (like XWindows servers) nowadays kernel functions are moved into virtual machines which create a runtime environment for new languages.

Examples of Virtualization

- A network filesystem mounted at a local machine virtualizes disk storage
- The proc filesystem virtualizes kernel parameters
- Distributed middleware virtualizes the concept of object access

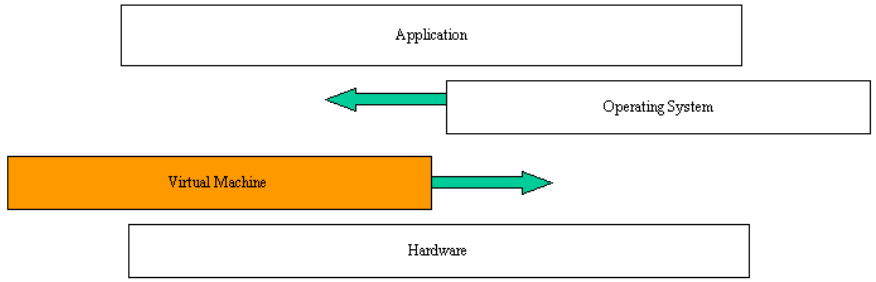
In a way every interface can be used to virtualize a certain functionality. Concrete implementations are hidden behind the interface. A VM typically virtualizes a complete physical processing environment.

Levels of Virtualization



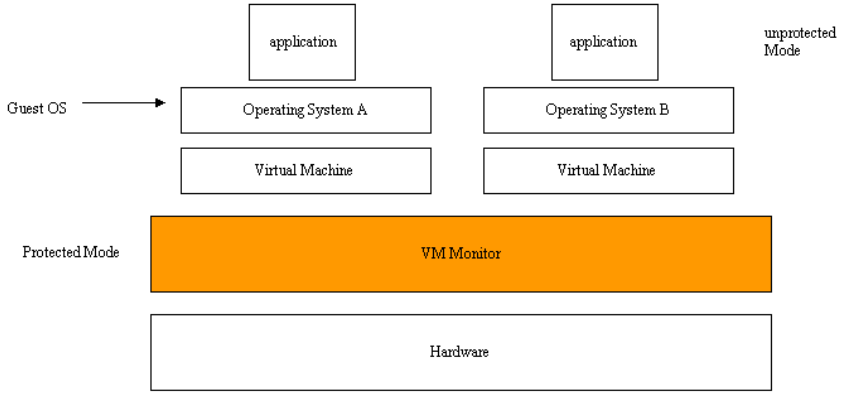
Typically the lower the level of virtualization the better (seamless) can the virtualization perform. It is usually not enough to emulate e.g. the MS-DOS interface if you want to run DOS programs on a different machine. Because DOS programs frequently program against the BIOS or even the hardware directly. PC Simulators usually have to re-build the hardware of a PC

History: Isolation from Change



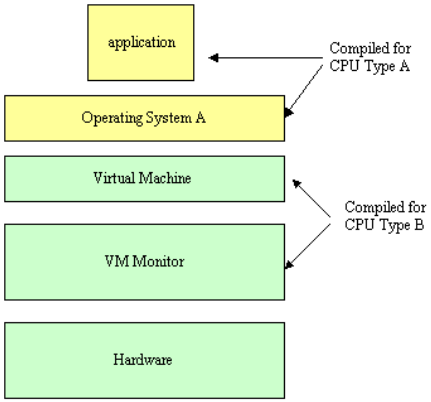
An application written directly to the hardware gives the best performance at the cost of high software efforts for programming low-level tasks like I/O handling etc. And another problem appeared: hardware could no longer be changed/evolved as this would break applications. To decrease the programming effort operating systems where introduced as mediators between applications and hardware. If hardware changed, the operating system had to be adjusted. Soon IBM discovered the value of another interface in front of the hardware: the virtual machine layer. Perhaps invented from sheer necessity to run older software on new hardware the VM design pattern soon became standard in IBM even for new machines.

The concept of a virtual machine



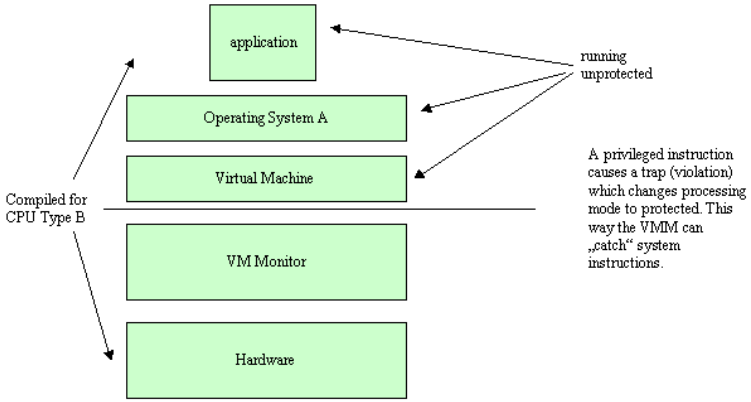
A virtual machine is a runtime environment for applications written against ist interfaces. A virtual machine can use an operating system or can be ported directly onto a specific hardware. If a VM has also a concept of users and processes it could as well be called an operating system.

Interpreting VMs



Here the VM needs to act as an INTERPRETER which reads the foreign CPU instructions and converts the commands to CPU type B instructions. This type of VM is extremely flexible but is usually not very fast due to the interpretation.

Direct Emulation VMs



Application and guest OS use the same CPU instruction set as the hosting system. Therefore regular (non-privileged) instructions can run at full CPU speed. Special (privileged) instructions which would change the state of the overall system or hinder other VMs are trapped and emulated by the VMM. A CPU is well suited for direct emulation if no special instruction can be called from unprivileged code without a trap happening.

Examples of VMs

- Simple PC Simulator (an interpreting VM)
- A historical mistake: 8086 mode in 386 CPU
- VMWare (hosted, direct emulating VM)
- XEN (with ported guest OS)
- (Java) VM (General processor for intermediate language)
- Terra (Trusted Virtual Machine Monitor)

The first example was an extension to a public-domain PC simulation engine which was made to run on National-Semiconductor cpu's, using X-Windows for a UI.

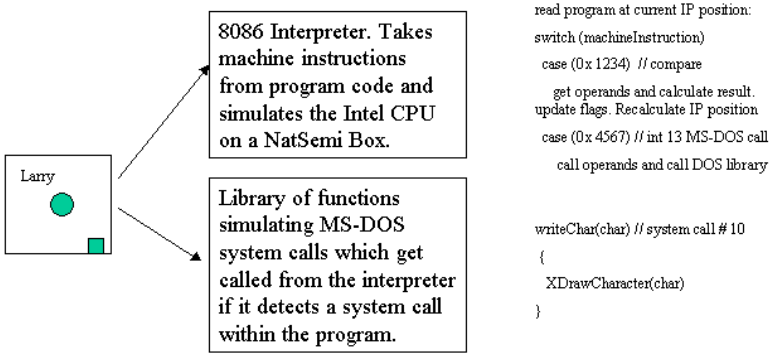
Building a Simple PC-Simulator

Ingredients:

- Simple 8086 Interpreter (a VM written in C, open source, incomplete and buggy)
- PC Architecture handbook (IBM) with all hardware and addresses
- List of BIOS calls
- List of MS-DOS system call interfaces (interrupts)
- A BIOS and DOS copied from an IBM PC
- SINIX on MX500 Multiprocessor
- Leisure Suit Larry (for testing purposes of course)
- Intel 8086/186/286 CPU instruction manual

The goal was to get MS-DOS based programs to run on a National-Semiconductor based multiprocessor

What Larry Needs...

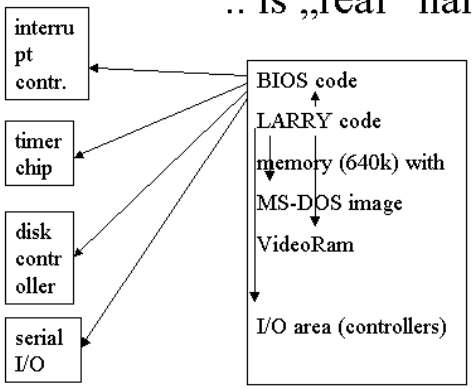



```
read program at current IP position:
switch (machineInstruction)
 case (0x1234)  // compare
   get operands and calculate result.
update flags. Recalculate IP position
 case (0x4567) // int 13 MS-DOS call
   call operands and call DOS library


writeChar(char) // system call # 10
{
  XDrawCharacter(char)
}
```

Sounds quite simple but is horribly wrong: A system is more than just a CPU. In/Out instructions for hardware access where missing in the interpreter. Actually – HARDWARE was missing too!! And MS-DOS turned out to have a horrible number of system calls which one would not like to re-implement. But worse: MS-DOS turned out to be the wrong VM emulation layer at all because programs used other (deeper) interfaces at the same time for performance reasons (e.g. BIOS or direct hardware access)

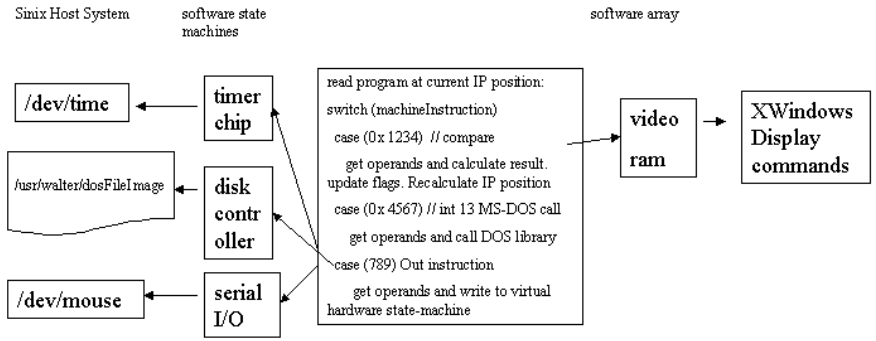
.. is „real“ hardware




```
read program at current IP position:
switch (machineInstruction)
 case (0x1234)  // compare
   get operands and calculate result.
update flags. Recalculate IP position
 case (0x4567) // int 13 MS-DOS call
   get operands and call DOS library
 case (789) Out instruction
   get operands and write to virtual
hardware state-machine
```

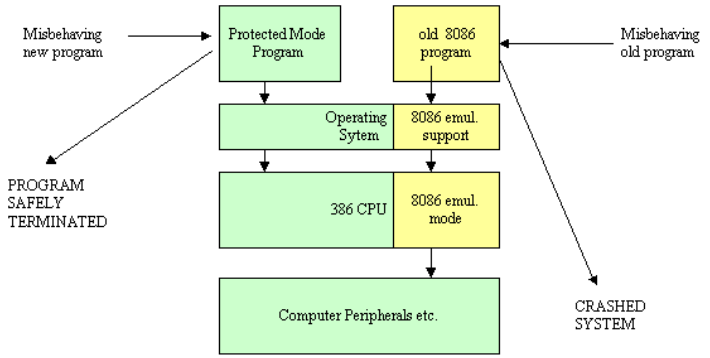
A most other DOS programs Larry used not one but three different architectural layers to get/use system resources. It used DOS system calls, called BIOS functions directly and on top of this knew hardware addresses in the I/O area and used them to manipulate ICs directly. This made it clear that merely re-writing MS-DOS in a library would not suffice. Nor would a re-write of the BIOS do. I decided to use existing BIOS/DOS code and just simulate the hardware by turning the 8086 emulator into a virtual PC.

The self-made Virtual PC



The VM booted a real MS-DOS version from a DOS Filesystem Image within a regular Sinix file. Some hardware was re-built as software state machines but some BIOS interfaces where just simulated directly in software. Video is always a problem because of the huge amount of data involved. The array simulating video array was split in different zones with dirty bit logic and block updates where used in XWindows to update the screen. CGA type computer games and some DOS based maintenance tools where able to run on this virtual PC. Even some Intel 186 and 286 instructions where implemented but no protected mode stuff. I could memorize Intel machine instructions after that for a while...

Simulated 8086 Mode on 386 CPU



The so called DOS-boxes e.g. on Unix or OS/2 machines used this emulation mode but experienced a loss of system stability. Security was basically non-existent for programs running in this special mode. But OS/2 made another mistake: never offer a compatibility mode with some Microsoft feature: Customers will never port their software to your API then. The same is true of DCOM-CORBA bridges from omg. Apple knows this better!

VMWare: hosted, direct emulation VM

Lessions learned:

•Dont re-invent the wheel. Use as much existing software as possible

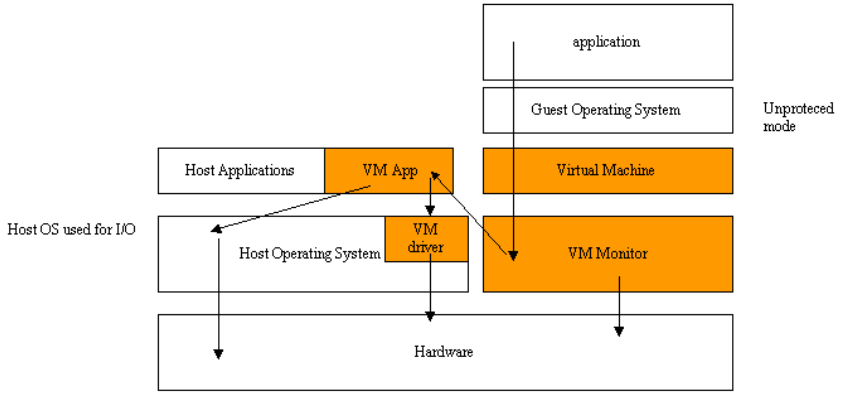
•Run as much code as possible natively on a CPU. Trap privileged instructions. This will ensure good performance

•Define a standard hardware environment which you have to simulate.

•Use an existing host operating system for convenience

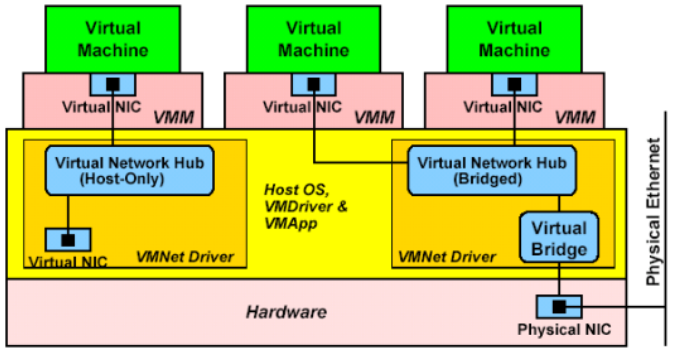
VMWare provides high-speed virtual machines for the PC platforms. It runs on Linux and Windows operating systems. A key feature is the direct emulation mode where code is not interpreted but runs natively on a CPU.

VMWare virtual machine architecture



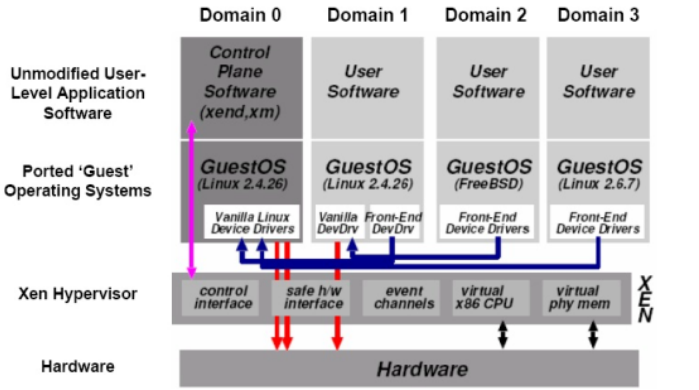
All I/O instructions are intercepted by the VMM and re-directed to a VM applications. From there regular system calls are used against the host OS to fulfill the requests. Switching from VM World to host world is called a „world switch“ and is rather expensive. For every request both drivers (guest and host OS) are processed and several context switches happen. For a description see Sugerman et.al in resources.

Virtualizing I/O in VMWare



From: Ping-Chuan Lai, Virtual machine – memory and I/O management. Notice that a vmware VM can have its own MAC addresss e.g. Also notice that not every I/O request needs to go through the host OS and driver. If it only changes the state of the virtual NIC no world switch is needed.

Xen 2.0 Architecture



From: Ian Pratt et.al (see resources). Note that XEN needs ported guest operating systems unlike vmware which runs with the originals. The XEN approach can deliver better performance with the same level of isolation as provided by vmware.

The Java Virtual Machine

- Abstract virtual machines specification (class format, instructions, stack machine etc.)
- Implementation issues (language, performance)
- Runtime (instance) issues (user, isolation)

Bill Venners „Inside the Java Virtual Machine“ is an excellent introduction (see resources). Please note that the „Java“ VM is not really specific to the Java language and can execute other high-level languages which are compiled to the intermediate bytecode as well. A Java principle therefore is to consider the Java environment the main API. As a consequence e.g. security decisions are implemented in the Java environment (SecurityManager, AccessController) and not hidden in the VM.

Abstraction in Computing

„Even hardware manufacturers have taken to abstraction, the Transmeta line of CPUs are sold as x86 CPUs but in reality they are not. They provide an abstraction in software which hides the inner details of the CPU which is not only not x86 but a completely different architecture. This is not unique to Transmeta or even x86, the internal architecture of most modern CPUs is very different from their programming model.“ (Nicholas Blachford)

Compare this view to some statements from Object-Oriented computing: The implementation object model is a one-to-one clone of the business object model (or analysis model). This would tie implementations structurally to specific form of business problems and make them extremely inflexible. Changes in business models would require changes in implementations. So what? This is happening all the time. What if we would consider a framework for business applications as a virtual machine. Business object models need to be compiled against the abstract interface of the virtual machine and the implementation of all business functionality inside the VM would be free. This of course could again run on a VM like the Java or .NET VMs. How does this compare to Model-driven Architecture?

Java Source to Execution Model

1.loading (find binary representation)

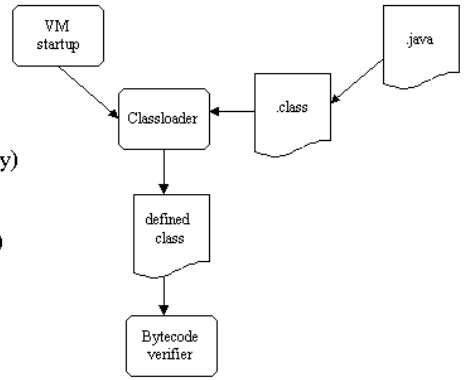
2 linking

Verification (legal operands, type safety)

Preparation (init standard defaults)

Resolution (symbolic names to indices)

3. Initialization (class members static values)



From the Java VMspec Version 2. For performance and resource reasons the J2ME versions of the VM perform pre-verification of bytecodes. This caused already some security leaks on embedded platforms.

Class File Format

CLASSFILE	
magic_number	4
version_numbers	4
constant_pool_count	2
constant_pool	n
access_flags	2
this_class	2
super_class	2
interfaces_count	2
interfaces	n
fields_count	2
fields	n
methods_count	2
methods	n
attributes_count	2
attributes	n

The java class file format is an extremely compact way to represent the information from a java source file. The compiler has added the java bytecodes for the methods and recorded all the relations between this class and superclass, interfaces etc. Note that this format is NOT java specific. It could represent other languages as well (see Groovy). Diagram taken from Kutschke et.al, (see resources)

The Classloader Mechanism

- Load java .class files (from anywhere), load resources and native code .dll's
- Isolate different classes with same name by using different class loaders
- Allows reloding of classes by reloading their classloader
- Rules: ask parent class loader first. Do not search by calling lower level class loaders.

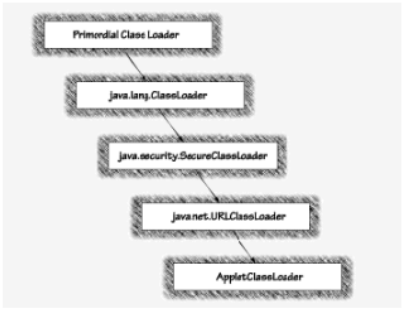
From Bill Martin, Websphere... (see resources). Remember that a typical Java VM runs within ONE address space and does not provide OS type isolation based on virtual memory. The order and mechanism of class loading is therefore used in Java to achieve code isolation. Based on where java code come (from which class loader) different privileges can be assigned (see Java 2 Security, granting permissions)

Application Server Class Loaders

System level	Bootstrap CL (java.*, jre/, –Xbootclasspath) Top level
	Extensions CL (jre/lib/ext, property java.ext.dirs)
	CLASSPATH CL (everything from classpath var)
Runtime	AppServer CL (no application code)
Firewall/Security	Protection CL (against normal CLs loading system classes)
Application level	Application CL (one for all or one for each. Search orders parent first/last)
	Web Module CL (one for all or one for each)
	Shared Library CL (for common libs)

From Bill Martin, Websphere... (see resources). Early application servers where suffering from different apps bringing different versions of utility libraries etc. with them and caused conflicts with previously installed or application server special versions. Java Security knows also a security classloader which is active when Java 2 Security is enabled.

Java Classloader Hierarchy



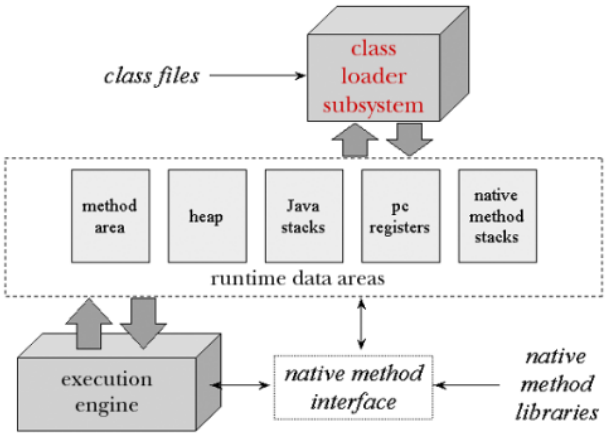
From Edward Felten, Securing Java

Classloader code example

```
public class SimpleClassLoader extends ClassLoader {
    public synchronized Class loadClass(String name, boolean resolve) {
        Class c = findLoadedClass(name);
        if (c != null) return c;
        try {       c = findSystemClass(name);
                    if (c != null) return c;
        } catch (ClassNotFoundException e) {}
        try {       RandomAccessFile file = new RandomAccessFile("test/" +
                    name + ".class", "r");
                    byte data[] = new byte[(int)file.length()];
                    file.readFully(data);
                    c = defineClass(name, data, 0, data.length);
        } catch (IOException e) {}
        if (resolve) resolveClass(c);
        return c;
    }}
```

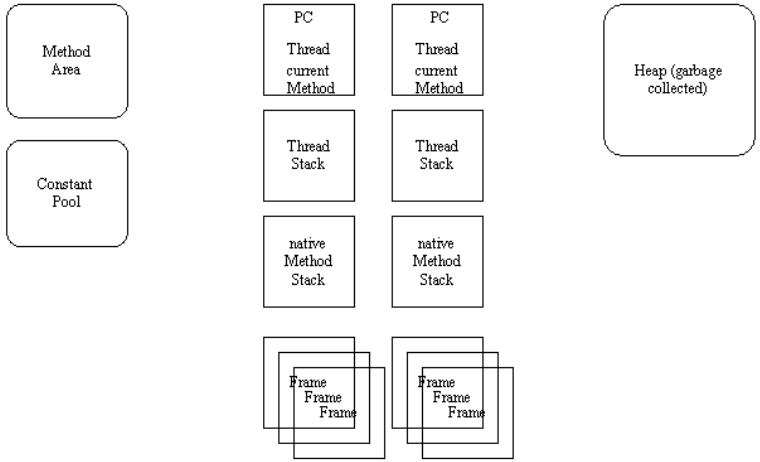
From Kutschke et.al., Order of resolving requests is extremely important. user written classloaders are today a major source of security problems in Java applications.

VM Runtime



Bill Venners „Inside the Java Virtual Machine“ is an excellent introduction (see resources).

Abstract Machine



Bill Venners „Inside the Java Virtual Machine“ is an excellent introduction (see resources). Notice the different spaces for every thread. And every method call is represented by a special frame object which encapsulates the execution engine (see later). Class files contain symbolic values which are later converted and stored in the respective pool/area.

Instruction Set

opcode	byte	short	int	long	float	double	char	reference
Tipush	bipush	sipush						
Tconst			iconst	lconst	fconst	dconst		aconst
Tload			iload	lload	fload	dload		aload
Tstore			istore	lstore	fstore	dstore		astore
Tinc			iinc					
Taload	baload	saload	iaload	laload	faload	daload	caload	aaload
Tastore	bastore	sastore	iastore	lastore	fastore	dastore	castore	aastore
Tadd			iadd	ladd	fadd	dadd		
Tsub			isub	lsub	fsub	dsub		
Tmul			imul	lmul	fmul	dmul		
Tdiv			idiv	ldiv	fdiv	ddiv		
Trem			irem	lrem	frem	drem		
Tneg			ineg	lneg	fneg	dneg		
Tshl			ishl	lshl				
Tshr			ishr	lshr				
Tushr			iushr	lushr				
Tand			iand	land				
Tor			ior	lor				
Txor			ixor	lxor				

The JVM opcodes are only 1 byte long and encode in most cases the type of the operand as well (iload, lload, fload, dload e.g.)

VM Bytecode Example

```
class stringtest {
static public void main(String []
args){
  String test1 = "AnExample";
  String test2 = "AnExample";
  if (test1 == test2){
  System.out.println("Sach ich doch");
  }
 }
}
```

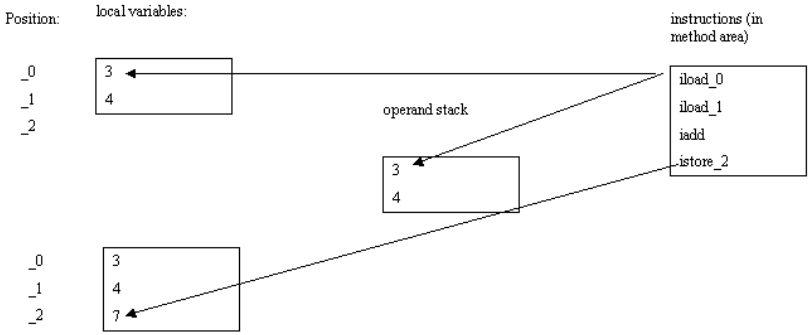
generated with javap –l –c from stringtest.class. Notive the index into the constant pool for strings

```
Line numbers for method stringtest()
   line 1: 0
Method void main(java.lang.String[])
   0 ldc #2 <String "AnExample">
   2 astore_1
   3 ldc #2 <String "AnExample">
   5 astore_2
   6 aload_1
   7 aload_2
   8 if_acmpne 19
  11 getstatic #3 <Field java.io.PrintStream out>
  14 ldc #4 <String "Sach ich doch">
  16 invokevirtual #5 <Method void
println(java.lang.String)>
  19 return
Line numbers for method void
main(java.lang.String[])
   line 5: 0
   line 6: 3
   line 8: 6
   line 10: 11
   line 12: 19
```

Loadtime Bytecode Verification

Please note that the type safety of a Java program is checked by the VM during class loading. This is fundamentally different to the compile time concept of e.g. c++ protection tokens (private, protected etc.)

Frame (execution engine)



Operands are pushed onto the operand stack – the only place where calculation can happen. In this case two variables where copied from the local variable array (_0, _1) to the operand stack. Iadd pop's both, performs the calculation and puts the result back onto the stack. From there, istore_2 takes the current value and moves it into position 3 of local variables.

Garbage Collection

A garbage collector claims all memory that can no longer be reached from within the program. In most cases the program stack(s) are walked and all memory that is referenced by something is marked as still needed. After marking memory the unmarked rest becomes garbage.

For a discussion of garbage collection techniques see Paul Wilsons paper (resources).

Empirical Data on Memory Use

1. Most memory which is allocated has a very short lifetime. E.g. strings in Java
2. Some memory has very long lifetimes, e.g. cached reference data (constants for languages and countries etc.)
3. Different ways of separating used from unused memory exist and they work differently for newly allocated and old memory.
4. Different applications have different garbage collection needs. Some applications don't allow large gaps in processing. Other applications need large throughput and cannot wait for one thread collection garbage.
5. Garbage collection can be optimized if the application is traced for speed and object creations

Without empirical data on an application it is impossible to optimize garbage collection. Most VM's provide many options to create performance data.

Why different Garbage Collectors?

- Application Types: no stops in multimedia apps. Huge memory in transactional enterprise apps. No GC problems in GUI apps because of wait-times.
- Data types: short lived strings vs. long lived reference data.
- Machine types: small machines running one CPU and little ram: GC times are short but frequent. Large machines with many CPUs and large ram: GC times are long but possibly infrequent causing problems for near-realtime apps.

Different Garbage Collectors will work differently in all these cases.

The traditional mark and sweep algorithm

Step one:

Stop all concurrently running threads

Step two:

Start walking all stacks and mark all referenced objects

Step three:

Compact the residual heap memory. Unused memory is freed and can now be re-used

The core characteristics of a mark and sweep algorithm are:

-either user threads OR the GC run. This basically stops the application for the time the GC is running.

-Compacting memory is expensive

We will look at the characteristics in detail and see how other GC algorithms handle them.

„Stop the world“ vs. „Concurrent GC“

- All application threads stopped.
- Bad performance for near-realtime applications
- Everything that cannot stop is treated badly by this algorithm
- Large memories increase the difficulties because it makes the runtime for GC even longer

- The GC runs concurrently with application threads. Those threads are stopped only for extremely short mark or re-mark phases.
- The applications keeps responsiveness even for short intervals.
- High demand for memory can force the GC to fall back to a regular mark-and-sweep.

Multimedia applications can take maximum profit from using a concurrent GC.

Amdahls law and GC serialization

Single-threaded vs. parallel GC

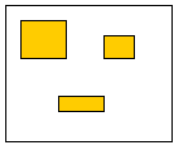
- Only one thread per VM is responsible for garbage collection.
- Large Memory causes large runtimes for the GC
- Multiple CPUs are only used for application threads (which make garbage).
- Collection cannot profit from multiple CPUs

- Several threads perform garbage collection concurrently.
- The application threads are stopped while the GC threads are running.
- Application stop-time is short because more GC threads mean shorter wait-times.
- Large memories can be checked by parallel GC threads.

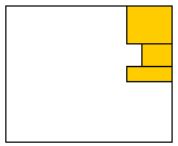
Large Servers (more than 8 CPUs) with lots of memory (more than 4 GB) will create huge amounts of garbage. They need parallel collection to keep up with the generation of garbage.

Copying vs. compacting GC

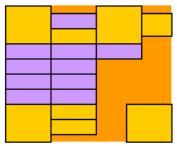
Phase one: valid memory is marked



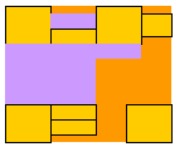
Phase two: valid memory is copied over into a new memory region. The old region is declared free. Next time the two regions are reversed.



Phase one: valid memory is marked

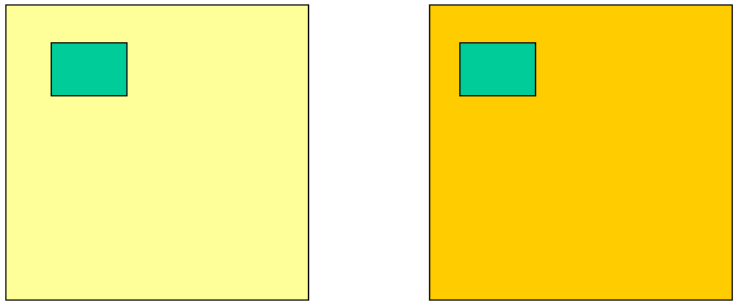


Phase two: free memory is joined with free adjacent memory to form larger free blocks (compacted). The valid memory is not moved.



Both techniques have different advantages. Copying only the memory still in use over to a new region pays off if most of the memory has become invalid and only small numbers need to be copied. The is typically true for the „new generation“ of memory allocated. Older blocks can become invalid as well but this does not happen so frequently. This means that instead of copying everyting to a new region only the memory no longer in use is recombined into larger free blocks. Moving blocks in memory also requires one indirection: a client cannot have a pointer (address) directly to memory because this would be wrong after moving the memory.

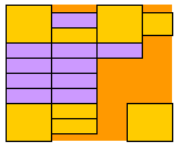
New Generation vs. Old Generation



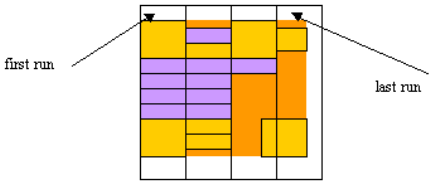
A generational GC first allocates newly requested memory in the „new generation“ area. After a while if this memory block is still in use it is copied over to the „old generation“ and the new generation region is released. Differentiating both memory regions allows the Garbage Collector to check the new generation frequently (new memory is often only short-lived) and the old generation (which often contains constent reference data) rarely.

Complete vs. Incremental GC

All memory is checked for liveness



Memory is divided in „trains“ which are checked in different GC runs.



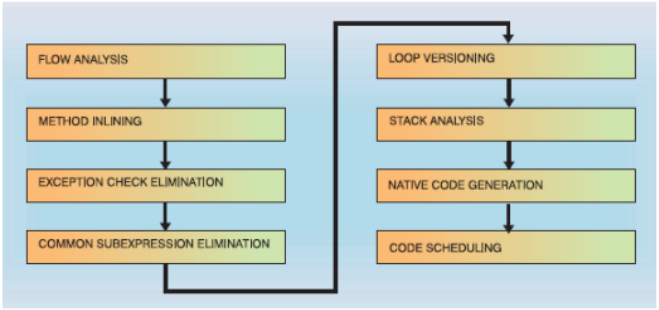
The advantage of an incremental GC is simply that the stop-times for applications are shorter, especially with large memories. But incremental collection can lead to not enough memory being freed and the GC must fall back to regular complete mark and sweep techniques.

How to diagnose memory allocation problems

- set GC to verbose. This generated lots of statistics about allocated objects, GC runtimes and effectivity.
- If the GC part in your application is beyond 15% you have a problem with too many objects being created.
- Run a memory leak checker like purify, optimizeit or jprobe to check your allocations.

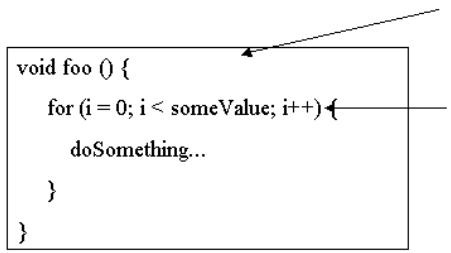
Too many objects allocated is the number one performance problem for java applications. The other bummer is underestimating network latency in distributed applications.

Just-In-Time Compilation



A JIT compiler converts bytecode „on the fly“ to native machine code. This can improve execution performance a lot but places a burden on the startup time due to the necessary compilation step. (From T.Sugerman et.al.)

Avoiding Premature Optimization



method counter tracks invocations

by matching „loop patterns“ the VM tries to detect loops to keep a counter on the loop count.

A JIT compiler can get help from the VM to decide when to compile bytecodes. If a method is not used frequently it is generally not worth compiling it into a faster version! Therefore the VM can keep counters on methods and loops. From T.Sugerman et.al. Pg 177-179.

Resources (1)

- A list of optional switches for the HotSpot VM: http://java.sun.com/docs/hotspot/VMOptions.html
- An excellent article on 1.4.1 garbage collection: "Improving Java Application Performance and Scalability by Reducing Garbage Collection Times and Sizing Memory Using JDK 1.4.1," Nagendra Nagarajayya and J. Steven Mayer (Sun Microsystems, November 2002): http://wireless.java.sun.com/midp/articles/garbagecollection2/
- J2SE 1.4.1 boosts garbage collection. Three new algorithms target near real-time applications
- Java HotSpot Performance FAQ java.sun.com/docs/hotspot/PerformanceFAQ.html
- Paul R. Wilson, "Uniprocessor Garbage Collection Techniques", http://www.cs.utexas.edu/users/oops/papers.html. The best resource on general GC questions.

Resources (2)

- J.Sugerman, Ganesh Venkitachalam, Beng-Hong Lim, Vmware Inc. Virtualizing I/O Devices on VMWare Workstations Hosted Virtual Machine Monitor (Usenix 2001 Proceedings). Excellent introduction to VMwares concept of direct emulation and hosted VMs. http://www.usenix.org/publications/library/proceedings/usenix01/sugerman/sugerman_html/
- Hideku Eiraku, Yasushi Shinjo, A lightweight virtual machine for running user-level operating systems.
- Benjamin Atkin, Emin Gün Sirer, PortOS: An Educational Operating System for the POST-PC Environment
- Ahmad-Reza Sadeghi, Christian Stüble et.al., European Multilateral Secure Computing Base. On PERSEUS and other approaches to trusted computing. Good links.
- Dinda Winter, Resource Virtualization Reading List, http://www.cs.northwestern.edu/~pdinda/virt-w04/reading_list.pdf excellent links on all kinds of VM research.

Resources (3)

- T.Suganuma et.al., Overview of the IBM Java Just-in-Time Compiler. IBM Systems Journal Vol.39, No 1. Good explanation of the inner workings of a JIT compiler. Includes incremental compilation technology.
- Eric Kohlbrenner et.al. Demonstration of an IBM VM concept. http://cne.gmu.edu/itcore/virtualmachine/demo.htm This is an applet demonstrating the memory closure provided by a VM
- Bytecode manipulation, by Ron Kutschke, Daniel Haag, Mirko Bley and Markus Block, A very good introduction with source examples on how to maniuplate java class files. Explains class file format, VM structure etc. http://www.kriha.de/krihaorg/dload/uni/generativecomputing/generation/Bytecode.zip
- Brian K. Martin, Understanding Websphere V.5 Classloaders. Describes how isolation and sharing can be implemented by chosing clever class-loading strategies. Explains class loader chain used. (developerworks)
- Using the ASM Toolkit for Bytecode Manipulation by Eugene Kuleshov
- http://www.onjava.com/pub/a/onjava/2005/01/26/classloading.html Explains the intricacies of Java class loading.
- Nicholas Blachford on CELL design. http://www.blachford.info/computer/Cells/Cell2.html

Resources (4)

- Ian Pratt et.al, XEN ant the art of virtualization. Describes XEN 2.0 architecture. http://www.cl.cam.ac.uk/Research/SRG/netos/papers/2004-xen-ols.pdf
- Ken Fraser et.al, Safe hardware access with the new XEN virtual machine monitor. Describes the new I/O interface architecture http://www.cl.cam.ac.uk/Research/SRG/netos/papers/2004-oasis-ngio.pdf
- The XEN portal at univ. of Camebridge: http://www.cl.cam.ac.uk/Research/SRG/netos/xen/