In the CLI, a code module can be verified for type safety in a manner similar to Java binary classes, and verifiability is a desirable property to have. The managed C++ compiler has a flag that forces all code to be verifiable (and generates errors when the C++ programmer uses unsafe constructs). Verification helps establish a runtime protection domain that allows untrusted code to execute safely. Nevertheless, unverifiable (unsafe) programs are allowed as part of the managed program environment. A programmer may choose to generate unverifiable code that can still interoperate with verifiable code. In order to use unverifiable code, however, the user must permit it through a security manager.

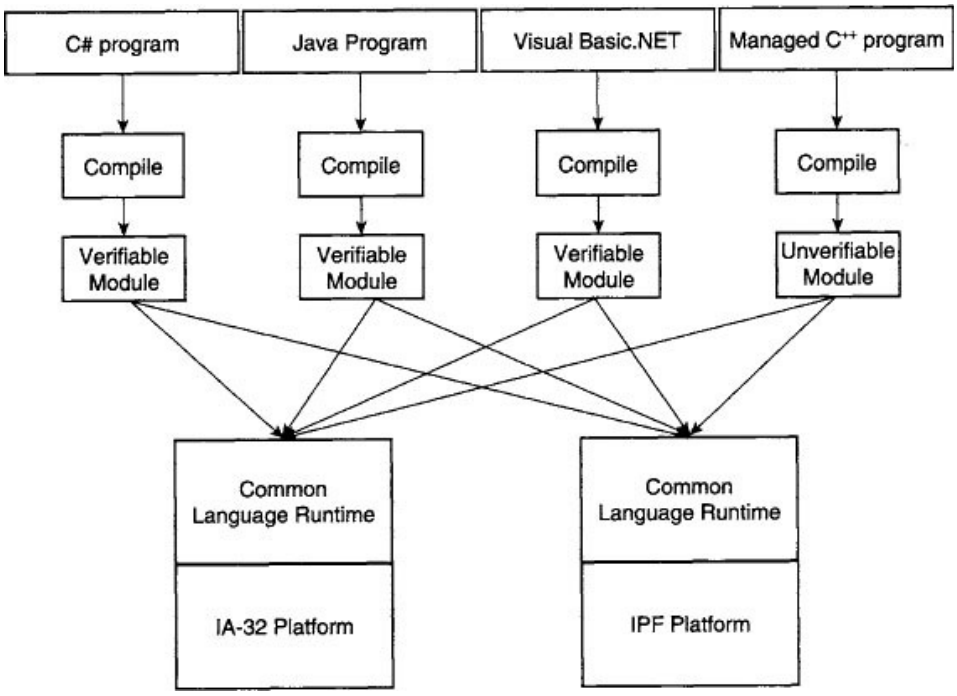


Figure 5.15 The Common Language Interface Supports Language Interoperability. *The CLI allows both platform independence and a high degree of high-level language independence.*

Regardless of whether they are verifiable, all programs must be valid. Some programs may be constructed in an illogical manner; for example, a program may have a stack underflow that is detectable by inspecting the code at load time. A program such as this is invalid (as well as being unverifiable) and should not be allowed to run. Hence, there are three categories of application programs:

those that are verifiable and valid, those that are unverifiable and valid, and those that are invalid. This is in contrast to Java, which only allows two types (verifiable and unverifiable). Furthermore, verifiable and unverifiable programs may be mixed (at the programmer's discretion), so, for example, a verified program can call an unverifiable library routine. Because multiple languages and both verifiable and unverifiable programs are supported in an integrated way, the CLI provides a more complete, all-purpose application program environment than the JVM. This feature may be more significant in the short term as unsafe legacy code is mixed with newly developed verifiable code. In the long term, one would expect that unsafe legacy code will eventually be replaced with safe, verifiable code.

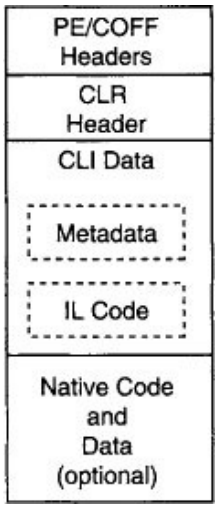


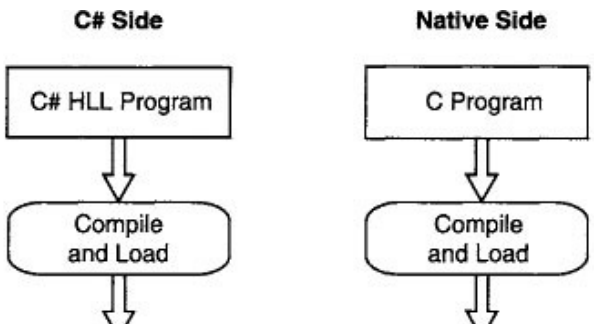
Figure 5.16 Overall Structure of a CLI Module.

CLI modules are organized into assemblies, just as Java binary classes are organized into packages. **Figure 5.16** shows an overview of a typical CLI module. It is contained in a standard Microsoft Portable Executable and Common Object File Format (PE/COFF), further highlighting interoperability; the OS handles it just like any other executable file. The module contains header information and CLI metadata and code, and it may also contain native code and data. The metadata consists of a number of regions that contain, among other things, object definitions and constants (called *streams*

in the CLI). In general the concepts are similar to those in a Java binary class.

It is interesting to compare language interoperability in the CLI with that provided in Java (as illustrated earlier in **Figure 5.13**). An analogous figure for the CLI is given in **Figure 5.17**. Here, all programs compile onto the MSIL bytecodes, including C programs, for example. The data structures are all in the common metadata format. However, as described in the next section, the metadata format includes blocks of data and unmanaged pointers, which an MSIL program can manipulate in any manner (as in C). This code is unverifiable, but it can still execute within the managed environment.

Hence, in the CLI, interoperability is not implemented at the method call level (as in Java), but interoperability is supported in a more integrated way that also extends to data: a type defined in one language can be used across languages. This more highly integrated approach does not come without a cost, however. It may require significant changes to existing language implementations (as was the case with Visual Basic, whose object model had to be changed to the C# model). Achieving interoperability may also require standardization in programming conventions. For example, if one language has case-sensitive variable names and the other does not, then for interoperability they should probably stick to a common denominator and not rely on case sensitivity. The Common Language Specification (CLS) contains a set of standard rules intended to ensure interoperability among high-level languages.



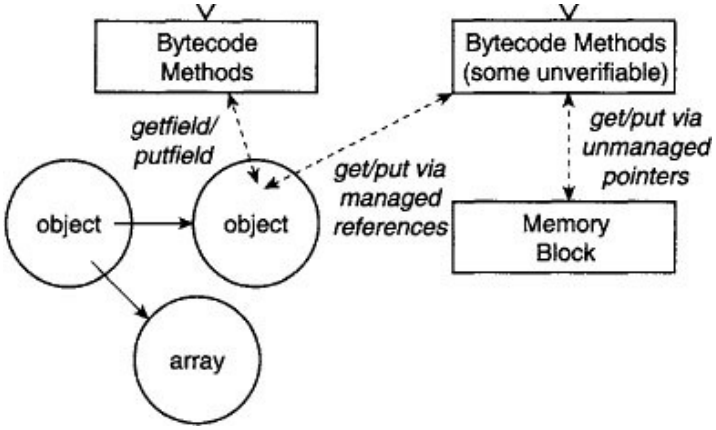


Figure 5.17 Verfiable and Unverifiable Code Interoperate in the CLI. *MSIL bytecode programs interoperate via common metadata, including memory block data, that may be accessed via unmanaged pointers.*

5.5.2 Attributes

The CLI supports attributes to allow the programmer to pass information at run time via compiled metadata that is part of a package. This information may be directed at the runtime software or the executing program. Some attributes are built in; i.e., they are intrinsic to the CLI. Custom attributes are also supported as a way of enabling user-defined extensions. The programmer can assign an attribute to virtually any item held in a module, including fields, classes, methods, or parameters. Then a running program can access the attribute via reflection, implemented with the GetCustomAttribute method. Based on the attribute, the program can then take some action. For example, consider a situation where a number of different companies are working together on a large international engineering design product. Some of the engineers may be working in units of inches and others in centimeters, and each is writing software based on one unit or the other. Traditionally, this type information is conveyed by comments placed in the code or via other external documentation, including word of mouth. However, this places the burden on the user to carefully check for such documentation to avoid mistakes. By assigning an attribute of either [inches] or [centimeters] to an object's data field, the unit being used is automatically conveyed to other programs as part of the meta-

data. A method accessing a data field can use reflection to check the unit attribute and convert the value in the field, if necessary.

Attributes can also pass information to the runtime software—for example, that an object has to have its fields laid out in a specific way (so that it can be accessed via native code, perhaps). A built-in attribute [serializable] is passed to the runtime to indicate than an object may be serialized and should therefore be implemented in a particular way.

5.5.3 Microsoft Intermediate Language

The instructions contained in the intermediate language MSIL (**Lidin 2002**) are similar in concept to Java bytecodes. The MSIL is stack oriented, with all operations taking place via an operand (or evaluation) stack. Rather than provide the same level of detail as we did for Java bytecodes, we instead focus on the significant ways in which the MSIL differs from Java bytecodes.

Figure 5.18 illustrates the MSIL memory architecture. For a given method, a local data area and an argument area are defined, but they are not defined as part of a stack frame as in Java. In an implementation they would probably be held on an implementation-specific stack, but there are no instructions in the MSIL that assume they are part of the same stack. That is, they are simply regions of memory accessible when a method is executing. Rather than having a constant pool, MSIL supports a number of metadata tables, referred to as *streams,* which contain information similar to that held in the Java constant pool. Furthermore, instructions can push constant values, either long (four bytes) or short (one byte), onto the operand stack. The metadata tables are accessed via *tokens* that are supported by the MSIL. A token contains four bytes: One byte is a metadata table (stream) identifier, and the other three bytes point to a particular entry in the table. One of the constant streams holds program-referenced character strings, and there is a special MSIL

instruction that can retrieve a string.

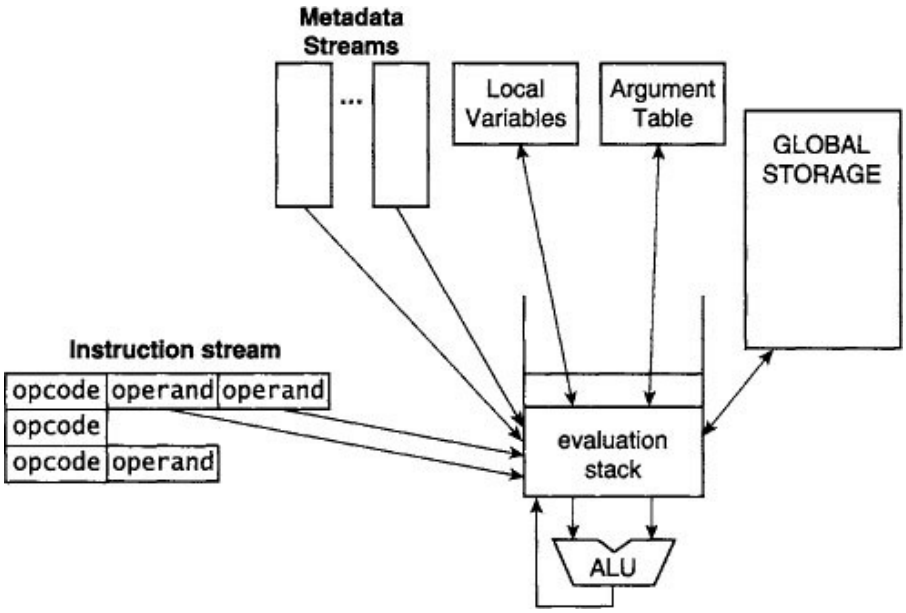


Figure 5.18 MSIL Memory Architecture.

The instruction set depends on the same stack-tracking feature as Java to enable static type checking. Also, as with Java bytecodes, all verifiable control flow instructions either have constant PC relative offsets or are method call/returns. Both arrays and vectors are intrinsic data types, but there are explicit MSIL instructions only for vectors, which are similar to the one-dimensional Java arrays. The multidimensional MSIL arrays are accessed via standard API routines. It is also possible to create a managed pointer to a particular array element, to avoid repeated address arithmetic and range checking.

A small example of MSIL code, and its Java counterpart, are given in **Figure 5.19**. In this example, the instructions match one to one. One ISA difference is that MSIL has more flexibility in specifying data widths; for example, the .i4 indicates an integer of four bytes. Other possibilities (depending on the instruction) are i1, i2, and i8 (integers) and u1, u2, u4, and u8 (unsigned). Another evident difference is that when performing arithmetic, the Java bytecodes explicitly

specify the type of operation to be performed. For example, in Java there are separate add-integer and add-float instructions. In contrast, in MSIL there is a single generic add instruction, and the particular type can be inferred from the types of operands at the top of the stack. In a sense, putting the type of operation in the opcode as in Java is redundant, because stack tracking permits the inference of the type of operation required. However, implementing this inference as part of an interpreter would slow down interpretation. It is apparently the intention (although not an absolute requirement) that MSIL will not be interpreted but will always be compiled before being executed, e.g., by a JIT compiler. Finally, in Java, items such as field descriptors are all in the constant pool, whereas MSIL uses metadata tokens, and the tokens point to items in one of the several streams contained in a program module.

```
0:  iconst_2          0:  ldc.i4.2
1:  aload_0           1:  ldarg.0
2:  getfield  #2      2:  ldobj     <token>
5:  iconst_0          5:  ldc.i4.0
6:  iaload            6:  ldelem.i4
7:  aload_0           7:  ldarg.0
8:  getfield  #2      8:  ldobj     <token>
11: iconst_1          11: ldc.i4.1
12: iaload            12: ldelem.i4
13: iadd              13: add
14: imul              14: mul
15: ireturn           15: ret
      (a)                   (b)
```

Figure 5.19 Example of (a) Java Code and (b) Corresponding MSIL Code.

If they are not specifically designed to do so, conventional programming languages such as C and C++ do not run as verified code in CLI. In C and C++ programs, memory can be allocated and accessed without the restrictions of Java or C# programs. Blocks of memory of unspecified type can be allocated, and pointers can be manipulated as data; i.e., arbitrary arithmetic, logical, and shift instructions can be performed on pointers, and they can be freely copied. To support C-like memory operations (and control flow), MSIL contains a number of unverifiable instructions.

Unmanaged pointers are simply represented as 32-bit integers on the native platform. With unverifiable code, it is possible to allocate and access C-style blocks of memory. The local block-allocate instruction (localloc) creates a block of untyped memory; a copy-block instruction (cpblk) copies a block of memory from a location indicated by one unmanaged pointer to an area indicated by a second unmanaged pointer. The initialize-block (initblk) instruction initializes all the elements of a memory block to a specified value. Because the pointers to the blocks of memory are unmanaged, they can be used in much the same way as in C.

With regard to control flow, the managed code branch instructions are adequate for both verifiable and unverifiable code. For procedures calls, there is a call-indirect (calli) instruction that uses an unmanaged pointer for identifying the called procedure.

5.5.4 Isolation and AppDomains

A final feature of the CLI of interest is its support for isolated, independent applications running on the same VM. This feature is probably most useful for increasing overall system efficiency when multiple application programs are running at the same time; this would be the common mode of operation in a server, for example. As a matter of security, these application programs should be isolated from each other, because they typically perform functions for a number of different users. A straightforward way of implementing isolation is to give each program its own virtual machine, which, in turn is supported as a user-level process by the host platform's operating system. This method is illustrated in **Figure 5.20**a. It is a rather heavy-weight resource-intensive approach, however, with lots of unneeded replication at both the VM level and the host process level.

A possible solution in Java is to implement isolation via the class loader system, by giving each application a different namespace

(see Section 6.1). However, class loaders, as it turns out, do not provide absolute separation, and this has resulted in security holes (McGraw and Felten 1999).

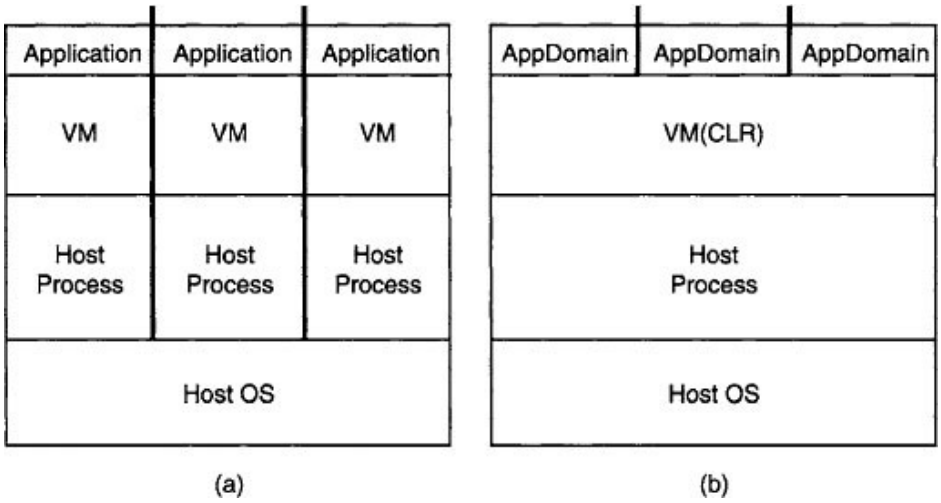


Figure 5.20 Supporting a Number of Identical Applications. *(a) By running each on its own virtual machine; (b) by using AppDomains.*

To solve the isolation problem in an efficient way, the CLI supports AppDomains (application domains), which provide the desired lightweight isolation. A number of independent processes can share the same virtual machine, with complete isolation (see Figure 5.20b). Hence, only one VM and one host platform are required, thereby maximizing the system resources available to the programs doing the work. Note that each of the processes may contain multiple threads, as is always the case with CLI processes.

5.6 Summary: Virtual ISA Features

Process VMs targeted at conventional ISAs are in a sense backward looking, in that they are attempting to provide compatibility with platforms developed years (often decades) in the past. In contrast, The HLL virtual ISAs (e.g., Java bytecodes and the Microsoft CIL) are forward looking; they are developed with support for evolving and future applications as the goal. When coupled with libraries, the VM

platform provides flexible support for the object-oriented programming paradigm, secure network computing, and component-based program development.

To summarize this chapter, we compare the features of the virtual ISAs with those of conventional ISAs. This summary is centered on a number of issues identified in the preceding chapters, where process VMs are constructed and conventional ISAs are virtualized. This comparison provides additional insight regarding the features of existing virtual ISAs and may provide direction for developing future ones.

5.6.1 Metadata

A conventional ISA does not have metadata. The compiler uses information regarding declared data structures at the time it generates binary code and then throws it away; all data-structure information becomes implicit in the binary code. In a V-ISA, data-structure information is maintained along with the program's binary code. Then at load time and at run time, this metadata information enables type-safe code verification, which is a key part of enforcing security. And in the case of the CLI, it enables close interoperability of code generated from different programming languages. The metadata information also has potential for improving emulation performance by providing the underlying emulation engine with data-related information that would be very difficult to discover from binary code alone.

5.6.2 Memory Architecture

In a conventional ISA, logical memory is a fixed size, with addresses that are a feature of the ISA (and can be made known to a running program). The address space may be flat and linear, or it may be segmented; but if it is segmented, the segments are often of fixed size. This feature can lead to a "fit" problem when emulating a guest

ISA on a host platform that has a smaller logical memory size. There can also be fit problems even if the memory sizes are the same because in most process VMs, the VM software must share an address space with a guest application process.

Furthermore, in a conventional memory architecture, the ISA makes specific addresses visible to the user program. For example, if a sbrk() system call is performed in a UNIX system (to get more memory), a specific user-readable address pointer is returned. Furthermore, a user process can change the protections on the basis of specific memory addresses, at page-size granularity, for example, with the mmap() system call. This page-size granularity leads to process VM implementation problems, for example, if the guest page size is smaller than the host page size.

These problems do not occur in the common virtual ISAs because memory architecture is more abstract and is of indefinite size. The basic approach is to allocate memory regions (e.g., objects or stack frames) based on a process's logical requirements and to provide a reference to the base of the memory region when it is allocated. The stack and heap are of undefined size, and the actual contents of the stack pointer or an object reference are not made available to a user process because the V-ISA has no provisions for doing so. Memory accesses can be made relative to the stack pointer and object references, but the actual values they contain cannot be explicitly read or written. This restricts the V-ISA-level process to having only a logical view of memory, and real memory locations are irrelevant.

Although the architected memory space (for stack or heap) is of indefinite size, the actual implementation resources must be finite. Consequently, it is possible for a process to ask for more memory space resources than are available. When this happens, there is an exception that informs the user process this has occurred, and the process can, perhaps, take corrective action. This is not a large limitation, however, because many modern high-level programming

tation, however, because many modern high-level programming languages that work with stacks and heap objects have this limitation anyway. What is being done here is that the idea of a memory space of indefinite size is being moved down to the V-ISA level; e.g., even a V-ISA "assembly language" programmer must deal with an indefinite memory space.

5.6.3 Memory Address Formation

In a conventional ISA, address computation is essentially unrestricted. That is, addresses can be generated using any of the available instructions, so arbitrary addresses to any part of memory can be generated for use by virtually any load or store. This leads to a number of problems with respect to process VM implementations. One problem is that the regions of memory used by the VM software (the runtime) are difficult to protect from the emulated guest program (see Section 3.4.3). This becomes especially problematic if guest registers are memory mapped.

The solution used in the common V-ISAs is to prevent arbitrary address arithmetic for loads and stores. The first part of this solution is to force all addressing to take place via explicit memory pointers (references). In addition, performing arbitrary arithmetic on references is prohibited. This is done by having a special reference type and by restricting the operations that can be performed on a reference. Finally, if a reference is known to access only a given structure (an object), then the declared structure and properties of the object can be used to validate an address. Depending on the availability of information, the check can be performed statically at compile (translation) time or at run time.

5.6.4 Precise Exceptions

With a conventional ISA, the entire process state at the time of a trap or interrupt must be precise, and many instructions may trap.

Furthermore, it is often the case that global mask bits can enable or disable traps and interrupts, and these mask bits potentially can change over the course of a program's execution. As we saw in **Chapters 3** and **4**, the requirement for precise traps can lead to complications and/or performance loss when a VM is implemented.

In the case of V-ISAs, there are typically fewer instructions that can cause exceptions, and the requirement to test for exceptions is encoded into the program and cannot be changed via mask bits. Hence, the decision about whether an exception must be checked can be based on locally available information; it does not depend on global mask bits. Furthermore, the requirements for precise exceptions are somewhat relaxed. For example, in Java the state of the operand stack does not have to be precise (in fact the operand stack is discarded when an exception is thrown). Furthermore, if an exception handler is not local to a method, then none of the local variables held in the frame must be precise after an exception is thrown and the method's frame is discarded.

5.6.5 Instruction Set Features

With regard to conventional ISA features that can facilitate (or complicate) emulation, the two major ones are the register set and condition codes. It is awkward to emulate a guest register set having more registers than the host register set. To binary translate such a guest instruction set, a register context area has to be defined in the host memory, and register load/stores have to be performed to move data between the host registers and the register context area. Besides the data-movement overhead involved, protecting the register context area becomes a problem, as noted earlier.

Of course, the smallest possible register set has no registers at all; local values and operands can be maintained on a stack rather then being held in registers. It is for this reason (as well as code density) that most of the common V-ISAs are stack oriented. With respect to

condition codes, the bigger problems occur when emulating an ISA with condition codes on a host platform that does not support condition codes. Hence, it is preferable to avoid condition codes in ISAs, as in current V-ISAs.

5.6.6 Instruction Discovery

With arbitrary ISAs, code discovery is problematic (see **Section 2.6**). The root cause is indirect jumps to potentially arbitrary locations. When combined with variable-length instructions and embedded data, it is hard (or impossible) to discover code that goes beyond an indirect jump. Clearly, the solution is to restrict indirect jumps. In particular, in V-ISAs they are restricted to be explicit procedure (or method) calls and returns where the return addresses are protected from modifications. Then, because all conditional branches are to PC-relative addresses where the offset is a constant, control flow is statically discoverable. V-ISAs also separate data from code, although they do permit variable-length instructions. However, because all control flow is exposed and explicit and there is no embedded data, variable-length instructions by themselves do not pose a problem. The net result is that the virtual ISAs are explicitly designed so that at the time a method is first invoked, all the code contained in the method can be discovered immediately.

5.6.7 Self-Modifying and Self-Referencing Code

For most user applications, self-modifying and self-referencing code are not essential; in fact they are typically discouraged. Consequently, self-modifying and self-referencing code are most easily dealt with by simply making them impossible to do; such operations are not embodied in the commonly used V-ISAs. Interestingly, and somewhat ironically, the emulation engine running on the host platform may rely heavily on self-modifying code in order to give good performance. For example, writing translated

(or JIT-compiled) code into a code cache is a form of self-modifying code.

5.6.8 Operating System Dependences

As discussed in earlier chapters, when performing process emulation, it is guest operating system dependencies that are probably more difficult to deal with than instruction set dependencies. All instruction sets contain some elemental instructions that make them functionally complete—any function can be performed; it's just a matter of how efficiently it will be done. With operating systems, however, this may not be the case. Different systems may perform process management, file I/O, networking, graphics, etc. differently.

Hence, in HLL VMs, the preferred way to deal with OS dependencies is to arrive at some kind of least common denominator set of functions and to implement them with standard libraries. These standard libraries must be ported to all desired host platforms. This approach is more easily said than done, however, and on some platforms the fit may not be perfect. However, the library interface is usually at a higher level than a conventional OS interface, so the library writer has more flexibility in achieving at least an acceptable level of compatibility.

---

1. However, Sun (McGahn and O'Connor 1998) and ARM (**ARM 2002**) have developed processors that support the Java ISA.

2. Binary classes are often referred to as *class files,* but they do not, strictly speaking, have to be stored as files; hence, the more generic term *binary classes.*

3. However, system methods written in Java do not have to be trusted, strictly speaking; they can be verified just as any other method. Doing so would tend to increase system integrity, but it would also slow down the JVM start-up time, so it is not commonly done.

4. In the context of HLL VMs, translation is usually referred to as *compilation.*

[1] In the context of HLL VMs, translation is usually referred to as *compilation*.

Previous chapter
Chapter 4: Dynamic Binary Optimization

Next chapter
Chapter 6: High-Level Language Virtual Machine Implementation

Chapter Six

High-Level Language Virtual Machine Implementation

The collection of binary classes or modules that make up a program specify its operation in terms of the HLL VM *architecture*. It is up to the HLL VM *implementation* to carry out the specified operations. This chapter discusses HLL VM implementations, primarily by describing the Java VM, the best known of the HLL VMs. A CLI implementation is similar in many respects. We describe the major implementation components and then look at ways of improving performance in HLL VM implementations. We complete the chapter with a case study of a specific high-performance JVM, the Jikes research VM.

The organization of a typical JVM implementation (**Venners 1998**) is shown in **Figure 6.1**. The three major components are the class loader subsystem, the memory system, including a garbage-collected heap, and the emulation engine (sometimes referred to as the *execution engine*). These major components are, in turn, implemented with lower-level components. We first describe the major components briefly and then provide greater detail.

Memory and State Registers

The memory consists of an area for program code, a global memory area, and stacks for both the architected Java code and for native library code. The program area is implicitly defined for the most part; common sense tells us there must be some region for holding the program. However, program memory can only be accessed via the program counter; it is never explicitly accessed with load and store

instructions. Furthermore, the program counter contents can never be directly inspected.

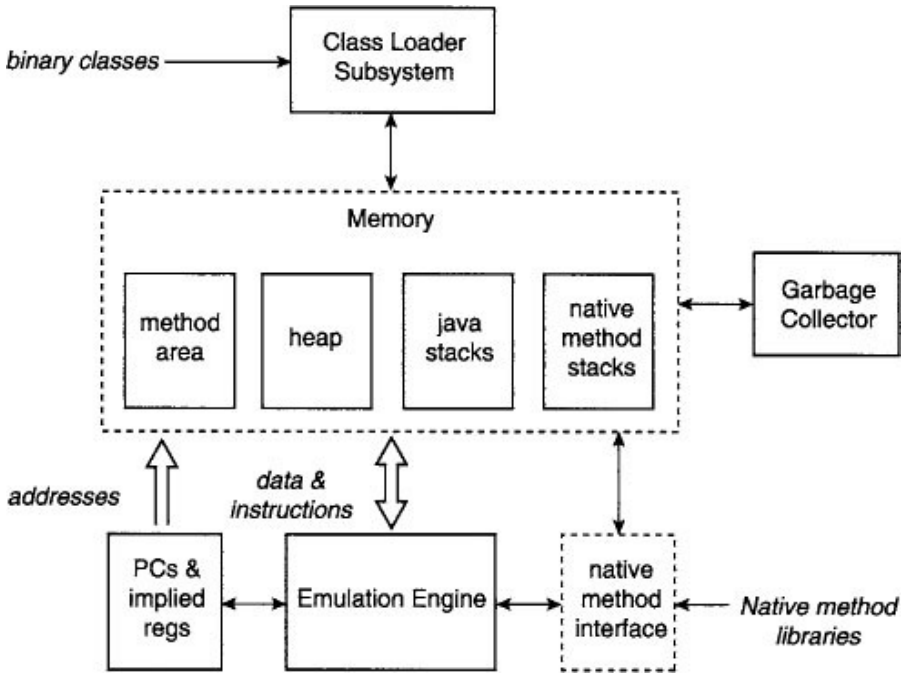


Figure 6.1 A Java Virtual Machine Implementation.

The Java stack area was described in **Section 5.3.2** and needs no further explanation. The native stack and its interaction with the Java stack are described subsequently as part of the native method interface. Associated with the stack is an implied register, the stack pointer. Just as with the program counter, the stack pointer is used by instructions for accessing memory, but it is not explicitly accessible; i.e., there is no way for a program to inspect the contents of the stack pointer. The exact size of the stack is not architected; that is, it is implementation dependent. If its size is ever exceeded by a running program, the StackOverflowError exception is thrown.

The global memory is a dynamically allocated heap. When a new object instance is created, its memory is allocated from the heap. As with the stack, the heap is of unspecified size. If the heap runs out of space, an OutOfMemoryError is thrown. To reduce the likelihood that this will happen, a garbage collector can be used to reclaim heap memory no longer needed by a running program

heap memory no longer needed by a running program.

Garbage Collector

Many Java objects are created, used, and then are no longer needed by the program. This occurs when the last (or only) reference to an object is destroyed or is overwritten. At that point, the object becomes *garbage,* because it is no longer useful to the running program. Because any real JVM implementation will have a limited amount of memory, it is advantageous to reclaim, or collect, the memory space used by the garbage objects so that the physical memory resources can be reused. Consequently, most JVM implementations have a garbage collector, which is responsible for finding those objects no longer needed in order to make room for other, new objects.

Emulation Engine

The emulation engine, supported by the native method interface and implied registers (i.e., the program counter and the stack pointer), is responsible for emulating the Java bytecode instructions. It can be a simple interpreter, or it can perform a full or partial translation (compilation) to native host instructions. Higher-performance implementations may use profiling and staged emulation techniques to first detect hotspots and then translate to native instructions held in a code cache, as discussed in earlier chapters. Even if a simple interpreter is used, some pretranslation of the program file is performed. For example, some of the indirection in the program file (such as the use of the constant pool) would likely be removed, with immediate values being embedded directly in instructions.

Native Method Interface

To access operating system–managed functions, the JVM uses a set of standard libraries. These libraries can perform file I/O and graph-

of standard libraries. These libraries can perform file I/O and graphics operations, for example. Many of the libraries are written in Java, and their associated binary classes are loaded and emulated in essentially the same way as the application-specific methods. However, in practice, at least some of the library code is written in the host's native code in order to bridge the gap between the platform-independent and platform-dependent parts of the overall system. Many calls to the host OS are made through native methods, for example. Providing these native methods (along with other standard library methods) is part of the overall JVM implementation process.

Class Loader Subsystem

The class loader subsystem performs a number of vital functions for supporting a networked computing environment. Of course, it converts the class file containing metadata and instructions into an implementation-dependent memory image. It is also responsible for finding the binary classes in the first place, often dynamically and on demand, either from the system or from other systems in a network. The loader subsystem is responsible for verifying correctness and consistency of binary classes and is an integral part of maintaining overall security within the network environment. We discuss class loading in more detail in the next section.

6.1 Dynamic Class Loading

When a method is called for the first time by a program, the class loader locates the requested binary class; typically it is held in a file. Then it checks the integrity of the binary class and performs any translation of code and metadata in order to make the requested method ready to run.

An important aspect of the network environment is that it must be possible to identify (name) variables, methods, and other data items

in a standard, universal way. All variables and methods must be declared as part of a class (which must have a name). Each compiled class is stored as a separate entity, e.g., in a separate file, and one or more classes can be combined into a logical entity known as a *package.* Thus every method or variable has what is called a *fully qualified name,* which consists of the package name, the class name, and the method or variable name, separated by periods, for example, testpackage.block.mass. For accessing packages across the Internet, there is a naming convention that is based on Internet domain names. For example, a package might be given the following fully qualified name:

edu.wisc.ece.jes.testpackage.shape.area

Here, the Internet domain name **ece.wisc.edu** (in reverse order, by convention) is prefixed to an internal path name, jes.testpackage, to provide a full package name. The access rights to packages, classes, and fields are defined in the following way.

- A package is accessible according to the access rights on the local system, for example, if the read or execute permission allows public access when it is being accessed by an external (nonlocal) user.
- Classes within a package are accessible to all other classes within the same package (this scoping property is one of the reasons for having packages in the first place).
- A class that is declared as public is accessible from other packages; a nonpublic class is only accessible from within its own package.
- All fields of a class are accessible from within its own class. Fields are accessible from a different class in the same package if they are not private.

The class loader system implements dynamic loading and is a critical part of the security system. A Java VM implementation must contain a *primordial class loader,* whose operation is defined as part of the JVM specification. In addition, users may also define class loaders as part of a class loader subsystem. The primordial class loader can be relied upon to do the right thing with respect to security; i.e., it is a trusted JVM component.

As just noted, there can be multiple user-defined class loaders, but each one defines a separate *namespace.* A loaded class is "tagged" with the name of the loader that loaded it, so if two classes loaded by different loaders happen to have the same name, they are in different namespaces and can be kept separate. Classes loaded into one namespace cannot interact with classes in other namespaces; indeed they cannot even become aware of the presence of the other namespaces. Hence, in effect, there is a barrier between the different namespaces. Different class loaders are typically used to load classes from different origins; hence, the namespace becomes identified with the origin.

Unlike the primordial class loader, which must be designed to be trustworthy, the user-supplied class loaders are only as trusted as the user who supplies them. This issue can be simplified somewhat because a user-supplied loader can be designed so that it relies on the primordial class loader for assistance in loading binary classes, for example, as a means for ensuring security.

After it locates a binary class, the loader parses it and translates it into internal data structures for emulation by the execution engine. This involves converting the metadata contained in the binary class into an implementation-dependent form that is more amenable to emulation. As part of this process the loader performs some consistency checks. It first checks the magic number at the beginning of the binary class to make sure the given data is at least claimed to be a binary class (this check will mostly turn up simple errors where the VM is being invoked for a "wrong" file). More importantly, the loader checks to make sure all the components are of the sizes indicated in the binary class and that proper formats are used in the various metadata structures. It can also check to make sure the numbers and types of arguments match between calling and called methods. Within the binary class, fully qualified references are resolved. This is often done on demand. The fully qualified references

are symbolic. After being resolved, the fully qualified (symbolic) reference is replaced with a direct reference.

Another important function of the loader is to verify the integrity of the bytecode program. That is, it checks to make sure the stack values can be tracked statically, as required of all well-formed programs. Then it performs all the static type checking to make sure there are no program errors and that the protection boundaries will be respected by the program. This process is described in more detail in the next subsection. The loader checks to make sure all references in the constant pool are within the bounds of the programs, and it checks to make sure all branch instructions are to addresses within the procedure where they reside. Finally, after the program is verified as being properly formed, memory is initialized and control passed to the emulation engine.

6.2 Implementing Security

The basic protection sandbox was described in general terms in **Section 5.2.1**. This is the default security model used in Java, although the model has been greatly enhanced in the Java 2 specification. In this section we first elaborate on the basic security model and then describe the Java 2 enhancements.

In the sandbox model, the overall objective is to form a barrier around the Java execution environment so that a user application is confined to operate within the bounds of the sandbox and cannot affect any other system or network resources, whether unintentionally or maliciously. As discussed in **Section 5.2.1**, there are three main parts to the sandbox.

First, remote files must be protected. As pointed out earlier, the protection of remote files is within the province of the remote system. Any files on the remote system are given access privileges by the

owner of the file, and, ultimately, these are maintained by the remote OS, whether Windows, Linux, or any other OS. These are the same privileges as for any type of remote access, whether inside or outside the Java execution framework.

Second, local files must be protected from a running Java program. That is, the owner of the local files, typically the user running the Java program, must control access to the files. This part of the protection sandbox is implemented via the security manager, described in more detail in **Section 6.2.2**.

Third, the JVM code and data must be protected from the running Java program. That is, the Java program should not be allowed to read or write data associated with the Java VM side tables, for example. Furthermore, the program should not be allowed to branch or jump to arbitrary locations in the JVM. This third part of the protection sandbox is implemented via a combination of static and dynamic checking, with static checking performed by the loader playing a key role in this process. The next subsection discusses this intraprocess protection part of the sandbox, and this is followed by the discussion of the security manager.

6.2.1 Intraprocess Protection

One of the more challenging aspects of building a modern network-oriented HLL VM is the ability to take untrusted program code from the network, combine it with trusted VM code to form a single host-supported user process, and then execute the untrusted program efficiently. A basic objective is to make sure the application code only accesses its own memory locations and can interact with the JVM (and the rest of the host platform) only via method calls to libraries containing locally trusted code.

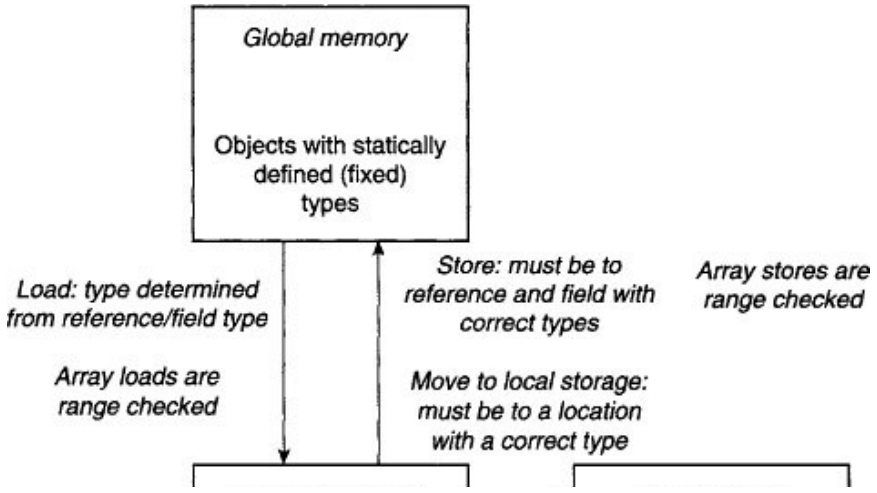
It is the ability to construct a protection sandbox that most clearly separates the modern HLL VMs from the earlier P-code-like VMs. To

separates the modern HLL VMs from the earlier P-code like VMs. To understand this, consider the typical P-code environment. Here, the compiler is a trusted element; i.e., it is assumed by the user to be correctly implemented. The Pascal language is designed so that the compiler can check data-type information at compile time, based on the data declarations made by the programmer. After this checking is complete, the P-code is generated. Then the static data-type information is no longer needed; rather, it is built into the code that accesses the data, and the user can be assured that the P-code will only access data in the correct way. The P-code alone can be passed to other users to run on their computer systems, thereby achieving platform independence. It is up to the individual users to trust the provider of the P-code program.

The earlier scenario would be fine if other users were always willing to trust the source of their programs, but of course this level of trust is often not present in the networked computing environment. Yet a user would like to be able to execute programs that are made widely available for network-based computing. A solution to the problem of untrusted code is to transmit not only the application code but also a checkable description of the data structures on which the code will operate, i.e., the metadata. Then when the program is loaded by the end user, the (trusted) loader will perform static checks of the program with the declared metadata information. If the program code is consistent with the metadata, then the user can be assured that the program will only access its data in a proper (protected) way. Furthermore, the loader can check the control flow information of the program to make sure it branches only to locations within the program itself or performs procedure calls using proper protocols. This is the reason that HLL ISAs stress static type checking and statically traceable control flow information. And, of course, the ability to perform this high degree of static checking derives from the strong typing features incorporated in the high-level language in which the program was written in the first place (e.g., Java or C#).

Now we will illustrate in more detail the features of the Java ISA that allows intraprocess protection. There are two main elements of intraprocess protection. First, it must be guaranteed that the program only accesses its own memory; i.e., it can never make data accesses outside the bounds of its declared memory region. Second, it must be guaranteed that the program will not branch outside its own code region. It can only leave the code region via a library call, and the libraries are trusted code.

We first give an informal inductive argument for the protection of data accesses. Just as data flow is centered on the stack (see **Figure 5.9**), our argument will revolve around the stack (**Figure 6.2**). Before any memory accesses have been performed, and with an empty operand stack, the only way to put a reference onto the operand stack is from the constant pool or from local memory. The constant-pool entries are statically checked by the loader, as noted earlier, so they can be guaranteed to be valid references (or are null). All local memory locations holding reference types are initialized to null references. Furthermore, all movement between the stack and local memory uses absolute addresses for the local memory locations. Hence, the loader can check the local memory types with the data-movement opcodes to make sure that only reference types are placed into local memory reference locations.



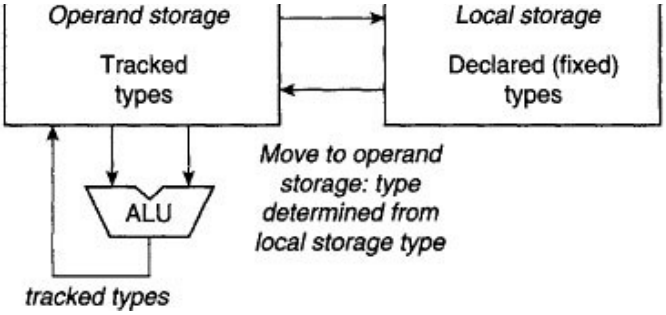


Figure 6.2 Data Flow and Intraprocess Protection Checking.

Once on the stack, the types of all elements can be statically tracked (as described near the end of **Section 5.3.3**). In particular, recall that for any valid program, the number and types of elements on the stack can always be determined from the static code itself. The loader verifies this is the case at the time the program is loaded. Among other things, this means that any data movement of references from the stack to local memory can be checked for the correct types statically.

Pulling all the foregoing together, starting from initial conditions, before any global memory accesses have been made, any reference values on the operand stack either are null or are verified reference values taken from the constant pool (either directly or indirectly through local memory). Furthermore, the types of these references are statically known.

Next, consider accesses to global memory. The first access to global memory is to a location specified by a reference on the operand stack. Because of the ability to track reference types on the stack and in local memory, this reference type is known statically. Hence, if an object is accessed, the field information for the access can also be checked statically (there is an exception for arrays, given in the next paragraph). If the access is a load, the data type being loaded is known statically; if it is a reference type, then the type is also known and can be tracked statically while it is on the stack or in local memory. Similarly, if a value (data or reference) is stored to an object field memory, it can also be checked prior to being stored because of

field memory, it can also be checked prior to being stored because of static tracking in local storage and static field definitions; hence the value stored to memory will be consistent with the static field information. By a similar argument, and to complete the induction step, if all data types can be determined from static information prior to a memory access *i;* then the next memory access, $i + 1$, must also yield statically trackable type information.

An important exception to the argument just given involves array accesses. All loads or stores to arrays must use an array reference; these can also be statically checked for type information. However, arrays differ from other objects in that the exact element being accessed may be computed at run time. This means that the actual index into an array must be checked dynamically to make sure it is within the bounds of the array. Furthermore, all references, whether for an array or for a general object, must be checked for null values, in the event that they have not been assigned to an object when they are first used.

Another important place where dynamic checking is required occurs with dynamic casting or conversion of an object reference to a different type. Allowing arbitrary casting of references can break that part of the protection sandbox enabled by static checking, because static checking depends on knowledge of object types. However, casting can be done in situations where the reference is being cast to a superclass (*upcasting*) or a subclass (*downcasting*). The safety of upcasting can be checked at compile time; however, downcasting requires a run-time check to make sure the reference being cast is to a subclass (or the same class) as the object type to which it currently refers.

To summarize, a program, even if it is from an untrusted source, can be checked to make sure it makes no unallowed memory accesses. This is done by a combination of static (load-time) and dynamic (run-time) checks. In Java, much of the checking is done stati-

cally. By relying on static type checking, dynamic checking is mostly confined to null pointer checking, bounds checking for array accesses, and dynamic casts to make sure they are being properly done within the class hierarchy. In the high-performance Java discussion to be presented later, we will include analysis methods and optimizations that are able to remove some of these dynamic checks.

Finally, consider the protection provided for control transfers, i.e., the constraint that there never be a branch or jump that leaves the program's valid code region. Control transfers are only through branches, switch statements, and method calls. All the branches and switch statements are to PC relative constant values. Therefore, all the branches and jumps can be statically checked to make sure they branch to instructions only within the given procedure. The only way to exit a method is through a method call/return (or when the program terminates). Therefore, if all methods are internally checked, the overall program is checked.

6.2.2 Security Enforcement

In a Java implementation, the security manager is a class belonging to the java.lang API that contains a number of methods for checking to make sure potentially unsafe operations cannot be performed. These operations include reading a specified file, writing a specified file, opening a socket connection to a specified host and port number, and creating a new process, among many others. Such operations are made available to untrusted user programs via methods in (trusted) Java libraries, e.g., the java.io API, that interact directly with the host OS. Before performing a requested action involving an OS call, the trusted library methods first check with the security manager, via the appropriate method call, to make sure it should be allowed.

A security manager check method simply checks the requested op-

eration to see if it is permitted and either throws a security exception if it is not permitted or returns if it is. The security manager is attached when a Java application is initiated and cannot be changed, deleted, or replaced with another thereafter. Hence, an individual user can specify the checks made via the security manager, i.e., which files should be accessed and how, which network ports can be used, etc. Then whenever a Java application is started, these checks will be in effect as long as the application runs.

If the user chooses to use no security manager, then the application has unrestricted access to the user's resources. Of course, the Java application cannot exceed the privileges of the user, because the JVM runs as a user process. Any attempt to go outside the user's resources will be caught by the underlying OS on the host platform.

The security manager cannot protect against everything, overallocating memory or spawning too many threads, for example. In situations like this, it is difficult or impossible to discern the difference between an erroneous or malicious program and one that has large resource requirements. For example, how could one determine the difference between runaway recursion and recursion that is simply very deep? In effect, detecting the difference would be equivalent to solving the classic Turing Machine halting problem. Consequently, some denial of-service attacks are possible, despite the presence of a security manager.

The security manager can be written to implement relatively complex policies. The loaders and security manager may be customized by a software developer or user. For example, the security manager can check to see if a request is made from a locally provided method or from one loaded over the network and can allow different resource accesses depending on which is the case.

6.2.3 Enhanced Security Model

The basic security sandbox was extended by adding signing in JDK 1.1 and then enhanced with access control in Java 2 (**McGraw and Felten 1999**). Adding the notion of identity and signing gets away from the all-or-nothing sandbox approach and allows more flexible, finer-grained security policies, where code from different sources may be granted different access privileges to user resources.

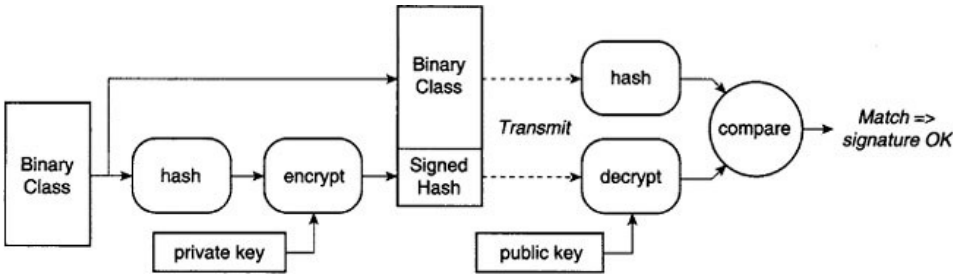


Figure 6.3 Using a Public Key Encryption System to Sign a Binary Class File.

If particular outside program sources can be securely identified, then a wide variety of security policies can be implemented, each one depending on the specific source of a given application program. A technology that supports secure identification of sources is *signing*. The basic idea of signing is simple: if a piece of code brought in from the outside can be signed in a nonforgeable way, then the local security system can inspect the signature and provide access privileges appropriate for the given (signed) piece of code.

Signing is based on public key encryption systems (**Diffie and Hellman 1976**). In a public key encryption system there is public key/private key pair. A message is encrypted with the public key and is decrypted with the private key. The critical feature of the system is that the private key cannot be determined from the public key in any practical way. Consequently, only the owner of the private key can decrypt a message.

The application of public key encryption technology to signing application code (and other types of data) is illustrated in **Figure 6.3**. The source of application code (a binary class) first hashes the code

down to a smaller size (for reasons of overall efficiency) and then encrypts the hashed version with the private key. The hashed, encrypted version is a signature appended to the binary class and is sent across a network to someone who wishes to execute it. At the receiving end, the received binary class is again hashed (with the same hash function as before), and the signature is decrypted. The two versions of the hashed code are then compared. If they are the same, then the user is assured that the owner (sender) of the binary class is as claimed. The security manager can then grant access to local resources in accordance with a policy established by the local user.

Because the identity of a program's source can be securely identified, the application of the all-or-nothing sandbox for all program sources is no longer necessary. Fine-grained, configurable security policies can be implemented. For example, programs from one outside source may be allowed to open network connections, while others cannot. One program may be given access only to file A, while another may be given access only to file B.

This flexible approach does pose some potential problems when binary classes from different sources are allowed to interoperate as part of the same program. For example, what if a method belonging to a binary class from a source not having access privileges to a given file calls a method from a source that does have access privileges to the file and then performs a file access indirectly through the called method? The security manager can solve this problem by inspecting the method call stack to check on access privileges not only of a method making a request but of all earlier methods in the call sequence (**Wallach and Felten 1998**).

Stack inspection is illustrated in **Figure 6.4**. Each stack frame is appended with *principal* information indicating whether the method came from a trusted system source or an untrusted source. Also, there is information regarding the access privileges that reside with

the source of the method. After a sequence of method calls, the resulting stack frames are shown in the figure. Untrusted method 2 had permissions to write file A but is restricted from accessing other files. On the other hand, method 4 does have permission to write to file B. Now, when method 4 attempts a write to file B, it will do so via the I/O API method 5 that first calls a security manager check method. The security manager walks the stack in reverse order, checking access permissions of all the methods that it finds on the stack. When it comes to method 2, it finds that permission to write file B should not be granted and throws an exception.

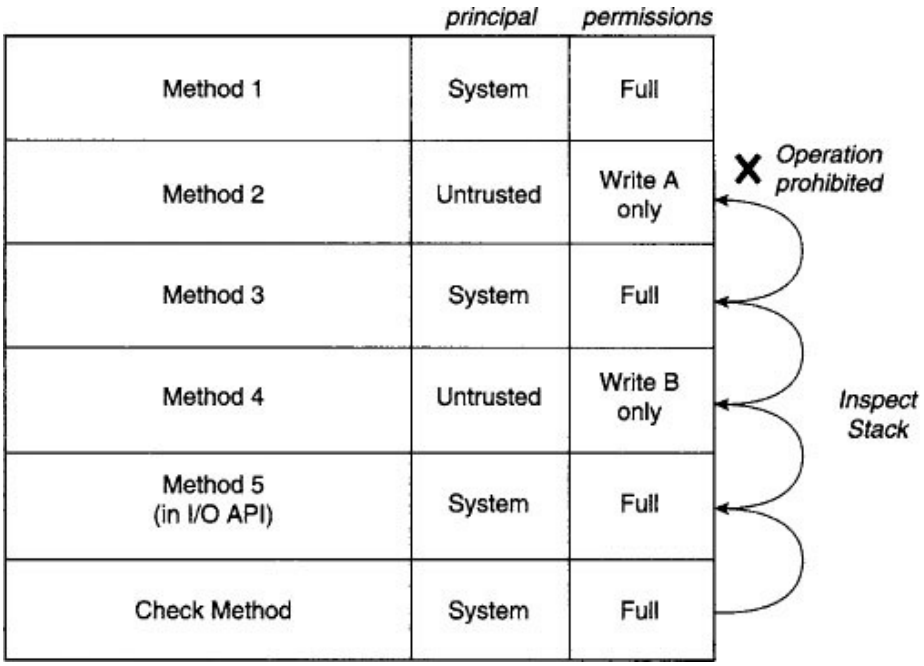


Figure 6.4 Method Call Stack Extended with Resource Access Permissions. *The security manager inspects the stack to enforce protection.*

6.3 Garbage Collection

In an object-oriented programming environment, objects can be freely created, used, and later discarded when they are no longer needed. The programmer is given the illusion of an unbounded memory space and does not have to worry about explicitly allocating and managing a fixed-size memory. Of course, in reality, mem-

ory resources are not unbounded, and eventually a program can run out of memory resources (which would result in an OutOfMemory exception being thrown). To prevent this from happening (or at least forestall it), objects that are no longer accessible (i.e., "garbage") can be collected and then reused for new objects.

In our simple example in **Figure 5.5**, when the reference a is assigned to a new Rectangle object, the prior Rectangle object it pointed to becomes inaccessible. This is the typical way that an object becomes garbage—the last (or only) reference to it is overwritten with a reference to a different object of the same type. In other cases, the last reference may simply be discarded, e.g., by popping it from the stack.

A garbage collector is part of essentially every JVM implementation, although it is not strictly required as part of the JVM specification. When the JVM is running low on memory resources or at periodic intervals, the garbage collector is invoked to find the inaccessible garbage objects and collect their memory resources for later use. A typical situation is illustrated in **Figure 6.5**. Here, a *root set* of references point to objects held in the global memory heap. Some of these objects, in turn, contain references that point to other objects, and so on. Also, there are some objects that cannot be reached through a sequence of references beginning with the root set; these are the garbage objects.

To begin a garbage collection operation, it is first necessary to identify the root pointers. By referring back to **Figure 5.7** (and considering the Java instruction set) we can see that the root set must contain references somewhere on the stack, including both local storage and the operand stack, or in the constant pool. The root set must also include references contained in static objects. Any access to global memory, i.e., through a getfield or putfield instruction, must get its reference from one of these places, and any object that cannot be accessed via a series of references beginning with some

not be accessed via a series of references beginning with some member of the root set simply cannot be reached. Hence, garbage collection finds all the objects that can be accessed, beginning with the root set of references; any objects not found during this process are garbage that can be collected. There are a number of approaches for performing these operations (**Jones 1996**; **Wilson 1992**; **Appel 1991**), and in the following subsections we summarize some of the major classes of garbage-collection techniques.

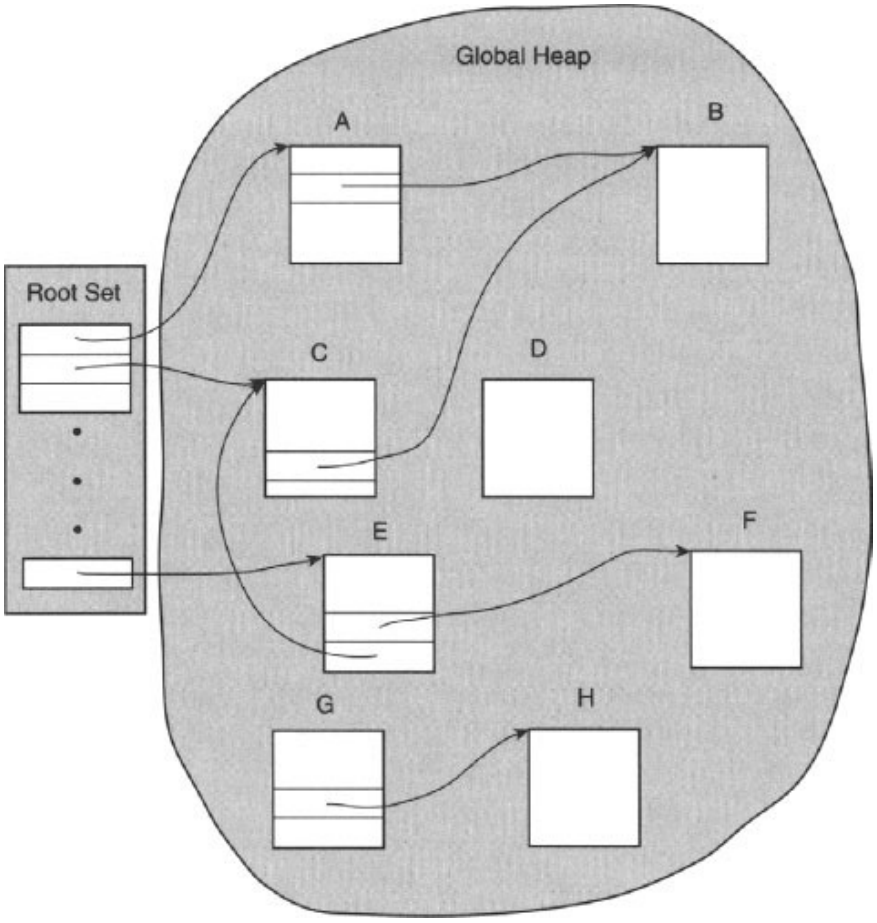


Figure 6.5 A Garbage-Collected Heap. *The objects G, D, and H are garbage.*

During the discussion, a number of tradeoffs are considered. Time spent collecting garbage is time not spent computing, so the overhead of garbage collection is an important overall implementation consideration. Moreover, different methods organize the heap differently, collect the garbage into free space differently, and may implement object references differently. Consequently, the important

tradeoffs to be considered involve garbage-collection time, object-allocation time, object-access time, and the efficiency with which the heap space is used.

Although there have been garbage collectors that keep a count of the number of references to each object on the heap and consider as garbage any object whose reference count goes to zero (**Collins 1960**), such reference counting collectors are relatively uncommon in JVMs. Most JVMs use the general method suggested earlier, that is, to start with a set of root references and then *trace* through chains of references in the heap to find all the objects that are reachable or are "live." Then all the unreachable objects are considered garbage and are recycled. The following discussion of garbage collectors will focus on tracing collectors.

6.3.1 Mark-and-Sweep Collectors

The mark-and-sweep collector is a basic collector that starts with the root references and traces through all the reachable objects, *mark*ing each one as it is reached. Marking may consist of setting a flag bit contained as part of the object implementation or in a separate bitmap, where each entry in the map is associated with an object. If an already-marked object is reached, then tracing down that particular path stops. After all live objects have been found and marked, there is a *sweep* stage, where all the objects are inspected and the unmarked ones are determined to be garbage and can be reused. As garbage objects are found during the sweep stage, they can be combined into a linked list of free objects.

Overall, this is a relatively fast way of identifying and collecting garbage. However, the free objects are of varying size and are scattered around the heap space, interspersed with the live objects. Simply linking together free objects leads to a memory fragmentation problem that in turn leads to a significant inefficiency when a new object is created and a suitably sized free space must be found

new object is created and a suitably sized free space must be found for it. Specifically, the allocation algorithm must search the linked list to find an appropriately sized block of contiguous free memory). Eventually fragmentation becomes so great that compaction is required.

Inefficiencies can be reduced, however, by using segregated free lists. That is, by dividing the heap into a set of fixed-size chunks having a range of sizes, e.g., from 16 bytes to 2KB, and maintaining the free space as multiple linked lists, one for each chunk size (**Comfort 1964**). Then the object allocator can simply go to a free list and get a chunk of the appropriate size (i.e., the smallest one that is large enough to hold the object being allocated). Provisions must be made for occasionally rebalancing the space in each size category, but overall allocation efficiency is significantly improved. In general, dynamic memory allocation is very closely associated with garbage collection. An excellent survey of memory allocation techniques, including segregated free lists, is **Wilson et al. (1995)**.

A conceptually straightforward approach to improving allocation efficiency is to consolidate all the garbage space into a large contiguous region from which new objects can be created. There are two ways of consolidating the garbage space: by compacting and by copying. These are described in the next two subsections.

6.3.2 Compacting Collectors

A compacting collector, as its name suggests, essentially “slides” the live objects to the bottom (or top) of the heap memory area so that all live objects are adjacent. What is left is a contiguous region of free space. For example, **Figure 6.6** shows the contents of memory with live objects (shaded in dark gray) and garbage objects (shaded in light gray). During compaction, the live objects are moved to a contiguous region of the heap (at the bottom in this example), and the unused space also becomes a contiguous region, from which

new objects can be allocated.

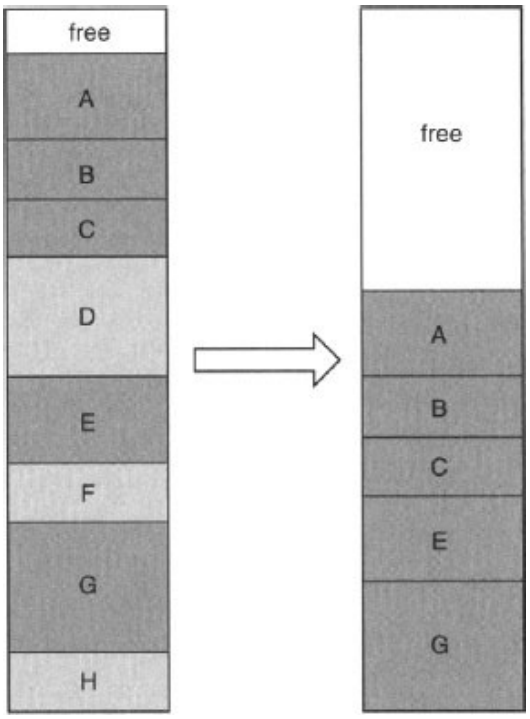


Figure 6.6 Example of Garbage Compaction.

Although conceptually simple, a compacting collector is relatively slow, in that it makes multiple passes through the heap. One pass does the marking, and then subsequent passes compute the new locations for the live objects, move the objects, and update all references to point to the new locations. Other methods, to be summarized shortly, improve efficiency by reducing the number of passes and/or analyzing only a subset of the heap during each collection step.

Compacting collection also highlights an issue that occurs with any of the schemes that move objects in memory as a part of garbage collection. That is, the references to the objects must be changed when objects are moved, and this complicates (and slows) the overall process. However, to reduce the number of reference updates, some systems have used a *handle pool* for consolidating pointers to each of the individual objects. Then a reference to an object points to the associated pointer in the handle pool. This is illustrated in

Figure 6.7. With the handle pool, it is not necessary to find and update all the references to an object when the object is moved; rather, only the pointer in the handle pool has to be changed, and all references are then automatically modified in the process. The big disadvantage of this approach is that every object access includes an additional level of indirection.

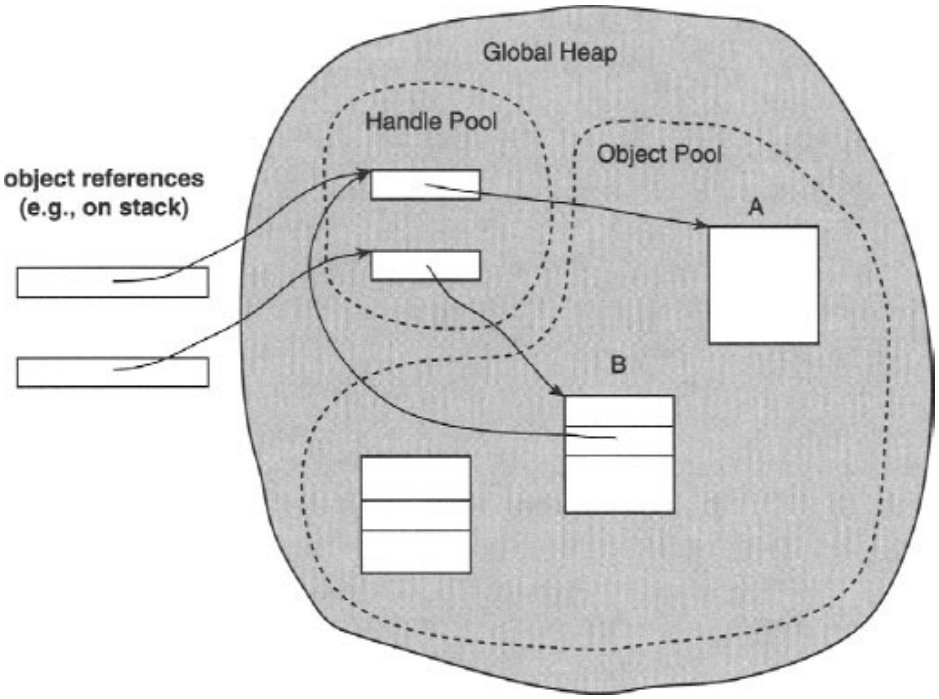


Figure 6.7 Handle Pool. *Using a level of indirection (through a handle) can simplify pointer updates when an object is moved during garbage collection.*

6.3.3 Copying Collectors

To reduce the number of passes through the heap during collection, a copying collector trades memory space for collection time. It does this by dividing the heap into two halves. At any point in time, one half is unused while the other contains the active heap. When the active half fills up, the collector makes a pass through the heap just as it does during a mark phase; however, it combines the sweep with the mark phase. When it finds a live object, it immediately moves it into the unused half of the heap and continues the pass through the heap. When the pass through the heap is complete, the

live objects are in contiguous locations in what was formerly the unused half of the heap, the remainder of the second half is free, and the first half of the heap (which was formerly active) becomes unused. A copying collector is illustrated in **Figure 6.8**. A copying collector is faster than a compacting collector, but it also has higher memory requirements because half of the heap space, by definition, is unused at any given time.

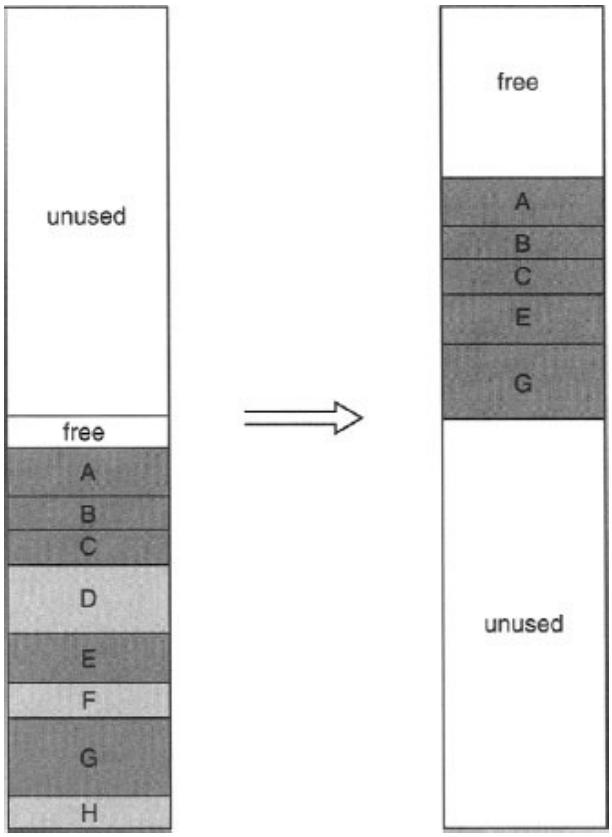


Figure 6.8 Example of Garbage Collection via Copying.

6.3.4 Generational Collectors

Both compacting and copying collectors move a very large fraction of dynamic objects every time collection is performed. A long-lived object may be moved many times during its lifetime; this rather wasteful movement can be avoided by observing that object lifetimes have a bimodal distribution. First, many objects have very short lifetimes; this is often a by-product of good object-oriented

programming practice. Second, objects that do not have short lifetimes tend to have very long lifetimes. Consequently, generational garbage collectors attempt to group objects according to their age in order to avoid repeated copying of the long-lived objects.

In a basic generational collector, the heap is divided into subheaps. For simplicity, we describe the implementation with two subheaps, but generalization to more than two subheaps is straightforward. Of the two subheaps, one is intended to hold the older, or *tenured,* objects. Meanwhile, the other subheap serves as a *nursery* for newly created objects. The nursery is garbage-collected much more frequently than the tenured subheap. If an object survives a certain (usually small) number of collections in the nursery, it is moved to the tenured subheap. Consequently, the longer-lived objects eventually are placed in the tenured section of the heap, where the garbage collections are infrequent, thus avoiding unnecessary object movement.

Not only is the total overhead for collecting reduced, but a smaller portion of the heap is collected each time the collector is invoked. This means that if the running process is stopped while collection takes place, the “pause” time is much reduced. With a large heap and a conventional compacting or copying collector, the time that a program is stopped for collection can be noticeable to the user.

The two subheaps of a generational collector do not necessarily have to be managed the same way. In fact, they can be managed differently to take advantage of performance tradeoffs. A specific hybrid algorithm studied as part of the Jikes JVM project (**Attanasio et al. 2001**) uses a copying collector for the nursery, which yields fast allocation of new objects, and a mark-and-sweep collector for the tenured objects, which reduces collection time by eliminating pointer updates.

6.3.5 Incremental and Concurrent Collectors

6.3.3 Incremental and Concurrent Collectors

All of the basic collectors described earlier stop program execution while they perform collection and then return control to the program. Collection can be time consuming (even if generational collection is used), so program execution may pause for a significant time period while the collector is working. Collection time may be spread out if collection is done incrementally as the program runs rather than all at once. Furthermore, in real-time applications, the garbage-collection time may be limited to provide adequate response latencies. If multiple processors are available, it may be advantageous to collect garbage concurrently with one thread while normal program execution proceeds using other threads. In both cases, a partially collected heap may be in a state of flux while the program is running. This implies there must be some synchronization between the running program and the garbage collector, so the program does not attempt to reference an object at the time it is being moved; i.e., at a time when pointers may be temporarily inconsistent. Or the program may be changing references at the time the collector is tracing a reference path, leading to similar synchronization problems.

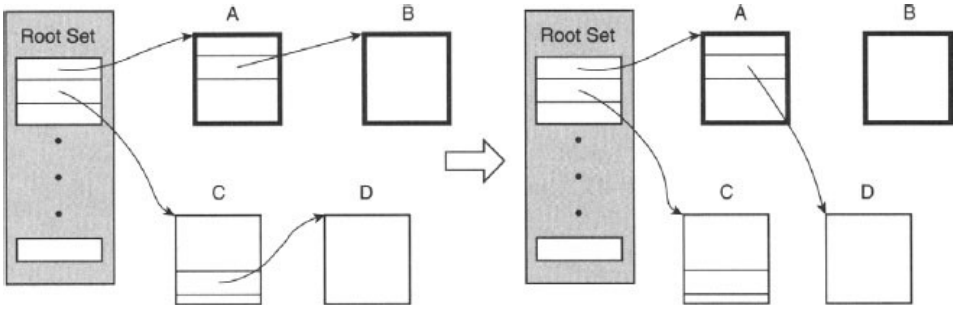


Figure 6.9 A Problem with Concurrent Garbage Collection. *A reference in an object may be modified after the object has been marked.*

Many of the conventional stop-and-collect methods can be converted to incremental versions. A basic problem with concurrent or incremental collections is that at any given time, an object may have been scanned and marked. Then while collection is still in progress, a reference in the already-scanned object may be changed to point

a reference in the already-scanned object may be changed to point to an object that has not yet been marked. An example is in **Figure 6.9**. In this example, the objects A and B have been marked (as indicated by the bold boxes). The other objects have not yet been marked. Then before they can be marked, the pointer to object B is replaced with a pointer to object D. If this is the only pointer to D, then D may never be marked and will be incorrectly discarded as garbage. The object B may also be incorrectly retained, but this is not a problem; it will be collected on the next garbage-collection pass.

There are a number of solutions to this problem; they all provide some form of synchronization between the running application (which may change the contents of the heap) and the collector. One of the common solutions is to provide *write barriers* for references to objects that have already been marked. These write barriers basically check for the case where a pointer in an already-marked object is overwritten. When this occurs, the object pointed to is marked (and put on the queue of objects whose pointers should be followed as marking proceeds).

6.3.6 Discovering the Root Set

As stated at the beginning of this section, before beginning garbage collection it is necessary to identify the root pointers. The root set for Java programs (and similarly for CLI) consists of references held in the stack, the constant pool, and contained in static objects. However, this is what one might call the “architected” root set. In a particular VM implementation, the actual pointers may reside in registers and various implementation-specific runtime memory locations. For example, the architected stack elements may be assigned to registers by the dynamic compiler, and some of these could be spilled to memory.

Therefore, at the time garbage collection is invoked, it is necessary

to construct the architected root set from the implementation storage locations that may potentially hold root set pointers. The basic solution for finding the root set is for the compilation/optimization system to keep side tables, or maps, that indicate where the architected root set elements may be found, if needed for garbage collection. This solution requires that the VM runtime and compilers keep track of the architected root values as dynamic compilation and optimization are performed (**Stichnoth, Lueh, and Cuerniak 1999**).

The foregoing approach is referred to as *type accurate* or *exact* because it uses the exact root set as a starting point. A less conservative solution is to take all the implementation registers and memory locations that can *possibly* hold root pointers and assume they *do* hold root pointers (**Boehm and Weiser 1988**). These will form a superset of the true root set. Some of these may be eliminated from consideration (e.g., if they hold small integer values that are clearly not memory addresses). Then the garbage collector begins with this root superset. This approach frees the compilation/optimization system from keeping track of root set references and saves runtime side table space for keeping track of the mappings at all potential garbage-collection points. A problem with this superset approach, however, is that a garbage collector that moves objects, e.g., a compacting collector, cannot be used, because a region of memory may be identified as an object of a given type when it is not really an object. Then if such an apparent "object" is moved, it will damage any real objects that contain the apparent object as part of their memory region.

6.3.7 Garbage-Collection Summary

No one collector works best across all programs because programs vary in working set size, object sizes, heap sizes, and the rate at which objects are created (and freed). Hence, we cannot arrive at hard-and-fast conclusions regarding the "best" collector (as the hundreds of papers on the topic attest). We can summarize the impor-

tant performance tradeoffs in general terms, however. **Table 6.1** qualitatively compares the major collector categories in terms of (1) collection time, (2) object allocation time, (3) object access time, and (4) memory efficiency (i.e., the relative total heap space that can maintain a working set of a given size). In addition to these criteria, all the "moving" collectors, i.e., the compacting and copying collectors, require an exact root set, as noted earlier.

The use of a generational collector is a somewhat orthogonal consideration because a generational collector can be used with any of the basic approaches (or as a hybrid). The advantage of a generational collector is that it reduces collection time by focusing only on those objects where garbage is most likely to be found (the nursery). Today, generational collection is usually considered to be a winning strategy. A study of garbage collection in the Jikes JVM (**Attanasio et al. 2001**) describes the performance tradeoffs with respect to specific benchmark programs.

TABLE 6.1 Comparison of Basic Garbage Collectors

Collector	Collection Time	Object Allocation Time	Access Time	Memory Efficiency	Comments
Mark and sweep	Good	Poor	Good	Medium	Allocation requires search for proper-size free memory block
Mark and sweep (multisizes)	Good	Medium	Good	Medium	Multisizes help allocation time more than it benefits memory efficiency
Compacting	Poor	Good	Good	Good	Collection requires multiple passes
Copying (with handles)	Medium	Good	Poor	Poor	Memory efficiency poor due to unused space
Copying (without handles)	Medium to poor	Good	Good	Poor	Pointers must be found and updated for all moved objects

6.4 Java Native Interface

The Java native interface (JNI), as described in Section 5.3.6, is a way of allowing Java programs and native compiled programs to interoperate. An important consideration when implementing the JNI is

the accessing of Java data structures in a safe way, i.e., in a way that maintains the integrity of Java objects and the stack.

From the perspective of a Java program, a call to a native method is in some respects similar to executing a single (very complex) instruction. There are arguments and a return value, but the Java stack is not used by the called method; rather, the native methods use their own platform-dependent stack. The management of this native stack must be done by trusted JVM code. The JVM implementation supports both a Java stack and a native stack, with one native stack per Java thread (see **Figure 6.10**). When there is a call to a native method, it is intercepted by the JVM, which then sets up the native stack frame, transfers arguments, and then transfers control to the called method. Similarly, on a return, it places a result onto the Java stack frame and returns control to emulation. In the JNI, a native method may "call back" a Java method; when this happens, a Java stack frame is placed on the stack frame of the calling native method.

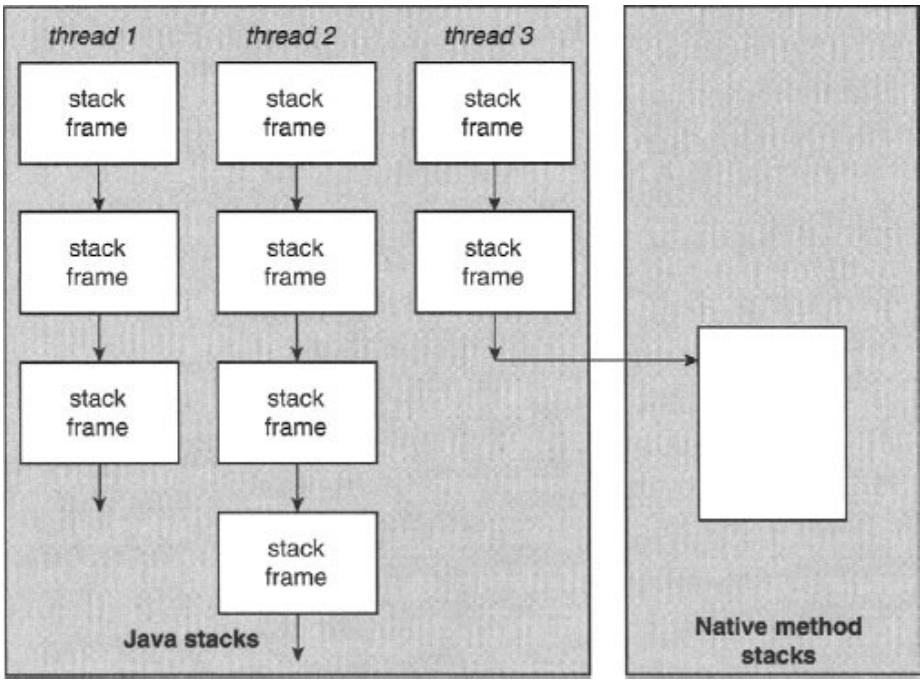


Figure 6.10 Java and Native Stacks. *In a VM implementation, the native library stack is managed separately from the architected VM stack.*

A native method can be given access to objects and arrays via the JNI. However, once the native method has an array or object reference, the array or object must be protected from garbage collection. That is, the native method assumes the object to be at a particular memory location, and an object-moving garbage collector would violate this assumption. Even with a nonmoving collector, if the native method should happen to have the only reference at some point, then the garbage collector may attempt to collect it (because the reference will not be in the garbage collector's normal root set). To prevent this from happening, the JNI method that passes the native method a reference also "pins" the referenced object so that it will not be moved or collected. With a non-moving collector, the JVM could also add the reference to the root set.

6.5 Basic Emulation

The emulation engine in a JVM can be implemented in a number of ways, with different complexities and performance levels. The simplest method uses straightforward interpretation of bytecode instructions. In general, this is no different than interpretation of a conventional ISA as described in detail in **Chapter 2**. A more advanced, and commonly used, emulation method performs just-in-time (JIT) compilation (**Tabatabai et al. 1998**; **Aycock 2003**). With a JIT compiler, methods are compiled at the time they are first invoked, i.e., "just in time" for execution. This compilation essentially performs a translation from the bytecode instructions to native host instructions. Method-at-a-time JIT compilation is enabled because, in contrast to a conventional ISA, the Java ISA is designed so that all the instructions in a method can easily be discovered at the time the method is first entered.

A JIT compiler bears a very close relationship to the binary translators discussed earlier in this book and could just as easily be called a