OpenJDK
Installing
Contributing
Sponsoring
Developers' Guide
Vulnerabilities
JDK GA/EA Builds
Mailing lists
Wiki · IRC
Mastodon
Bluesky
Bylaws · Census
Legal
Workshop
JEP Process
Source code
GitHub
Mercurial
Tools
Git
jtreg harness
Groups
(overview)
Adoption
Build
Client Libraries
Compatibility & Specification Review
Compiler
Conformance
Core Libraries
Governing Board
HotSpot
IDE Tooling & Support
Internationalization
JMX
Members
Networking
Porters
Quality
Security
Serviceability
Vulnerability
Web
Projects
(overview, archive)
Amber
Babylon
CRaC
Code Tools
Coin
Common VM Interface
Developers' Guide
Device I/O
Duke
Galahad
Graal
IcedTea
JDK 7
JDK 8
JDK 8 Updates
JDK 9
JDK (..., 23, 24, 25)
JDK Updates
Jigsaw
Kona
Kulla
Lanai
Leyden
Lilliput
Locale Enhancement
Loom

HotSpot Runtime Overview

This section introduces key concepts associated with the major subsystems of the HotSpot runtime system. The following topics are covered:

Command-Line Argument Processing

There are a number of command-line options and environment variables that can affect the performance characteristics of the Java HotSpot Virtual Machine. Some of these options are consumed by the launcher (such as '`-server`' or '`-client`'), some are processed by the launcher and passed to the JVM, while most are consumed directly by the JVM.

There are three main categories of options: standard options, non-standard options, and developer options. Standard options are expected to be accepted by all JVM implementations and are stable between releases (though they can be deprecated). Options that begin with `-X` are non-standard (not guaranteed to be supported on all JVM implementations), and are subject to change without notice in subsequent releases of the Java SDK. Options that begin with `-XX` are developer options and often have specific system requirements for correct operation and may require privileged access to system configuration parameters; they are not recommended for casual use. These options are also subject to change without notice.

Command-line flags control the values of internal variables in the JVM, all of which have a type and a default value. For boolean values, the mere presence or lack of presence of a flag on the command-line can control the value of the variables. For

Memory Model
Update
Metropolis
Mission Control
Multi-Language VM
Nashorn
New I/O
OpenJFX
Panama
Penrose
Port: AArch32
Port: AArch64
Port: BSD
Port: Haiku
Port: Mac OS X
Port: MIPS
Port: Mobile
Port: PowerPC/AIX
Port: RISC-V
Port: s390x
SCTP
Shenandoah
Skara
Sumatra
Tsan
Valhalla
Verona
VisualVM
Wakefield
Zero
ZGC

–XX boolean flags, a ‘+’ or '–' prefix before the name indicates a true or false value, respectively. For variables that require additional data, there are a number of different mechanisms used to pass that data in. Some flags accept the data passed in directly after the name of the flag without any delineator, while for other flags you have to separate the flag name from the data with a ‘:’ or a ‘=’ character. Unfortunately the method depends on the particular flag and its parsing mechanism. Developer flags (the –XX flags) appear in only three different forms: –XX:+*OptionName*, –XX:–*OptionName*, and –XX:*OptionName*=.

Most all of the options that take an integer size value will accept ‘k’, ‘m’, or ‘g’ suffixes which are used a kilo-, mega-, or giga- multipliers for the number. These are most often used for arguments that control memory sizes.

VM Lifecycle

The following sections gives an overview of the general purpose java launcher pertaining to the lifecyle of the HotSpot VM.

Launcher

There are several HotSpot VM launchers in the Java Standard Edition, the general purpose launcher typically used is the java command on Unix and on Windows java and javaw commands, not to be confused with javaws which is a network based launcher.

The launcher operations pertaining to VM startup are:

1. Parse the command line options, some of the command line options are consumed by the launcher itself, for example `-client` or `-server` is used to determine and load the appropriate VM library, others are passed to the VM using `JavaVMInitArgs`.
2. Establish the heap sizes and the compiler type (client or server) if these options are not explicitly specified on the command line.
3. Establish the environment variables such as `LD_LIBRARY_PATH` and `CLASSPATH`.
4. If the java Main-Class is not specified on the command line, fetch the Main-Class name from the JAR's manifest.
5. Create the VM using `JNI_CreateJavaVM` in a newly created thread (non primordial thread). Note: creating the VM in the primordial thread greatly reduces the ability to customize the VM, for example the stack size on Windows, and many other limitations
6. Once the VM is created and initialized, the Main-Class is loaded, and the launcher gets the main method's attributes from the Main-Class.
7. The java main method is then invoked in the VM using `CallStaticVoidMethod`, using the marshalled arguments from the command line.
8. Once the java main method completes, its very important to check and clear any pending exceptions that may have occurred and also pass back the exit status, the exception is cleared by calling `ExceptionOccurred`, the return value of this method is 0 if successful, any other value otherwise, this value is passed back to the calling process.
9. The main thread is detached using `DetachCurrentThread`, by doing so we decrement the thread count so the `DestroyJavaVM` can be called safely, also to ensure that the thread is not performing operations in the vm and that there are no active java frames on its stack.

The most important phases are the `JNI_CreateJavaVM` and `DestroyJavaVM` these are described in the next sections.

JNI_CreateJavaVM

The JNI invocation method performs the following:

1. Ensures that no two threads call this method at the same time and that no two VM instances are created in the same process. Noting that a VM cannot be created in the same process space once a point in initialization is reached, “point of no return”. This is so because the VM creates static data structures that cannot be re-initialized, at this time.

2. Checks to make sure the JNI version is supported, and the ostream is initialized for gc logging. The OS modules are initialized such as the random number generator, the current pid, high-resolution time, memory page sizes, and the guard pages.

3. The arguments and properties passed in are parsed and stored away for later use. The standard java system properties are initialized.

4. The OS modules are further created and initialized, based on the parsed arguments and properties, are initialized for synchronization, stack, memory, and safepoint pages. At this time other libraries such as libzip, libhpi, libjava, libthread are loaded, signal handlers are initialized and set, and the thread library is initialized.

5. The output stream logger is initialized. Any agent libraries (hprof, jdi) required are initialized and started.

6. The thread states and the thread local storage (TLS), which holds several thread specific data required for the operation of threads, are initialized.

7. The global data is initialized as part of the I phase, such as event log, OS synchronization primitives, perfMemory (performance memory), chunkPool (memory allocator).

8. At this point, we can create `Threads`. The Java version of the main thread is created and attached to the current OS thread. However this thread will not be yet added to the known list of the `Threads`. The Java level synchronization is initialized and enabled.

9. The rest of the global modules are initialized such as the `BootClassLoader`, `CodeCache`, `Interpreter`, `Compiler`, JNI, `SystemDictionary`, and `Universe`. Noting that, we have reached our “point of no return”, ie. we can no longer create another VM in the same process address space.

10. The main thread is added to the list, by first locking the `Thread_Lock`. The Universe, a set of required global data structures, is sanity checked. The `VMThread`, which performs all the VM's critical functions, is created. At this point the appropriate JVMTI events are posted to notify the current state.

11. The following classes `java.lang.String`, `java.lang.System`, `java.lang.Thread`, `java.lang.ThreadGroup`, `java.lang.reflect.Method`, `java.lang.ref.Finalizer`, `java.lang.Class`, and the rest of the `System` classes, are loaded

and initialized. At this point, the VM is initialized and operational, but not yet fully functional.

12. The Signal Handler thread is started, the compilers are initialized and the CompileBroker thread is started. The other helper threads StatSampler and WatcherThreads are started, at this time the VM is fully functional, the `JNIEnv` is populated and returned to the caller, and the VM is ready to service new JNI requests.

DestroyJavaVM

This method can be called from the launcher to tear down the VM, it can also be called by the VM itself when a very serious error occurs.

The tear down of the VM takes the following steps:

1. Wait until we are the last non-daemon thread to execute, noting that the VM is still functional.

2. Call `java.lang.Shutdown.shutdown()`, which will invoke Java level shutdown hooks, run finalizers if finalization-on-exit.

2. Call `before_exit()`, prepare for VM exit, run VM level shutdown hooks (they are registered through `JVM_OnExit()`), stop the `Profiler`, `StatSampler`, `Watcher` and `GC` threads. Post the status events to JVMTI/PI, disable JVMPI, and stop the Signal thread.

3. Call `JavaThread::exit()`, to release JNI handle blocks, remove stack guard pages, and remove this thread from Threads list. From this point on we cannot execute any more Java code.

4. Stop VM thread, it will bring the remaining VM to a safepoint and stop the compiler threads. At a safepoint, care should that we should not use anything that could get blocked by a Safepoint.

5. Disable tracing at JNI/JVM/JVMPI barriers.

6. Set `_vm_exited` flag for threads that are still running native code.

7. Delete this thread.

8. Call `exit_globals()`, which deletes IO and `PerfMemory` resources.

9. Return to caller.

VM Class Loading

The Java Hotspot VM supports class loading as defined by the Java Language Specification, Third Edition [1], the Java Virtual Machine Specification (JVMS), Second Edition [2] and as amended by the updated JVMS chapter 5, Loading, Linking and Initializing [3].

The VM is responsible for resolving constant pool symbols, which requires loading, linking and then initializing classes and interfaces. We will use the term “class loading” to describe the overall process of mapping a class or interface name to a class object, and the more specific terms loading, linking and initializing for the phases of class loading as defined by the JVMS.

The most common reason for class loading is during bytecode resolution, when a constant pool symbol in the classfile requires resolution. Java APIs such as `Class.forName()`, `classLoader.loadClass()`, reflection APIs, and `JNI_FindClass` can initiate class loading. The VM itself can initiate class loading. The VM loads core classes such as `java.lang.Object`, `java.lang.Thread`, etc. at JVM startup. Loading a class requires loading all superclasses and superinterfaces. And classfile verification, which is part of the linking phase, can require loading additional classes.

The VM and Java SE class loading libraries share the responsibility for class loading. The VM performs constant pool resolution, linking and initialization for classes and interfaces. The loading phase is a cooperative effort between the VM and specific class loaders (`java.lang.classLoader`).

Class Loading Phases

The load class phase takes a class or interface name, finds the binary in classfile format, defines the class and creates the `java.lang.Class` object. The load class phase can throw a `NoClassDefFound` error if a binary representation can not be found. In addition, the load class phase does format checking on the syntax of the classfile, which can throw a `ClassFormatError` or `UnsupportedClassVersionError`. Prior to completing loading of a class, the VM must load all of its superclasses and superinterfaces. If the class hierarchy has a problem such that this class is its own superclass or superinterface (recursively), then the VM will throw a `ClassCircularityError`. The VM also throws `IncompatibleClassChangeError` if the direct superinterface is not an interface, or the direct superclass is an interface.

The link class phase first does verification, which checks the classfile semantics, checks the constant pool symbols and does type checking. These checks can throw a `VerifyError`. Linking then does preparation, which creates and initializes static fields to standard defaults and allocates method tables. Note that no Java code has yet been run. Linking then optionally does resolution of symbolic references.

Class initialization runs the static initializers, and initializers for static fields. This is the first Java code which runs for this class. Note that class initialization requires superclass initialization, although not superinterface initialization.

The JVMS specifies that class initialization occurs on the first “active use” of a class. The JLS allows flexibility in when the symbolic resolution step of linking occurs as long as we respect the semantics of the language, finish each step of loading, linking and initializing before performing the next step, and throw errors when programs would expect them. For performance, the HotSpot VM generally waits until class initialization to load and link a class. So if class `A` references class `B`, loading class `A` will not necessarily cause loading of class `B` (unless required for verification). Execution of the first instruction that references `B` will cause initialization of `B`, which requires loading and linking of class `B`.

Class Loader Delegation

When a class loader is asked to find and load a class, it can ask another class loader to do the actual loading. This is called class loader delegation. The first loader is an initiating loader, and the class loading that ultimately defines the class is called the defining loader. In the case of bytecode resolution, the initiating loader is the class loader for the class whose constant pool symbol we are resolving.

Class loaders are defined hierarchically and each class loader has a delegation parent. The delegation defines a search order for binary class representations. The Java SE class loader hierarchy searches the bootstrap class loader, the extension class loader and the system class loader in that order. The system class loader is the default application class loader, which runs "main" and loads classes from the classpath. The application class loader can be a class loader from the Java SE class loader libraries, or it can be provided by an application developer. The Java SE class loader libraries implement the extension class loader which loads classes from the lib/ext directory of the JRE.

Bootstrap Class Loader

The VM implements the bootstrap class loader, which loads classes from the `BOOTPATH`, including for example rt.jar. For faster startup, the VM can also process preloaded classes via Class Data Sharing.

Type Safety

A class or interface name is defined as a fully qualified name which includes the package name. A class type is uniquely determined by that fully qualified name and the class loader. So a class loader defines a namespace, and the same class name loaded by two distinct defining class loaders results in two distinct class types.

Given the existence of custom class loaders, the VM is responsible for ensuring that non-well-behaved class loaders can not violate type safety. See Dynamic Class Loading in the Java Virtual Machine [4], and the JVMS 5.3.4 [2]. The VM ensures that when class `A` calls `B.foo()`, `A`'s class loader and `B`'s class loader agree on `foo`'s parameters and return value, by tracking and checking loader constraints.

Class Metadata in HotSpot

Class loading creates either an `instanceKlass` or an `arrayKlass` in the GC permanent generation. The instanceKlass refers to a java mirror, which is the instance of `java.lang.Class` mirroring this class. The VM C++ access to the `instanceKlass` is via a `klassOop`.

HotSpot Internal Class Loading Data

The HotSpot VM maintains three main hash tables to track class loading. The `SystemDictionary` contains loaded classes, which maps a class name/class loader pair to a `klassOop`. The `SystemDictionary` contains both class name/initiating loader pairs and class name/defining loader pairs. Entries are currently only removed at a safepoint. The `PlaceholderTable` contains classes which are currently being loaded. It is used for `ClassCircularityError` checking and for parallel class loading for class loaders that support multi-threaded classloading. The `LoaderConstraintTable` tracks constraints for type safety checking.

These hash tables are all protected by the `SystemDictionary_lock`. In general the load class phase in the VM is serialized using the Class loader object lock.

Bytecode Verifier and Format Checker

The Java language is a type-safe language, and standard Java compilers produce valid classfiles and type-safe code, but the JVM can't guarantee that the code was produced by a trustworthy compiler, so it must reestablish that type-safety through a process at link-time called bytecode verification.

Bytecode verification is specified in section 4.8 of the Java Virtual Machine Specification. The specification prescribes both static and dynamic constraints on the code which the JVM verifies. If any violations are found, the VM will throw a VerifyError and prevent the class from being linked.

Many of the constraints on the bytecodes can be checked statically, such as the operand of an 'ldc' code must be a valid constant pool index whose type is CONSTANT_Integer, CONSTANT_String or CONSTANT_Float. Other constraints which check the type and number of arguments for other instructions requires dynamic analysis of the code to determine which operands will be present on the expression stack during execution.

There are currently two methods of analyzing the bytecodes to determine the types and number of operands that will be present for each instruction. The traditional method is called "type inference", and operates by performing an abstract interpretation of each bytecode and merging type states at branch targets or exception handles. The analysis iterates over the bytecode until a steady state for the types are found. If a steady state cannot be found, or if the resulting types violate some bytecode constraint, then a VerifyError is thrown. The code for this verification step is present in the libverify.so external library, and uses JNI to gather whatever information is needed about classes and types.

New in JDK6 is the second method for verification which is called "type verification". In this method the Java compiler provides the steady-state type information for each branch or exception target, via the code attribute, StackMapTable. The StackMapTable consists of a number of stack map frames, each which indicates the types of the items on the expression stack and in the local variables at some offset in the method. The JVM needs to then only perform one pass through the bytecode to verify the correctness of the types to verify the bytecode. This is the method already used by JavaME CLDC. Since it it smaller and faster, this method of verification is built directly in the VM itself.

For all classfiles with a version number less than 50, such as those created prior to JDK6, the JVM will use the traditional type inference method to verify the classfiles. For classfiles greater than or equal to

50, the `StackMapTable` attributes will be present and the new verifier will be used. Because of the possibility of older external tools that might instrument the bytecode but neglect to update the `StackMapTable` attribute, certain verification errors that occur during type-checking verification may failover to the type-inference method. Should that pass succeed, the class file will be verified.

Class Data Sharing

Class data sharing (CDS) is a feature introduced in J2SE 5.0 that is intended to reduce the startup time for Java programming language applications, in particular smaller applications, as well as reduce footprint. When the JRE is installed on 32-bit platforms using the Sun provided installer, the installer loads a set of classes from the system jar file into a private internal representation, and dumps that representation to a file, called a “shared archive”. If the Sun JRE installer is not being used, this can be done manually, as explained below. During subsequent JVM invocations, the shared archive is memory-mapped in, saving the cost of loading those classes and allowing much of the JVM's metadata for these classes to be shared among multiple JVM processes.

Class data sharing is supported only with the Java HotSpot Client VM, and only with the serial garbage collector.

The primary motivation for including CDS is the decrease in startup time it provides. CDS produces better results for smaller applications because it eliminates a fixed cost: that of loading certain core classes. The smaller the application relative to the number of core classes it uses, the larger the saved fraction of startup time.

The footprint cost of new JVM instances has been reduced in two ways. First, a portion of the shared archive, currently between five and six megabytes, is mapped read-only and therefore shared among multiple JVM processes. Previously this data was replicated in each JVM instance. Second, since the shared archive contains class data in the form in which the Java Hotspot VM uses it, the memory which would otherwise be required to access the original class information in `rt.jar` is not needed. These savings allow more applications to be run concurrently on the same machine. On Microsoft Windows, the footprint of a process, as measured by various tools, may appear to increase, because a larger number of pages are being mapped in to the process' address space. This is offset by the reduction in the amount of memory (inside Microsoft Windows) which is needed to hold portions on `rt.jar`. Reducing footprint remains a high priority.

In HotSpot, the class data sharing implementation introduces new Spaces into the permanent generation which contain the shared data. The classes.jsa shared archive is memory mapped into these Spaces at VM startup. Subsequently, the shared region is managed by the existing VM memory management subsystem.

Read-only shared data includes constant method objects (`constMethodOops`), symbol objects (`symbolOops`), and arrays of primitives, mostly character arrays.

Read-write shared data consists of mutable method objects (`methodOops`), constant pool objects (`constantPoolOops`), VM internal representation of Java classes and arrays (`instanceKlass`es and `arrayKlass`es), and various `String`, `Class`, and `Exception` objects.

Interpreter

The current HotSpot interpreter, which is used for executing bytecodes, is a template based interpreter. The HotSpot runtime a.k.a. `InterpreterGenerator` generates an interpreter in memory at the startup using the information in the `TemplateTable` (assembly code corresponding to each bytecode). A template is a description of each bytecode. The `TemplateTable` defines all the templates and provides accessor functions to get the template for a given bytecode. The non-product flag `–XX:+PrintInterpreter` can be used to view the template table generated in memory during the VM's startup process.

The template design performs better than a classic switch-statement loop for several reasons. First, the switch statement performs repeated compare operations, and in the worst case it may be required to compare a given command with all but one bytecodes to locate the required one. Second, it uses a separate software stack to pass Java arguments, while the native C stack is used by the VM itself. A number of JVM internal variables, such as the program counter or the stack pointer for a Java thread, are stored in C variables, which are not guaranteed to be always kept in the hardware registers. Management of these software interpreter structures consumes a considerable share of total execution time.[5]

Overall, the gap between the VM and the real machine is significantly narrowed by the HotSpot interpreter, which makes the interpretation speed considerably higher. This, however, comes at a price of e.g. large machine-specific chunks of code (roughly about 10 KLOC (thousand lines of code) of Intel-specific and 14 KLOC of SPARC-specific code). Overall code size and complexity is also significantly higher, since e.g. the code supporting dynamic code generation is needed. Obviously, debugging dynamically generated machine code is significantly more difficult than static code. These properties certainly do not facilitate implementation of runtime evolution, but they don’t make it infeasible either.[5]

The interpreter calls out to the VM runtime for complex operations (basically anything too complicated to do in assembly language) such as constant pool lookup.

The HotSpot interpreter is also a critical part of the overall HotSpot adaptive optimization story. Adaptive optimization solves the problems of JIT compilation by taking advantage of an interesting program property. Virtually all programs spend the vast majority of their time executing a minority of their code. Rather than compiling method by method, just in time, the Java HotSpot VM immediately runs the program using an interpreter, and analyzes the code as it runs to detect the critical hot spots in the program. Then it focuses the attention of a global native-code optimizer on the hot spots. By avoiding compilation of infrequently executed code (most of the program), the Java HotSpot compiler can devote more attention to the performance-critical parts of the program, without necessarily increasing the overall compilation time. This hot spot monitoring is continued dynamically as the program runs, so that it literally adapts its performance on the fly to the user's needs.

Java Exception Handling

Java virtual machines use exceptions to signal that a program has violated the semantic constraints of the Java language. For example, an attempt to index outside the bounds of an array will cause an exception. An exception causes a non-local transfer of control from the point where the exception occurred (or was *thrown*) to a point specified by the programmer (or where the exception is *caught)*. [6]

The HotSpot interpreter, dynamic compilers, and runtime all cooperate to implement exception handling. There are two general cases of exception handling: either the exception is thrown and caught in the same method, or it's caught by a caller. The latter case is more complicated and requires *stack unwinding* to find the appropriate handler.

Exceptions can be initiated by the *throw* bytecode, a return from a VM-internal call, a return from a JNI call, or a return from a Java call. (The last case is really just a later stage of the first 3.) When the VM recognizes that an exception has been thrown, the runtime system is invoked to find the nearest handler for that exception. Three pieces of information are used to find the handler; the current method, the current bytecode, and the exception object. If a handler is not found in the current method, as mentioned above, the current activation stack frame is popped and the process is iteratively repeated for previous frames.

Once the correct handler is found, the VM execution state is updated, and we jump to the handler as Java code execution is resumed.

Synchronization

Broadly, we can define “synchronization” as a mechanism that prevents, avoids or recovers from the inopportune interleavings (commonly called “races”) of concurrent operations. In Java, concurrency is expressed through the thread construct. Mutual exclusion is a special case of synchronization where at most a single thread is permitted access to protected code or data.

HotSpot provides Java monitors by which threads running application code may participate in a mutual exclusion protocol. A monitor is either locked or unlocked, and only one thread may own the monitor at any one time. Only after acquiring ownership of a monitor may a thread enter the critical section protected by the monitor. In Java, critical sections are referred to as "synchronized blocks", and are delineated in code by the `synchronized` statement.

If a thread attempts to lock a monitor and the monitor is in an unlocked state, the thread will immediately gain ownership of the monitor. If a subsequent thread attempts to gain ownership of the monitor while the monitor is locked that thread will not be permitted to proceed into the critical section until the owner releases the lock and the 2nd thread manages to gain (or is granted) exclusive ownership of the lock.

Some additional terminology: to “enter” a monitor means to acquire exclusive ownership of the monitor and enter the associated critical section. Likewise, to “exit” a monitor means to release ownership of the monitor and exit the critical section. We also say that a thread that has locked a monitor now “owns” that monitor. “Uncontended” refers to synchronization operations on an otherwise unowned monitor by only a single thread.

The HotSpot VM incorporates leading-edge techniques for both uncontended and contended synchronization operations which boost synchronization performance by a large factor.

Uncontended synchronization operations, which comprise the majority of synchronizations, are implemented with constant-time techniques. With *biased locking*, in the best case these operations are essentially free of cost. Since most objects are locked by at most one thread during their lifetime, we allow that thread to *bias* an object toward itself. Once biased, that thread can subsequently lock and unlock the object without resorting to expensive atomic instructions.[7]

Contended synchronization operations use advanced adaptive spinning techniques to improve throughput even for applications with significant amounts of lock contention. As a result, synchronization performance becomes so fast that it is not a significant performance issue for the vast majority of real-world programs.

In HotSpot, most synchronization is handled through what we call ””fast-path” code. We have two just-in-time compilers (JITs) and an interpreter, all of which will emit fast-path code. The two JITs are “C1”, which is the `–client` compiler, and “C2”, which is the `–server` compiler. C1 and C2 both emit fast-path code directly at the synchronization site. In the normal case when there's no contention, the synchronization operation will be completed entirely in the fast-path. If, however, we need to block or wake a thread (in monitorenter or monitorexit, respectively), the fast-path code will call into the slow-path. The slow-path implementation is in native C++ code while the fast-path is emitted by the JITs.

Per-object synchronization state is encoded in the first word (the so-called *mark word*) of the VM's object representation. For several states, the mark word is multiplexed to point to additional synchronization metadata. (As an aside, in addition, the mark word is also multiplexed to contain GC age data, and the object's identity hashCode value.) The states are:

- Neutral: Unlocked
- Biased: Locked/Unlocked + Unshared
- Stack-Locked: Locked + Shared but uncontended
  The mark points to displaced mark word on the owner thread's stack.
- Inflated: Locked/Unlocked + Shared and contended
  Threads are blocked in monitorenter or wait().
  The mark points to heavy-weight "objectmonitor" structure.[8]

Thread Management

Thread management covers all aspects of the thread lifecycle, from creation through to termination, and the coordination of threads within the VM. This involves management of threads created from Java code (whether application code or library code), native threads that attach directly to the VM, or internal VM threads created for a range of purposes. While the broader aspects of thread management are platform independent, the details necessarily vary depending on the underlying operating system.

Threading Model

The basic threading model in Hotspot is a 1:1 mapping between Java threads (an instance of `java.lang.Thread`) and native operating system threads. The native thread is created when the Java thread is started, and is reclaimed once it terminates. The operating system is responsible for scheduling all threads and dispatching to any available CPU.

The relationship between Java thread priorities and operating system thread priorities is a complex one that varies across systems. These details are covered later.

Thread Creation and Destruction

There are two basic ways for a thread to be introduced into the VM: execution of Java code that calls start() on a java.lang.Thread object; or attaching an existing native thread to the VM using JNI. Other threads created by the VM for internal purposes are discussed below.

There are a number of objects associated with a given thread in the VM (remembering that Hotspot is written in the C++ object-oriented programming language):

- The java.lang.Thread instance that represents a thread in Java code
- A JavaThread instance that represents the java.lang.Thread instance inside the VM. It contains additional information to track the state of the thread. A JavaThread holds a reference to its associated java.lang.Thread object (as an oop), and the java.lang.Thread object also stores a reference to its JavaThread (as a raw int). A JavaThread also holds a reference to its associated OSThread instance.
- An OSThread instance represents an operating system thread, and contains additional operating-system-level information needed to track thread state. The OSThread then contains a platform specific "handle" to identify the actual thread to the operating system

When a java.lang.Thread is started the VM creates the associated JavaThread and OSThread objects, and ultimately the native thread. After preparing all of the VM state (such as thread-local storage and allocation buffers, synchronization objects and so forth) the native thread is started. The native thread completes initialization and then executes a start-up method that leads to the execution of the java.lang.Thread object's run() method, and then, upon its return, terminates the thread after dealing with any uncaught exceptions, and interacting with the VM to check if termination of this thread requires termination of the whole VM. Thread termination releases all allocated resources, removes the JavaThread from the set of known threads, invokes destructors for the OSThread and JavaThread and ultimately ceases execution when its initial startup method completes.

A native thread attaches to the VM using the JNI call AttachCurrentThread. In response to this an associated OSThread and JavaThread instance is created and basic initialization is performed. Next a java.lang.Thread object must be created for the attached thread, which is done by reflectively invoking the Java code for the Thread class constructor, based on the arguments supplied when the thread attached. Once attached, a thread can invoke whatever Java code it needs to via the other JNI methods available. Finally when the native thread no longer wishes to be involved with the VM it can call the JNI DetachCurrentThread method to disassociate it from the VM (release resources, drop the reference to the java.lang.Thread instance, destruct the JavaThread and OSThread objects and so forth).

A special case of attaching a native thread is the initial creation of the VM via the JNI CreateJavaVM call, which can be done by a native application or by the launcher (`java.c`). This causes a range of initialization operations to take place and then acts effectively as if a call to `AttachCurrentThread` was made. The thread can then invoke Java code as needed, such as reflective invocation of the `main` method of an application. See the JNI section for further details.

Thread States

The VM uses a number of different internal thread states to characterize what each thread is doing. This is necessary both for coordinating the interactions of threads, and for providing useful debugging information if things go wrong. A thread's state transitions as different actions are performed, and these transition points are used to check that it is appropriate for a thread to proceed with the requested action at that point in time – see the discussion of safepoints below.

The main thread states from the VM perspective are as follows:

- `_thread_new`: a new thread in the process of being initialized
- `_thread_in_Java`: a thread that is executing Java code
- `_thread_in_vm`: a thread that is executing inside the VM
- `_thread_blocked`: the thread is blocked for some reason (acquiring a lock, waiting for a condition, sleeping, performing a blocking I/O operation and so forth)

For debugging purposes additional state information is also maintained for reporting by tools, in thread dumps, stack traces etc. This is maintained in the `OSThread` and some of it has fallen into dis-use, but states reported in thread dumps etc include:

- `MONITOR_WAIT`: a thread is waiting to acquire a contended monitor lock
- `CONDVAR_WAIT`: a thread is waiting on an internal condition variable used by the VM (not associated with any Java level object)
- `OBJECT_WAIT`: a thread is performing an `Object.wait()` call

Other subsystems and libraries impose their own state information, such as the JVMTI system and the `ThreadState` exposed by the `java.lang.Thread` class itself. Such information is generally not accessible to, nor relevant to, the management of threads inside the VM.

Internal VM Threads

People are often surprised to discover that even executing a simple “Hello World” program can result in the creation of a dozen or more threads in the system. These arise from a combination of internal VM threads, and library related threads (such as reference handler and finalizer threads). The main kinds of VM threads are as follows:

- VM thread: This singleton instance of `VMThread` is responsible for executing VM operations, which are discussed below
- Periodic task thread: This singleton instance of `WatcherThread` simulates timer interrupts for executing periodic operations within the VM

- GC threads: These threads, of different types, support parallel and concurrent garbage collection
- Compiler threads: These threads perform runtime compilation of bytecode to native code
- Signal dispatcher thread: This thread waits for process directed signals and dispatches them to a Java level signal handling method

All threads are instances of the `Thread` class, and all threads that execute Java code are `JavaThread` instances (a subclass of `Thread`). The VM keeps track of all threads in a linked-list known as the `Threads_list`, and which is protected by the `Threads_lock` – one of the key synchronization locks used within the VM.

VM Operations and Safepoints

The `VMThread` spends its time waiting for operations to appear in the `VMOperationQueue`, and then executing those operations. Typically these operations are passed on to the `VMThread` because they require that the VM reach a *safepoin*t before they can be executed. In simple terms, when the VM is at safepoint all threads inside the VM have been blocked, and any threads executing in native code are prevented from returning to the VM while the safepoint is in progress. This means that the VM operation can be executed knowing that no thread can be in the middle of modifying the Java heap, and all threads are in a state such that their Java stacks are unchanging and can be examined.

The most familiar VM operation is for garbage collection, or more specifically for the “stop-the-world” phase of garbage collection that is common to many garbage collection algorithms. But many other safepoint based VM operations exist, for example: biased locking revocation, thread stack dumps, thread suspension or stopping (i.e. The `java.lang.Thread.stop()` method) and numerous inspection/modification operations requested through JVMTI.

Many VM operations are synchronous, that is the requestor blocks until the operation has completed, but some are asynchronous or concurrent, meaning that the requestor can proceed in parallel with the `VMThread` (assuming no safepoint is initiated of course).

Safepoints are initiated using a cooperative, polling-based mechanism. In simple terms, every so often a thread asks “should I block for a safepoint?”. Asking this question efficiently is not so simple. One place where the question is often asked is during a thread state transition. Not all state transitions do this, for example a thread leaving the VM to go to native code, but many do. The other places where a thread asks are in compiled code when returning from a method or at certain stages during loop iteration. Threads executing interpreted code don't usually ask the question, instead when the safepoint is requested the interpreter switches to a different dispatch table that includes the code to ask the question; when the safepoint is over, the dispatch table is switched back again. Once a safepoint has been requested, the `VMThread` must wait until all threads are known to be in a safepoint-safe state before proceeding to execute the VM operation. During a safepoint the `Threads_lock` is used to block any threads that were running, with the `VMThread` finally releasing the `Threads_lock` after the VM operation has been performed.

C++ Heap Management

In addition to the Java heap, which is maintained by the Java heap manager and garbage collectors, HotSpot also uses the C/C++ heap (also called the malloc heap) for storage of VM-internal objects and data. A set of C++ classes derived from the base class *`Arena`* is used to manage C++ heap operations.

`Arena` and its subclasses provide a fast allocation layer that sits on top of malloc/free. Each `Arena` allocates memory blocks (or *`Chunks`)* out of 3 global *`ChunkPool`s.* Each `ChunkPool` satisfies allocation requests for a distinct range of allocation sizes. For example, a request for 1k of memory will be allocated from the “small” `ChunkPool`, while a 10K allocation will be made from the "medium" `ChunkPool`. This is done to avoid wasteful memory fragmentation.

The `Arena` system also provides better performance than pure malloc/free. The latter operations may require acquisition of global OS locks, which affects scalability and can hurt performance. `Arena`s are thread-local objects which cache a certain amount of storage, so that in the fast-path allocation case a lock is not required. Likewise, `Arena` free operations do not require a lock in the common case.

`Arena`s are used for thread-local resource management (*`ResourceArea`*) and handle management (*`HandleArea`*). They are also used by both the client and server compilers during compilation.

Java Native Interface (JNI)

The JNI is a native programming interface. It allows Java code that runs inside a Java virtual machine to interoperate with applications and libraries written in other programming languages, such as C, C++, and assembly.

While applications can be written entirely in Java, there are situations where Java alone does not meet the needs of an application. Programmers use the JNI to write *Java native methods* to handle those situations when an application cannot be written entirely in Java.

JNI native methods can be used to create, inspect, and update Java objects, call Java methods, catch and throw exceptions, load classes and obtain class information, and perform runtime type checking.

The JNI may also be used with the *Invocation API* to enable an arbitrary native application to embed the Java VM. This allows programmers to easily make their existing applications Java-enabled without having to link with the VM source code. [9]

It is important to remember that once an application uses the JNI, it risks losing two benefits of the Java platform.

First, Java applications that depend on the JNI can no longer readily run on multiple host environments. Even though the part of an application written in the Java programming language is portable to multiple host environments, it will be necessary to recompile the part of the application written in native programming languages.

Second, while the Java programming language is type-safe and secure, native languages such as C or C++ are not. As a result, Java developers must use extra care when writing applications using the JNI. A misbehaving native method can corrupt the entire application. For this reason, Java applications are subject to security checks before invoking JNI features.

As a general rule, developers should architect the application so that native methods are defined in as few classes as possible. This entails a cleaner isolation between native code and the rest of the application.[10]

In HotSpot, the implementation of the JNI functions is relatively straightforward. It uses various VM internal primitives to perform activities such as object creation, method invocation, etc. In general, these are the same runtime primitives used by other subsystems such as the interpreter.

A command line option, `-Xcheck:jni`, is provided to aid in debugging problems in JNI usage by native methods. Specifying `-Xcheck:jni` causes an alternate set of debugging interfaces to be used by JNI calls. The alternate interface verifies arguments to JNI calls more stringently, as well as performing additional internal consistency checks.

HotSpot must take special care to keep track of which threads are currently executing in native methods. During some VM activities, notably some phases of garbage collection, one or more threads must be halted at a *safepoint* in order to guarantee that the Java memory heap is not modified during the sensitive activity. When we wish to bring a thread executing in native code to a safepoint, it is allowed to continue executing native code, but the thread will be stopped when it attempts to return into Java code or make a JNI call.

VM Fatal Error Handling

It is very important to provide easy ways to handle fatal errors for any software. Java Virtual Machine, i.e. JVM is not an exception. A typical fatal error would be `OutOfMemoryError`. Another common fatal error on Windows is called `Access Violation` error which is equivalent to `Segmentation Fault` on Solaris/Linux platforms. It is critical to understand the cause of these kind of fatal errors in order to fix them either in your application or sometimes, in JVM itself.

Usually when JVM crashes on a fatal error, it will dump a hotspot error log file called `hs_err_pid<pid>.log`, (where *<pid>* is replaced by the crashed java process id) to the Windows desktop or the current application directory on Solaris/Linux. Several enhancements have been made to improve the diagnosability of this file since JDK 6 and many of them have been back ported to the JDK-1.4.2_09 release. Here are some highlights of these improvements:

- Memory map is included in the error log file so it is easy to see how memory was laid out during crash.
- `-XX:ErrorFile=` option is provided so that user can set the path name of the error log file.
- `OutOfMemoryError` will trigger the file to be generated as well.

Another important feature is you can specify `-XX:OnError="`*cmd1 args...;com2 ...*`"` to the java command so that whenever VM crashes, it will execute a list of commands you specified within the quotes shown above. A typical usage of this feature is you can invoke the debugger such as dbx or Windbg to look into the crash when that happens. For the earlier releases, you can specify -XX:+ShowMessageBoxOnError as a runtime option so that when VM crashes, you can attach the running Java process to your favorite debugger.

Having said something about HotSpot error log files, here is a brief summary on how JVM internally handles fatal errors.

- The VMError class was invented for aggregating and dumping the hs_err_pid*<pid>*.log file. It is invoked by the OS-specific code when an unrecognized signal/exception is seen.
- The VM uses signals internally for communication. The fatal error handler is invoked when the signal is not recognized. In the unrecognized case, it may come from a fault in application JNI code, OS native libraries, JRE native libraries, or the JVM itself.
- The fatal error handler was carefully written to avoid causing faults itself, in the case of StackOverflow or crashes when critical locks are held (like malloc lock).

Since OutOfMemoryError is so common to some large scale applications, it is critical to provide useful diagnostic message to users so that they could quickly identify a solution, sometimes by just specifying a larger Java heap size. When OutOfMemoryError happens, the error message will indicate which type of memory is problematic. For example, it could be Java heap space or PermGen space etc. Since JDK 6, a stack trace will be included in the error message. Also, -XX:OnOutOfMemoryError="*<cmd>*" option was invented so that a command will be run when the first OutOfMemoryError is thrown. Another nice feature that is worth mentioning is a built-in heap dump at OutOfMemoryError. It is enabled by specifying -XX:+HeapDumpOnOutOfMemoryError option and you can also tell the VM where to put the heap dump file by specifying -XX:HeapDumpPath=*<pathname>*.

Even though applications are carefully written to avoid deadlocks, sometimes it still happens. When deadlock occurs, you can type “Ctrl+Break” on Windows or grab the Java process id and send SIGQUIT to the hang process on Solaris/Linux. A Java level stack trace will be dumped out to the standard out so that you can analyze the reasons of deadlock. Since JDK 6, this feature has been built into jconsole which is a very useful tool in the JDK. So when the application hangs on a deadlock, use jconsole to attach the process and it will analyze which lock is problematic. Most of the time, the deadlock is caused by acquiring locks in the wrong order.

We strongly encourage you to check out the “Trouble-Shooting and Diagnostic Guide”[11]. It contains a lot of information which might be very useful to diagnose fatal errors.

Further Reading

“Resolving the Mysteries of Java SE Classloader”, Jeff Nisewanger, Karen Kinnear, JavaOne 2006.

References

[1] Java Language Specification, Third Edition. Gosling, Joy, Steele, Bracha. http://java.sun.com/docs/books/jls/third_edition/html/execution.html#12.2

[2] Java Virtual Machine Specification, Second Edition. Tim Lindholm, Frank Yellin. http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html

[3] Amendment to Java Virtual Machine Specification. Chapter 5: Loading, Linking and Initializing. http://java.sun.com/docs/books/vmspec/2nd-edition/ConstantPool.pdf

[4] Dynamic Class Loading in the Java Virtual Machine. Shen Liang, Gilad Bracha. Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications, October 1998 http://www.bracha.org/classloaders.ps

[5] “Safe Clsss and Data Evolution in Large and Long-Lived Java Applications”, Mikhail Dmitriev, http://research.sun.com/techrep/2001/smli_tr-2001-98.pdf

[6] Java Language Specification, Third Edition. Gosling, Joy, Steele, Bracha. http://java.sun.com/docs/books/jls/third_edition/html/exceptions.html

[7] “Biased Locking in HotSpot”. http://blogs.oracle.com/dave/entry/biased_locking_in_hotspot

[8] “Let’s say you’re interested in using HotSpot as a vehicle for synchronization research ...”. http://blogs.oracle.com/dave/entry/lets_say_you_re_interested

[9] “Java Native Interface Specifications” http://java.sun.com/javase/6/docs/technotes/guides/jni/spec/jniTOC.html

[10] “The Java Native Interface Programmer’s Guide and Specification”, Sheng Liang, http://java.sun.com/docs/books/jni/html/titlepage.html

[11] “Trouble-Shooting and Diagnostic Guide” https://docs.oracle.com/en/java/javase/21/troubleshoot/troubleshooting-guide.pdf