Diagnostics Guide

Understanding Threads and Locks

A running application is usually made up of one process with its own memory space. A computer is generally running several processes at the same time. For example, a word processor application process might be running alongside a media player application process. Furthermore, a process consists of many concurrently running threads. When you run a Java application, a new JVM process is started.

Each Java process has at least one application thread. Besides the threads of the running Java application, there are also Oracle JRockit JVM internal threads that take care of garbage collection or code generation.

This section contains basic information about threads and locks in the JRockit JVM. The following subjects are discussed:

- Understanding Threads
- Understanding Locks

For information about how to make so-called *thread dumps*, printouts of the stacks of all the threads in an application, see Using Thread Dumps. Thread dumps can be used to diagnose problems and optimize application and JVM performance.

Understanding Threads

A java application consists of one or more threads that run Java code. The entire JVM process consists of the Java threads and some JVM internal threads, for example one or more garbage collection threads, a code optimizer thread and one or more finalizer threads.

From the operating system's point of view the Java threads are just like any application threads. Scheduling of the threads is handled by the operating system, as well as thread priorities.

Within Java, the Java threads are represented by thread objects. Each thread also has a stack, used for storing runtime data. The thread stack has a specific size. If a thread tries to store more items on the stack than the stack size allows, the thread will throw a stack overflow error.

Default Stack Size for Java Threads

This section lists the default stack sizes. You can change the thread stack size with the `-Xss` command line option, for example:

```
java -Xss:512k MyApplication
```

The default stack sizes differ depending upon whether you are using IA32 and X64, as shown in Table 1:

Table 1 Default Stack Size

OS	Default Stack Size
Windows IA32	64 kB
Windows IA64	320 KB
Windows x64	128 kB
Linux IA32	128 kB
Linux IA64	1024 KB
Linux x64	256 kB
Solaris/SPARC	512 KB

Default Stack Size for JVM Internal Threads

A special “system” stack size is used for JVM internal threads; for example, the garbage collection and code generation threads.

The default system stack size is 256 KB on all platforms.

Note: The `-Xss` command line option sets the stack size of both application threads and JVM internal threads.

Understanding Locks

When threads in a process share and update the same data, their activities must be synchronized to avoid errors. In Java, this is done with the `synchronized` keyword, or with `wait` and `notify`. Synchronization is achieved by the use of locks, each of which is associated with an object by the JVM. For a thread to work on an object, it must have control over the lock associated with it, it must “hold” the lock. Only one thread can hold a lock at a time. If a thread tries to take a lock that is already held by another thread, then it must wait until the lock is released. When this happens, there is so called “contention” for the lock.

There are four different kinds of locks:

- *Fat locks:* A fat lock is a lock with a history of contention (several threads trying to take the lock simultaneously), or a lock that has been waited on (for notification).
- *Thin locks:* A thin lock is a lock that does not have any contention.
- *Recursive locks:* A recursive lock is a lock that has been taken by a thread several times without having been released.
- *Lazy locks:* A lazy lock is a lock that is not released when a critical section is exited. Once a lazy lock is acquired by a thread, other threads that try to acquire the lock have to ensure that the lock is, or can be, released. Lazy locks are used by default in Oracle JRockit JVM 27.6. In older releases, lazy locks are only used if you have started the JVM with the `-XXlazyUnlocking` option.

A thin lock can be inflated to a fat lock and a fat lock can be *deflated* to a thin lock. The JRockit JVM uses a complex set of heuristics to determine when to inflate a thin lock to a fat lock and when to deflate a fat lock to a thin lock.

Spinning and Sleeping

Spinning occurs when a thread that wants a specific lock continuously checks that lock to see if it is still taken, instead of yielding CPU-time to another thread.

Alternatively, a thread that tries to take a lock that is already held waits for notification from the lock and goes into a sleeping state. The thread will then wait passively for the lock to be released.

Lock Chains

Several threads can be tied up in what is called *lock chains*. Although they appear somewhat complex, lock chains are fairly straightforward. They can be defined as follows:

- Threads A and B form a lock chain if thread A holds a lock that thread B is trying to take. If A is not trying to take a lock, then the lock chain is “open.”
- If A->B is a lock chain, and B->C is a lock chain, then A->B->C is a more complete lock chain.
- If there is no additional thread waiting for a lock held by C, then A->B->C is a complete and open lock chain.

Lock Chain Types

The JRockit JVM analyzes the threads and forms complete lock chains. There are three possible kinds of lock chains: Open, Deadlocked and Blocked lock chains.

Open Chains

Open lock chains represent a straight dependency, thread A is waiting for B which is waiting for C, and so on. If you have long open lock chains, your application might be wasting time waiting for locks. You may then want to reconsider how locks are used for synchronization in your application.

Deadlock Chains

A deadlocked, or circular, lock chain consists of a chain of threads, in which the first thread in the chain is waiting for the last thread in the chain. In the simplest case, thread A is waiting for thread B, while thread B is waiting for thread A. Note that a deadlocked chain has no head. In thread dumps, the Oracle JRockit JVM selects an arbitrary thread to display as the first thread in the chain.

Deadlocks can never be resolved, and the application will be stuck waiting indefinitely.

Blocked Chains

A blocked lock chain is made up of a lock chain whose head thread is also part of another lock chain, which can be either open or deadlocked. For example, if thread A is waiting for thread B, thread B is waiting for thread A, and thread C is waiting for thread A, then thread A and B form a deadlocked lock chain, while thread C and thread A form a blocked lock chain.

back to top previous next