How to Read a Thread Dump

Want to learn how to read a thread dump? Click here to learn more about thread dumps in Java applications and how to decipher them.

by Justin Albano MVB · CORE · Jun. 23, 18 · Java Zone · Tutorial

Most Java applications developed today involve multiple threads, which, in contrast to its benefits, carries with it a number of subtle difficulties. In a single-threaded application, all resources (shared data, Input/Output (IO) devices, etc.) can be accessed without coordination, knowing that the single thread of execution will be the only thread that utilizes the resource at any given time within the application.

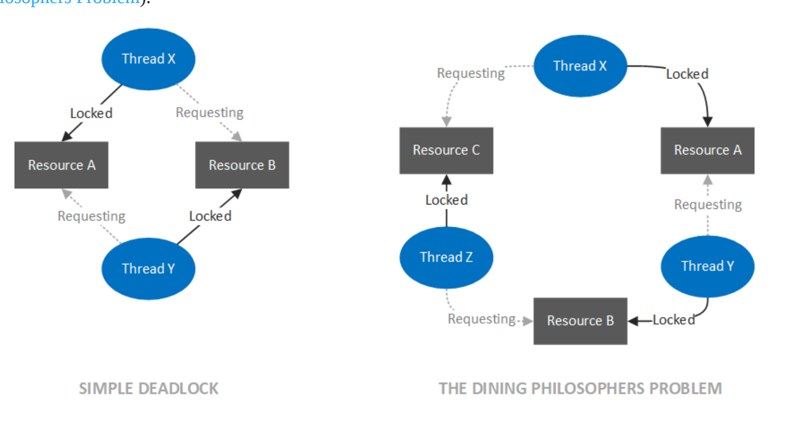
In the case of multithreaded applications, a trade-off is made — increased complexity for a possible gain in performance, where multiple threads can utilize the available (often more than one) Central Processing Unit (CPU) cores. In the right conditions, an application can see a significant performance increase using multiple threads (formalized by Amdahl's Law), but special attention must be paid to ensure that multiple threads coordinate properly when accessing a resource that is needed by two threads. In many cases, frameworks, such as Spring, will abstract direct thread management, but even the improper use of these abstracted threads can cause some hard-to-debug issues. Taking all of these difficulties into consideration, it is likely that, eventually, something will go wrong, and we, as developers, will have to start diagnosing the indeterministic realm of threads.

Fortunately, Java has a mechanism for inspecting the state of all threads in an application at any given time —the thread dump. In this article, we will look at the importance of thread dumps and how to decipher their compact format, as well as how to generate and analyze thread dumps in realistically-sized applications. This article assumes the reader has a basic understanding of threads and the various issues that surround threads, including thread contention and shared resource management. Even with this understanding, before generating and examining a thread dump, it is important to solidify some central threading terminology.

Understanding the Terminology

Java thread dumps can appear cryptic at first, but making sense of thread dumps requires an understanding of some basic terminology. In general, the following terms are key in grasping the meaning and context of a Java thread dump:

- **Thread —** A discrete unit of concurrency that is managed by the Java Virtual Machine (JVM). Threads are mapped to Operating System (OS) threads, called **native threads**, which provide a mechanism for the execution of instructions (code). Each thread has a unique identifier, name, and may be categorized as a **daemon thread** or **non-daemon thread**, where a daemon thread runs independent of other threads in the system and is only killed when either the `Runtime.exit` method has been called (and the security manager authorizes the exiting of the program) or all non-daemon threads have died. For more information, see the `Thread` class documentation.
  - **Alive thread —** a running thread that is performing some work (the *normal* thread state).
  - **Blocked thread —** a thread that attempted to enter a synchronized block but another thread already locked the same synchronized block.
  - **Waiting thread —** a thread that has called the `wait` method (with a possible timeout) on an object and is currently waiting for another thread to call the `notify` method (or `notifyAll`) on the same object. Note that a thread is not considered waiting if it calls the `wait` method on an object with a timeout and the specified timeout has expired.
  - **Sleeping thread —** a thread that is currently not executing as a result of calling the `Thread.sleep` method (with a specified sleep length).
- **Monitor —** a mechanism employed by the JVM to facilitate concurrent access to a single object. This mechanism is instituted using the `synchronized` keyword, where each object in Java has an associated monitor allowing any thread to synchronize, or **lock**, an object, ensuring that no other thread accesses the locked object until the lock is released (the synchronized block is exited). For more information, see the Synchronization section (17.1) of the Java Langauge Specification (JLS).
- **Deadlock —** a scenario in which one thread holds some resource, *A*, and is blocked, waiting for some resource, *B*, to become available, while another thread holds resource *B* and is blocked, waiting for resource *A* to become available. When a deadlock occurs, no progress is made within a program. It is important to note that a deadlock may also occur with more than two threads, where three or more threads all hold a resource required by another thread and are simultaneously blocked, waiting for a resource held by another thread. A special case of this occurs when some thread, *X*, holds resource *A* and requires resource *C*, thread *Y* holds resource *B* and requires resource *A*, and thread *Z* holds resource *C* and requires resource *B* (formally known as the Dining Philosophers Problem).



SIMPLE DEADLOCK THE DINING PHILOSOPHERS PROBLEM

- **Livelock —** a scenario in which thread *A* performs an action that causes thread *B* to perform an action that in turn causes thread *A* to perform its original action. This situation can be visualized as a dog chasing its tail. Similar to deadlock, live-locked threads do not make progress, but unlike deadlock, the threads are not blocked (and instead, are alive).

The above definitions do not constitute a comprehensive vocabulary for Java threads or thread dumps but make up a large portion of the terminology that will be experienced when reading a typical thread dump. For a more detailed lexicon of Java threads and thread dumps, see Section 17 of the JLS and Java Concurrency in Practice.

With this basic understanding of Java threads, we can progress to creating an application from which we will generate a thread dump and, later, examine the key portion of the thread dump to garner useful information about the threads in the program.

Creating an Example Program

In order to generate a thread dump, we need to first execute a Java application. While a simple "hello, world!" application results in an overly simplistic thread dump, a thread dump from an even moderately-sized multithreaded application can be overwhelming. For the sake of understanding the basics of a thread dump, we will use the following program, which starts two threads that eventually become deadlocked:

```
public class DeadlockProgram {

    public static void main(String[] args) throws Exception {

        Object resourceA = new Object();
        Object resourceB = new Object();

        Thread threadLockingResourceAFirst = new Thread(new DeadlockRunnable(resourceA, resourceB));
        Thread threadLockingResourceBFirst = new Thread(new DeadlockRunnable(resourceB, resourceA));

        threadLockingResourceAFirst.start();
```

This program simply creates two resources, `resourceA` and `resourceB`, and starts two threads, `threadLockingResourceAFirst` and `threadLockingResourceBFirst`, that lock each of these resources. The key to causing deadlock is ensuring that `threadLockingResourceAFirst` tries to lock `resourceA` and then lock `resourceB` while `threadLockingResourceBFirst` tries to lock `resourceB` and then `resourceA`. Delays are added to ensure that `threadLockingResourceAFirst` sleeps before it is able to lock `resourceB` and `threadLockingResourceBFirst` is given enough time to lock `resourceB` before `threadLockingResourceAFirst` wakes. `threadLockingResourceBFirst` then sleeps and when both threads await, they find that the second resource they desired has already been locked and both threads block, waiting for the other thread to relinquish its locked resource (which never occurs).

Executing this program results in the following output, where the object hashes (the numeric following `java.lang.Object@`) will vary between each execution:

```
Thread-0: locked resource -> java.lang.Object@149bc794
Thread-1: locked resource -> java.lang.Object@17c10009
```

At the completion of this output, the program appears as though it is running (the process executing this program does not terminate), but no further work is being done. This is a deadlock in practice. In order to troubleshoot the issue at hand, we must generate a thread dump manually and inspect the state of the threads in the dump.

Generating a Thread Dump

In practice, a Java program might terminate abnormally and generate a thread dump automatically, but, in some cases (such as with many deadlocks), the program does not terminate but appears as though it is *stuck*. To generate a thread dump for this stuck program, we must first discover the Process ID (PID) for the program. To do this, we use the JVM Process Status (JPS) tool that is included with all Java Development Kit (JDK) 7+ installations. To find the PID for our deadlocked program, we simply execute `jps` in the terminal (either Windows or Linux):

```
$ jps
11568 DeadlockProgram
15584 Jps
15636
```

The first column represents the Local VM ID (lvmid) for the running Java process. In the context of a local JVM, the lvmid maps to the PID for the Java process. Note that this value will likely differ from the value above. The second column represents the name of the application, which may map to the name of the main class, a Java Archive (JAR) file, or `Unknown`, depending on the characteristics of the program run.

In our case, the application name is `DeadlockProgram`, which matches the name of the main class file that was executed when our program started. In the above example, the PID for our program is `11568`, which provides us with enough information to generate thread dump. To generate the dump, we use the `jstack` program (included with all JDK 7+ installations), supplying the `-l` flag (which creates a long listing) and the PID of our deadlocked program, and piping the output to some text file (i.e. `thread_dump.txt`):

```
jstack -l 11568 > thread_dump.txt
```

This `thread_dump.txt` file now contains the thread dump for our deadlocked program and includes some very useful information for diagnosis the root cause of our deadlock problem. Note that if we did not have a JDK 7+ installed, we could also generate a thread dump by quitting the deadlocked program with a `SIGQUIT` signal. To do this on Linux, simply kill deadlocked program using its PID (`11568` in our example), along with the `-3` flag:

```
kill -3 11568
```

Reading a Simple Thread Dump

Opening the `thread_dump.txt` file, we see that it contains the following:

```
2018-06-19 16:44:44
Full thread dump Java HotSpot(TM) 64-Bit Server VM (10.0.1+10 mixed mode):

Threads class SMR info:
_java_thread_list=0x00000250e5488a00, length=13, elements={
0x00000250e4979000, 0x00000250e4982800, 0x00000250e52f2800, 0x00000250e4992800,
0x00000250e4995800, 0x00000250e49a5800, 0x00000250e49ae800, 0x00000250e5324000,
0x00000250e54cd800, 0x00000250e54cf000, 0x00000250e54d1800, 0x00000250e54d2000,
0x00000250e54d0800
}

```

Introductory Information

Although this file may appear overwhelming at first, it is actually simple if we take each section one step at a time. The first line of the dump displays the timestamp of when the dump was generated, while the second line contains the diagnostic information about the JVM from which the dump was generated:

```
2018-06-19 16:44:44
Full thread dump Java HotSpot(TM) 64-Bit Server VM (10.0.1+10 mixed mode):
```

While these lines do not provide any information with regards to the threads in our system, they provide a context from which the rest of the dump can be framed (i.e. which JVM generated the dump and when the dump was generated).

General Threading Information

The next section begins to provide us with some useful information about the threads that were running at the time the thread dump was taken:

```
Threads class SMR info:
_java_thread_list=0x00000250e5488a00, length=13, elements={
0x00000250e4979000, 0x00000250e4982800, 0x00000250e52f2800, 0x00000250e4992800,
0x00000250e4995800, 0x00000250e49a5800, 0x00000250e49ae800, 0x00000250e5324000,
0x00000250e54cd800, 0x00000250e54cf000, 0x00000250e54d1800, 0x00000250e54d2000,
0x00000250e54d0800
}
```

This section contains the thread list Safe Memory Reclamation (SMR) information[1], which enumerates the addresses of all non-JVM internal threads (e.g. non-VM and non-Garbage Collection (GC)). If we examine these addresses, we see that they correspond to the `tid` value — the address of the native thread object, not the Thread ID, as we will see shortly — of each of the numbered threads in the dump (note that ellipses are used to hide superfluous information):

```
"Reference Handler" #2 ... tid=0x00000250e4979000 ...
"Finalizer" #3 ... tid=0x00000250e4982800 ...
"Signal Dispatcher" #4 ... tid=0x00000250e52f2800 ...
"Attach Listener" #5 ... tid=0x00000250e4992800 ...
"C2 CompilerThread0" #6 ... tid=0x00000250e4995800 ...
"C2 CompilerThread1" #7 ... tid=0x00000250e49a5800 ...
"C1 CompilerThread2" #8 ... tid=0x00000250e49ae800 ...
"Sweeper thread" #9 ... tid=0x00000250e5324000 ...
"Service Thread" #10 ... tid=0x00000250e54cd800 ...
"Common-Cleaner" #11 ... tid=0x00000250e54cf000 ...
"Thread-0" #12 ... tid=0x00000250e54d1800 ...
```

Threads

Directly following the SMR information is the list of threads. The first thread listed in for our deadlocked program is the `Reference Handler` thread:

```
"Reference Handler" #2 daemon prio=10 os_prio=2 tid=0x00000250e4979000 nid=0x3c28 waiting on condition  [0x000000b82a9ff000]
   java.lang.Thread.State: RUNNABLE
    at java.lang.ref.Reference.waitForReferencePendingList(java.base@10.0.1/Native Method)
    at java.lang.ref.Reference.processPendingReferences(java.base@10.0.1/Reference.java:174)
    at java.lang.ref.Reference.access$000(java.base@10.0.1/Reference.java:44)
    at java.lang.ref.Reference$ReferenceHandler.run(java.base@10.0.1/Reference.java:138)

   Locked ownable synchronizers:
    - None
```

Thread Summary

The first line of each thread represents the thread summary, which contains the following items:

SECTION	EXAMPLE	DESCRIPTION
Name	`"Reference Handler"`	Human-readable name of the thread. This name can be set by calling the `setName` method on a `Thread` object and be obtained by calling `getName` on the object.
ID	`#2`	A unique ID associated with each `Thread` object. This number is generated, starting at `1`, for all threads in the system. Each time a `Thread` object is created, the sequence number is incremented and then assigned to the newly created `Thread`. This ID is read-only and can be obtained by calling `getId` on a `Thread` object.
Daemon status	`daemon`	A tag denoting if the thread is a daemon thread. If the thread is a daemon, this tag will be present; if the thread is a non-daemon thread, no tag will be present. For example, `Thread-0` is not a daemon thread and therefore has no associated `daemon` tag in its summary: `Thread-0" #12 prio=5...`.
Priority	`prio=10`	The numeric priority of the Java thread. Note that this does not necessarily correspond to the priority of the OS thread to with the Java thread is dispatched. The priority of a `Thread` object can be set using the `setPriority` method and obtained using the `getPriority` method.
OS Thread Priority	`os_prio=2`	The OS thread priority. This priority can differ from the Java thread priority and corresponds to the OS thread on which the Java thread is dispatched.
Address	`tid=0x00000250e4979000`	The address of the Java thread. This address represents the pointer address of the Java Native Interface (JNI) native `Thread` object (the C++ `Thread` object that backs the Java thread through the JNI). This value is obtained by converting the pointer to this (of the C++ object that backs the Java `Thread` object) to an integer on line 879 of `hotspot/share/runtime/thread.cpp`: `st->print("tid=" INTPTR_FORMAT " ", p2i(this));` Although the key for this item (`tid`) may appear to be the thread ID, it is actually the address of the underlying JNI C++ `Thread` object and thus is *not* the ID returned when calling `getId` on a Java `Thread` object.
OS Thread ID	`nid=0x3c28`	The unique ID of the OS thread to which the Java `Thread` is mapped. This value is printed on line 42 of `hotspot/share/runtime/osThread.cpp`: `st->print("nid=0x%x ", thread_id());`
Status	`waiting on condition`	A human-readable string depicting the current status of the thread. This string provides supplementary information beyond the basic thread state (see below) and can be useful in discovering the intended actions of a thread (i.e. was the thread trying to acquire a lock or waiting on a condition when it blocked).
Last Known Java Stack Pointer	`[0x000000b82a9ff000]`	The last known Stack Pointer (SP) for the stack associated with the thread. This value is supplied using native C++ code and is interlaced with the Java `Thread` class using the JNI. This value is obtained using the `last_Java_sp()` native method and is formatted into the thread dump on line 2886 of hotspot/share/runtime/thread.cpp: `st->print_cr("[" INTPTR_FORMAT "]", (intptr_t)last_Java_sp() & ~right_n_bits(12));` For simple thread dumps, this information may not be useful, but for more complex diagnostics, this SP value can be used to trace lock acquisition through a program.

Thread State

The second line represents the current state of the thread. The possible states for a thread are captured in the `Thread.State` enumeration:

- `NEW`
- `RUNNABLE`
- `BLOCKED`
- `WAITING`
- `TIMED_WAITING`
- `TERMINATED`

For more information on the meaning of each state, see the `Thread.State` documentation.

Thread Stack Trace

The next section contains the stack trace for the thread at the time of the dump. This stack trace resembles the stack trace printed when an uncaught exception occurs and simply denotes the class and line that the thread was executing when the dump was taken. In the case of the `Reference Handler` thread, there is nothing of particular importance that we see in the stack trace, but if we look at the stack trace for `Thread-0`[2], we see a difference from the standard stack trace:

```
"Thread-0" #12 prio=5 os_prio=0 tid=0x00000250e54d1800 nid=0xdec waiting for monitor entry  [0x000000b82b4ff000]
   java.lang.Thread.State: BLOCKED (on object monitor)
    at DeadlockProgram$DeadlockRunnable.run(DeadlockProgram.java:34)
    - waiting to lock <0x00000000894465b0> (a java.lang.Object)
    - locked <0x00000000894465a0> (a java.lang.Object)
    at java.lang.Thread.run(java.base@10.0.1/Thread.java:844)

   Locked ownable synchronizers:
    - None
```

Within this stack trace, we can see that locking information has been added, which tells us that this thread is waiting for a lock on an object with an address of `0x00000000894465b0` (and a type of `java.lang.Object`) and, at this point in the stack trace, holds a lock on an object with an address of `0x00000000894465a0` (also of type `java.lang.Object`). This supplemental lock information is important when diagnosing deadlocks, as we will see in the following sections.

Locked Ownable Synchronizer

The last portion of the thread information contains a list of synchronizers (objects that can be used for synchronization, such as locks) that are exclusively owned by a thread. According to the official Java documentation, "an ownable synchronizer is a synchronizer that may be exclusively owned by a thread and uses `AbstractOwnableSynchronizer` (or its subclass) to implement its synchronization property. `ReentrantLock` and the write-lock (but not the read-lock) of `ReentrantReadWriteLock` are two examples of ownable synchronizers provided by the platform.

For more information on locked ownable synchronizers, see this Stack Overflow post.

JVM Threads

The next section of the thread dump contains the JVM-internal (non-application) threads that are bound to the OS. Since these threads do not exist within a Java application, they do not have a thread ID. These threads are usually composed of GC threads and other threads used by the JVM to run and maintain a Java application:

```
"VM Thread" os_prio=2 tid=0x00000250e496d800 nid=0x1920 runnable

"GC Thread#0" os_prio=2 tid=0x00000250c35b5800 nid=0x310c runnable

"GC Thread#1" os_prio=2 tid=0x00000250c35b8000 nid=0x12b4 runnable

"GC Thread#2" os_prio=2 tid=0x00000250c35ba800 nid=0x43f8 runnable

"GC Thread#3" os_prio=2 tid=0x00000250c35c0800 nid=0x20c0 runnable

"G1 Main Marker" os_prio=2 tid=0x00000250c3633000 nid=0x4068 runnable
```

JNI Global References

This section captures the number of global references maintained by the JVM through the JNI. These references may cause memory leaks under certain circumstances and are *not* automatically garbage collected.

```
JNI global references: 2
```

For many simple issues, this information is unused, but it is important to understand the importance of these global references. For more information, see this Stack Overflow post.

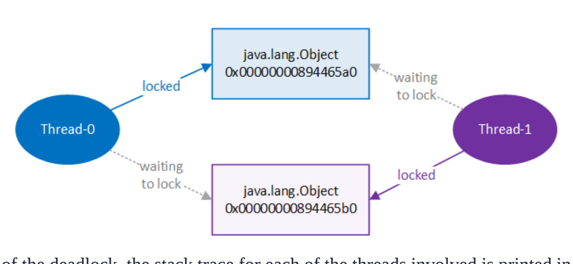
Deadlocked Threads

The final section of the thread dump contains information about discovered deadlocks. This is not always the case: If the application does not have one or more detected deadlocks, this section will be omitted. Since our application was designed with a deadlock, the thread dump correctly captures this contention with the following message:

```
Found one Java-level deadlock:
=============================
"Thread-0":
  waiting to lock monitor 0x00000250e4982480 (object 0x00000000894465b0, a java.lang.Object),
  which is held by "Thread-1"
"Thread-1":
  waiting to lock monitor 0x00000250e4982380 (object 0x00000000894465a0, a java.lang.Object),
  which is held by "Thread-0"

Java stack information for the threads listed above:
===================================================
```

The first subsection describes the deadlock scenario: `Thread-0` is waiting to lock a monitor (through the `synchronized` statement around the `firstResource` and `secondResource` in our application) that is held while `Thread-1` is waiting to lock a monitor held by `Thread-0`. This circular dependency is the textbook definition of a deadlock (contrived by our application) and is illustrated in the figure below:



In addition to the description of the deadlock, the stack trace for each of the threads involved is printed in the second subsection. This allows us to track down the line and locks (the objects being used as monitor locks in this case) that are causing the deadlock. For example, if we examine line 34 of our application, we find the following content:

```
printLockedResource(secondResource);
```

This line represents the first line of the `synchronized` block causing the deadlock and tips us off to the fact that synchronizing on `secondResource` is the root of the deadlock. In order to solve this deadlock, we would have to instead synchronize on `resourceA` and `resourceB` in the same order in both threads. If we do this, we end up with the following application:

```
public class DeadlockProgram {

    public static void main(String[] args) throws Exception {

        Object resourceA = new Object();
        Object resourceB = new Object();

        Thread threadLockingResourceAFirst = new Thread(new DeadlockRunnable(resourceA, resourceB));
        Thread threadLockingResourceBFirst = new Thread(new DeadlockRunnable(resourceA, resourceB));

        threadLockingResourceAFirst.start();
```

This application produces the following output and completes without deadlocking (note that the addresses of the `Object` objects will vary by execution):

```
Thread-0: locked resource -> java.lang.Object@1ad895d1
Thread-0: locked resource -> java.lang.Object@6e41d7dd
Thread-1: locked resource -> java.lang.Object@1ad895d1
Thread-1: locked resource -> java.lang.Object@6e41d7dd
```

In summary, using only the information provided in the thread dump, we can find and fix a deadlocked application. Although this inspection technique is sufficient for many simple applications (or applications that have only a small number of deadlocks), dealing with more complex thread dumps may need to be handled in a different way.

Handling More Complex Thread Dumps

When handling production applications, thread dumps can become overwhelming very quickly. A single JVM may have hundreds of threads running at the same time and deadlocks may involve more than two threads (or there may be more than one concurrency issue as a side-effect of a single cause) and parsing through this firehose of information can be tedious and unruly.

In order to handle these large-scale situations, Thread Dump Analyzers (TDAs) should be the tool of choice. These tools parse Java thread dumps display otherwise confusing information in a manageable form (commonly with a graph or other visual aid) and may even perform static analysis of the dump to discover issues. While the best tool for a situation will vary by circumstance, some of the most common TDAs include the following:

- fastThread
- Spotify TDA
- IBM Thread and Monitor Dump Analyze for Java
- irockel TDA

While this is far from a comprehensive list of TDAs, each performs enough analysis and visual sorting to reduce the manual burden of decyphering thread dumps.

Conclusion

Thread dumps are an excellent mechanism for analyzing the state of a Java application, especially a misbehaving, multithreaded application, but without proper knowledge, they can quickly add more confusion to an already difficult problem. In this article, we developed a deadlocked application and generated a thread dump of the stuck program. Upon analyzing the dump, we found the root cause of the deadlock and fixed it accordingly. This is not always so easy, and for many production applications, the help of a TDA may be required.

In either case, each professional Java developer should understand the basics of thread dumps, including their structure, the information that can be garnered from them, and how to utilize them to find the root cause of common multithreading problems. While a thread dump is not a silver bullet for all multithreading woes, it is an important tool in quantifying and reducing the complexity of diagnosing a common problem in the world of Java applications.

Footnotes

1. Thread SMR is an involved topic and beyond the scope of this article. The interested reader can find more information under the Hazard Pointer Wikipedia page, as well as Michael Maged's 2004 article on the topic. For more information on the implementation of SMR in Java Threads, see the `threadSMR.cpp` implementation file for the HotSpot VM.
2. This thread, along with `Thread-1`, are called **anonymous threads** because they are not explicitly named (i.e. no name was provided by calling `setName` on the `Thread` object or naming the thread using a constructor argument). The default name for anonymous threads in Java is `Thread-` followed by a 0-indexed ID that is incremented for each anonymous thread (e.g. `Thread-0`, `Thread-1`, etc.). The code used by the Thread class to generate names for anonymous classes is as follows:

```
class Thread implements Runnable {
    private static int threadInitNumber;
    private static synchronized int nextThreadNum() {
        return threadInitNumber++;
    }
    public Thread(Runnable target) {
        init(null, target, "Thread-" + nextThreadNum(), 0);
    }
    // ...
}
```

Topics: THREADS, DUMP FILE, DEADLOCK, CONTENTION, THREAD DUMP, ANALYSIS, TUTORIAL, JAVA