

Revelations on Java signal handling and termination

Taking advantage of improvements to JVM 1.3.1

Skill Level: Intermediate

[Chris White \(Chris.White@uk.ibm.com\)](mailto:Chris.White@uk.ibm.com)
Software Engineer
IBM

01 Jan 2002

Java Virtual Machine (JVM) signal handling and termination behavior got a makeover in version 1.3.1. Many Java developers might not know about the JVM's use of signals and the facilities it provides to an application during the final stages of the JVM's life. In this article, JVM engineer Chris White gives insight into why the JVM uses signal handlers and describes how to deploy your own application signal handlers without fear of compromising the JVM. This article also shows how to write your own hooks that can be called when the JVM terminates normally or in an application crash. **Note:** *The signal handling described in this article is available with the IBM JVM, versions 1.3.1 and 1.4.2 only.*

What is signal handling?

This article assumes a basic understanding of signal handling, Java, and Java Native Interface (JNI). It discusses the signal handling behavior of IBM's JVM 1.3.1 for

- IBM OMVS on z/OS (OS/390), called z/OS in this article
- IBM AIX
- Microsoft Windows

Although this article also refers to Linux signal handling, IBM's latest release of the

Java Virtual Machine on Linux is at 1.3.0, so the overall behavior will differ. For other JVMs and operating systems the techniques described in this article are applicable, but the overall behavior is likely to differ.

With signals, an operating system can interrupt a running application and cause it to enter a special handler function. Signals can be raised for many different reasons; for example, a SIGSEGV will be raised when a process attempts to access an address for which it does not have permission. A SIGINT might be raised when the user requests termination by typing CTRL-C. With a signal handler your application can trap signals and perform any necessary processing. In the case of a SIGSEGV your handler could report essential diagnostic information to help diagnose the fault, or even recover and thus improve the reliability of your application.

To install a signal handler for your native application use the system function `sigaction()` (on Unix type platforms) or `signal()` (on Windows or Unix platforms). In both cases, specify the number of the signal that is to be handled and a reference to your signal handler function. If the call is successful, a reference to the previously installed signal handler will be returned.

When a signal handler is installed for a signal it overrides any previously-installed handler for that signal. And, a signal handler is processed widely and hence services a signal sent to any thread. There are some interesting and fundamental differences between operating systems regarding signal handling. For all platforms, whenever a signal is directed at a particular thread (such as with a `pthread_kill()` or `raise()` system call, or when an exception occurs on that thread), the signal handler runs within that thread's context. However if a signal is raised at the process level, by another process such as SIGINT raised when CTRL-C is entered, then the behavior is platform-specific, as follows.

Table 1. Operating systems and behaviors

On z/OS and AIX	A single thread, chosen by the operating system, receives the signal.
Linux	All threads receive the signal, and the signal handler is invoked on each thread. Linux threads are just separate processes that share the same address space, so it is also possible for another application to raise a signal on a specific thread.
Windows	A new thread is created for executing the signal handler. This thread dies once the signal handler is complete.

The JVM's use of signals is platform-dependent; for certain platforms the JVM may use signals for efficient byte code interpretation, or for suspending, resuming, and interrupting threads. However, signals are used commonly across all JVMs with abnormal termination, where the JVM performs the necessary cleanup and attempts to gather useful diagnostic information for a dump.

By understanding how the JVM deals with signals and the processing it does on termination, you can exploit signal handling in your applications and handle abnormal termination reliably.

IBM JVM 1.3.1 improvements

With IBM's JVM 1.3.1, signal handling has been improved to give a consistent termination sequence and to ensure that the JVM behaves itself when signals raised for an application are not destined for the JVM. The reliability of the termination logic has also been greatly improved. In particular, the JVM can ensure that your Java application shutdown hooks are run under normal and interrupted exit (for example, CTRL-C) conditions. Abnormal termination of the JVM now returns the correct status to the operating system, and the resulting OS-generated diagnostic information gives an accurate account of the problem. Previously, on some platforms, a SIGABRT condition was generated that occasionally led to confusion.

Signals used by the JVM

The [table](#) below shows the signals used by the JVM for the platforms supported by IBM. The signals have been grouped in the table by type or use, as follows.

- **Exceptions** (in red) - The operating system synchronously raises an appropriate exception signal whenever a fatal condition occurs.
- **Errors** (in blue) - The JVM raises a SIGABRT if it detects a situation from which it cannot recover.
- **Interrupts** (in green) - Interrupt signals are raised asynchronously, from outside a JVM process, to request shutdown.
- **Controls** - Other signals used by the JVM for control purposes.

Table 2. Signals used by the JVM

SIGSEGV	Incorrect access to memory (write to inaccessible memory)	No	Yes	Yes	Yes
SIGILL	Illegal instruction (attempt to invoke a unknown machine instruction)	No	Yes	Yes	Yes
SIGFPE	Floating point exception (divide by zero)	No	Yes	Yes	Yes

SIGBUS	Bus error (attempt to address nonexistent memory location)	Yes	Yes	Yes	No
SIGSYS	Bad system call issued	Yes	Yes	Yes	No
SIGXCPU	CPU time limit exceeded (you've been running too long!)	Yes	Yes	Yes	No
SIGXFSZ	File size limit exceeded	Yes	Yes	Yes	No
SIGEMT	EMT instruction (AIX specific)	Yes	No	Yes	No
SIGABRT	Abnormal termination. The JVM raises this signal whenever it detects a JVM fault.	Yes	Yes	Yes	Yes
SIGINT	Interactive attention (CTRL-C). JVM will exit normally.	Yes	Yes	Yes	Yes
SIGTERM	Termination request. JVM will exit normally.	Yes	Yes	Yes	Yes
SIGHUP	Hang up. JVM will exit normally.	Yes	Yes	Yes	No
SIGUSR1	User defined. Used by some JVMs for internal control purposes.	No	Yes	No	No
SIGUSR2	User defined. Used by some JVMs for internal control purposes.	No	No	Yes	No

SIGQUIT	A quit signal for a terminal. JVM uses this for taking Java core dumps.	Yes	Yes	Yes	No
SIGBREAK	A break signal from a terminal. JVM uses this for taking Java core dumps.	Yes	No	No	Yes
SIGTRAP	Internal for use by dbx or ptrace. Used by some JVMs for internal control purposes.	Yes (not for AIX)	Yes	Yes	No
SIGPIPE	A write to a pipe that is not being read. JVM ignores this.	No	Yes	Yes	No
No Name (40)	An AIX reserved signal. Used by the AIX JVM for internal control purposes.	No	No	Yes	No

Note that **-Xrs** (reduce signal usage) is a JVM option that can be used to prevent the JVM from using most signals. See Sun's Java application launcher page (in [Resources](#) for more information).

How the JVM processes signals

When a signal is raised that is of interest to the JVM, a signal handler is called. This handler determines whether it has been called for a Java or non-Java thread. If the signal is for a Java thread, the JVM takes control of the signal handling. If the signal is for a non-Java thread, and the application that installed the JVM had previously installed its own handle for the signal, then control is given to that handler.

Otherwise the signal is ignored (even if this is not the signal's default action). The one exception to this rule is on Windows, where for an externally generated signal (you enter CTRL-C or CTRL-BREAK, for example) a new thread is created to execute the signal handler. In this case the JVM signal handler assumes that the signal is for the JVM.

What are Java threads?

A Java thread is one that is known to the JVM. The following statements define when code is running as a Java or non-Java thread:

- All Java code runs under a Java thread.
- All native code called from Java code runs under a Java thread.
- A native application thread that successfully calls JNI function: `JNI_CreateJavaVM()` or `AttachCurrentThread()`, becomes a Java thread.
- A native application thread that successfully calls JNI function: `DestroyJavaVM()` or `DetachCurrentThread()`, becomes a non-Java thread.
- All other native application threads will be non-Java threads.

For exception and error signals, as shown in the [table](#) above, the JVM enters a controlled shutdown sequence where it:

- Outputs a Java core dump, to describe the JVM state at the point of failure
- Calls any application installed abort hook
- Performs the necessary cleanup to give a clean shutdown.

For interrupt signals the JVM also enters a controlled shutdown sequence, but this time it is treated as a normal termination that:

- Runs all application shutdown hooks
- Calls any application installed exit hook
- Performs the necessary JVM cleanup.

The shutdown is identical to a call to the Java method `System.exit()`.

Other signals used by the JVM are for internal control purposes and do not cause it to terminate. The only control signal of interest is `SIGQUIT` (on Unix type platforms) and `SIGBREAK` (on Windows), which cause a Java core dump to be generated.

Writing well-behaved native signal handlers

If your native application, which creates a JVM through `JNI_CreateJavaVM()`, is

to use its own signal handlers then ideally these should be installed prior to creating the JVM. If you install handlers for signals that are used by the JVM *after* JVM installation, then you must ensure they chain back to the JVM's signal handler. Otherwise, the JVM might malfunction. The following examples show how to write a well-behaved signal handler that chains back to a previously installed handler. In a perfect world your signal handler should know whether or not the signal was destined for it, by looking at some other state; in this case it should not chain to the previous handler.

See the example signal handler code [for Unix type platforms](#) or [for Windows platforms](#).

Application signal handlers should be installed prior to the JVM or it can be detrimental to JVM performance. Some JVMs use signals to trap recoverable error conditions during runtime, so any application signal handler will slow the JVM.

Under certain conditions it is desirable for the JVM to be configured to use the least number of signals it can. For example, on Windows if the JVM is run as a service, such as the servlet engine for a Web server, it can receive CTRL_LOGOFF_EVENT but should not initiate shutdown since the operating system will not actually terminate the process. To avoid this type of problem the -Xrs JVM option has been provided to reduce the number of signals that the JVM uses (see [Table 1](#) for details). But if the -Xrs option is used, certain JVM behavior will be disabled:

- For an interrupted exit, Java shutdown and native JVM exit hooks will not be run.
- Under certain exception conditions JVM abort hooks will not be run, and a Java core dump will not be produced.
- Sending a SIGQUIT/SIGBREAK signal to the JVM process will not produce a Java core dump.

Writing Java signal handlers

A little-known feature of Java is the ability of an application to install its own signal handler, which is supported through the `sun.misc.Signal` class. However, use caution when using classes from the `sun.misc` package because it contains undocumented support classes that may change between releases of Java. You can install a Java handler for any signal that is not used by the JVM. These signal handlers are similar to native handlers because they're invoked when a native system signal is raised, but they will always run as a separate Java thread. Essentially, when a signal is raised for which a Java signal handler is available, the JVM's "signal dispatcher thread" is woken up and informed of the signal. The signal dispatcher thread then invokes a Java method to create and start a new thread for

the installed Java signal handler.

To write a Java signal handler, define a class that implements the `sun.misc.SignalHandler` interface and register the handler by using the `sun.misc.Signal.handle()` method. The following example, which installs a Java signal handler for the SIGALRM signal, shows how it is done. The idea is to wrap another Java application, specified on the command line, so that whenever a SIGALRM signal is received a list of the current threads (for the default thread group) will be output. It would be very easy to extend the signal handler to output additional information about the application or take some specific action, such as calling `System.gc()`.

```
import sun.misc.Signal;
import sun.misc.SignalHandler;
import java.lang.reflect.*;

// Application Wrapper
// usage: java AppWrap <app name> <app arg1> ... <app argn>
// where: <app name> is the name of the wrapped application class
//         containing a main method
//         <app arg1> ... <app argn> are the application's arguments
class AppWrap {
    public static void main(String[] args) {
        try {
            // Install diagnostic signal handler
            DiagSignalHandler.install("ALRM");

            // Get the passed application's class
            Class wrappedClass = Class.forName(args[0]);

            // Setup application's input arguments
            String wrappedArgs[] = new String[args.length-1];
            for (int i = 0; i < wrappedArgs.length; i++) {
                wrappedArgs[i] = args[i+1];
            }

            // Get the main method for the application
            Class[] argTypes = new Class[1];
            argTypes[0] = wrappedArgs.getClass();
            Method mainMethod = wrappedClass.getMethod("main", argTypes);

            // Invoke the application's main method
            Object[] argValues = new Object[1];
            argValues[0] = wrappedArgs;
            mainMethod.invoke(wrappedClass, argValues);

        } catch (Exception e) {
            System.out.println("AppWrap exception "+e);
        }
    }
}

// Diagnostic Signal Handler class definition
class DiagSignalHandler implements SignalHandler {

    private SignalHandler oldHandler;

    // Static method to install the signal handler
    public static DiagSignalHandler install(String signalName) {
        Signal diagSignal = new Signal(signalName);
        DiagSignalHandler diagHandler = new DiagSignalHandler();
        diagHandler.oldHandler = Signal.handle(diagSignal,diagHandler);
    }
}
```



```

        return diagHandler;
    }

    // Signal handler method
    public void handle(Signal sig) {
        System.out.println("Diagnostic Signal handler called for signal "+sig);
        try {
            // Output information for each thread
            Thread[] threadArray = new Thread[Thread.activeCount()];
            int numThreads = Thread.enumerate(threadArray);
            System.out.println("Current threads:");
            for (int i=0; i < numThreads; i++) {
                System.out.println("    "+threadArray[i]);
            }

            // Chain back to previous handler, if one exists
            if ( oldHandler != SIG_DFL && oldHandler != SIG_IGN ) {
                oldHandler.handle(sig);
            }

        } catch (Exception e) {
            System.out.println("Signal handler failed, reason "+e);
        }
    }
}

```

Because of the platform-specific nature of signals, this application will not run under Windows since it does not support the SIGALRM signal. But you could modify the application to use SIGINT, and start up with the -Xrs Java option.

An advantage of using Java signal handlers instead of native signal handlers is that your implementation can be completely in Java, keeping the application simple. Also, with Java signal handling you keep an object-oriented approach and refer to signals by name, making the code more readable than its C equivalent. There are many uses for Java signal handlers; you could create a simple interprocess communication between native and Java applications, and the signal could instruct the Java application to dump out useful diagnostics (as demonstrated by the above application) or simply suspend/resume itself.

Writing JVM abort and exit hooks

JVM abort and exit hooks enable your native applications to perform some last minute tidying prior to JVM termination. They are analogous to native exit hooks, installed with the `atexit()` system function. Exit hooks are called during normal JVM termination or when the JVM is requested to shutdown by an interrupt signal (for example, entering CTRL-C). Abort hooks are called during abnormal JVM termination, such as when an exception signal is raised or `abort()` is called. Abort hooks are particularly useful because they allow your native application to perform last minute tidying action prior to termination. Note that abnormal JVM termination during a JNI call will mean that it never returns to your application.

You set up abort and exit hooks with the abort and exit JVM options, passed to the `JNI_CreateJavaVM()` function. Each option takes as its parameter the address of

the abort or exit hook function as appropriate. The hook function should have the following C signature:

- Abort hook - (void *)hookFunc(void *);
- Exit hook - (void *)hookFunc(int);

The following example shows how to specify an application abort and exit hook for the `JNI_CreateJavaVM()` call.

```
#include <stdlib.h>
#include <jni.h>

/* Example JVM abort hook function */
void myAbortHook() {
    fprintf(stderr, "myAbortHook called. JVM terminated abnormally\n");
}

/* Example JVM exit hook function */
void myExitHook(int status) {
    fprintf(stdout, "myExitHook called with status %d\n", status);
}

/* Application to install the above abort and exit hook functions */
int main(int argc, char *argv[]) {
    JavaVMInitArgs jvm_args;
    JavaVMOption options[2];
    jint status;
    JavaVM *jvm = 0;
    JNIEnv *env;

    /* Setup the JVM options to include the hook functions */
    options[0].optionString = "abort";
    options[0].extraInfo = (void *)&myAbortHook;
    options[1].optionString = "exit";
    options[1].extraInfo = (void *)&myExitHook;
    __etoa(options[0].optionString); /* Include for z/OS only */
    __etoa(options[1].optionString); /* Include for z/OS only */

    jvm_args.nOptions = 2;
    jvm_args.version = JNI_VERSION_1_2;
    jvm_args.options = options;

    /* Start the JVM */
    status = JNI_CreateJavaVM(&jvm, (void **)&env, &jvm_args);
    if (status) {
        fprintf(stderr, "JVM create failed, shutting down\n");
        exit(1);
    }
    fprintf(stdout, "JVM installed\n");

    /* If "fail" was specified on the command line then cause */
    /* the application to crash, thus invoking the abort hook */
    if (argc > 1 && !strcmp(argv[1], "fail")) {
        int *p = (int *)123;
        fprintf(stdout, "About to crash\n");
        *p = 1;
    }

    /* Destroy the JVM. If we reach here the exit hook will be invoked */
    status = (*jvm)->DestroyJavaVM(jvm);
    fprintf(stdout, "JVM destroyed\n");
    return 0;
}
```

```
}
```

When run with no arguments, this application outputs the following text to the command line:

```
JVM installed
myExitHook called with status 0
JVM destroyed
```

When run with the "fail" argument this application outputs the following:

```
JVM installed
About to crash
<platform specific info from JVM>
Writing Java core file ....
Written Java core to <filename>
myAbortHook called. JVM terminated abnormally
<platform specific info>
```

Writing Java shutdown hooks

Shutdown hooks are a Java feature that let you have a piece of Java code run whenever the JVM terminates under one of the following conditions:

- The program exits normally, such as when the last non-daemon thread exits or when the `Runtime.exit()` method is invoked.
- The virtual machine is terminated in response to a user interrupt, such as typing CTRL-C, or a system-wide event, such as user logoff or system shutdown (for example, the JVM receives one of the interrupt signals SIGHUP (Unix Only), SIGINT, or SIGTERM).

Shutdown hooks will not be run if

- `Runtime.halt()` method is called to terminate the JVM. `Runtime.halt()` is provided to allow a quick shutdown of the JVM.
- The `-Xrs` JVM option is specified.
- The JVM exits abnormally, such as an exception condition or forced abort generated by the JVM software.

A shutdown hook is a Java class that extends `java.lang.Thread` and is installed with the `Runtime.addShutdownHook()` method. Your application may install multiple shutdown hooks. On JVM termination, each shutdown hook will be started and will run concurrently, so the normal rules of thread synchronization apply. You can write shutdown hooks to do anything that a normal thread would do; the JVM will treat them like any other. Generally we write a shutdown hook to do any last-minute tidying, such as flushing memory buffers, closing files, or displaying an exit message. The JVM will always wait until all shutdown hooks have completed before continuing with the rest of the shutdown sequence, so it is important that your shutdown hooks do actually complete.

The following example shows how to write and install a Java shutdown hook.

```
class ExampleShutdownHook {
    public static void main(String[] args) {
        // Java code to install shutdown hook: MyShutdown
        MyShutdown sh = new MyShutdown();
        Runtime.getRuntime().addShutdownHook(sh);
    }
}

// Example shutdown hook class
class MyShutdown extends Thread {
    public void run() {
        System.out.println("MyShutdown hook called");
    }
}
```

If at any time during the application you decide the shutdown hook is no longer required, it can be removed with the `Runtime.removeShutdownHook()` method, (or `Runtime.getRuntime().removeShutdownHook(sh)`; for the above shutdown hook example).

Signals and Java dumps

What have Java dumps got to do with signals? If the JVM terminates abnormally, such as when a SIGSEGV condition occurs, a Java core dump will be generated. This dump contains useful information, including the reason for the termination, environment data, the state of each thread, and more.

A useful feature of the JVM is the ability for a Java core dump to be generated on demand. On a Windows platform you can do this by typing CTRL-BREAK, which sends a SIGBREAK signal to the application. On Unix type platforms you send a SIGQUIT signal to the application (this will usually be with CTRL-V or CTRL-\\, depending on the platform, or command: `kill -s SIGQUIT <process id>`). Java core dumps generated this way can be particularly useful when a lockup is suspected. A useful technique for debugging lockups is taking two Java core dumps, one immediately after the other, and looking for differences.

From a signal handling point of view, for Unix type platforms a Java core dump contains the list of platform signals and identifies the installed signal handler for each. (However, for z/OS only the library containing the signal handler is referenced.) If you have an application that uses signal handlers, then a Java core dump can be useful to determine which function (or library) is actually being used as the handler. For Java signal handlers the handler identified in a Java core dump should be the JVM's `intrDispatchMD()` function, from library `libhpi`.

Summary

I hope this article gave you useful insight into the signal handling and termination behavior of IBM's JVM 1.3.1. Knowing why JVM uses signals, and writing your own native signal handlers in a well-behaved manner that won't compromise JVM operation, can circumvent problems. Understanding Java application signal handlers and writing your own handler function is also useful. Using and writing termination hook functions called by the JVM's termination logic during normal and abnormal application shutdown helps with your last-minute tidying.

Downloads

Description	Name	Size	Download method
	SignalHandling.zip		HTTP

[Information about download methods](#)

About the author

Chris White

Chris White is a software engineer with 13 years of industry-wide development experience. He now works as a lead software engineer at the IBM Centre for Java Technology in Hursley, England, on core JVM technology for the z/OS platform. Contact Chris at Chris.White@uk.ibm.com.