“Too many open files...”? Not with JPicus Java I/O Analysis Framework

Applies to:

All Java based SAP and third party products and applications that run on Java SE 5 or 6.

For more information, visit the Java homepage.

Summary

Having I/O operations failing with a message like “Too many open files” is a sure symptom of leaking or exhausted file descriptors (aka file handles). Using the JPicus Java I/O analysis framework can greatly simplify the analysis of such problems and help you proof your software for the real world, before it goes to your customers.

Author: Pavel Genevski

Company: SAP AG

Created on: 3 June 2009

Author Bio

Pavel Genevski is a Java/C++ developer with more than 6 years of experience. Throughout his career, he has worked on various projects, ranging from software for microcontrollers to CRM and ERP solutions. Currently he is a Senior Java developer in the Technology Development Group at SAP, with responsibilities in the core server runtime and deployment.

Table of Contents

Introduction

Leaking or exhausted file descriptors (aka handles) is a relatively common problem that a lot of people are facing during the software development process. Sometimes such issues even make their way to the end users.

Symptoms

> In a Java application, an attempt to create a new I/O object like a FileInputStream or launch an external process with Runtime.getRuntime().exec() fails with a message like "Too many open files".

Diagnosis

This is a clear sign that the application process has reached its file descriptor limit.

Root causes

There are two common causes of this problem, leaking and exhausted file handles.

- **Leaking handles** – They normally occur because of coding mistakes or usage of components with unclear or not well understood contract. Leaks tend to accumulate in a long running application until the process limit is reached and the "Too many open files" message occurs. Since the process of accumulation is relatively slow, the problem might not manifest itself during unit or functional (black box) testing or even during load testing, but show up when your customers have run the application for several weeks or months.
- **Exhausted handles** – Even if you don't have leaking handles, your application might still experience the "Too many open files" problem. This usually happens under load, when the lifespan (age) of a handle is a lot bigger than the time, for which it is actually used. In difference with leaking file handles, the factor here is not the time for which the application has been up but the load (i.e. number of users simulatenously working with the application etc). Such problems are not likely to manifest themselves during unit or functional testing but shall be rather easy to reproduce with load testing (of the given kind).

Consequences

The consequences of both the problems might be very severe, because thei lead to downtime. In the case with leaking file handles, the only way to return such an application back to productive state is to restart it.

Leaking handles

Let's now have a more detailed look at the first problem. The most trivial example of a leaking file handle looks like this:

```
FileInputStream in = new FileInputStream("MyFile.txt");
// use the stream
```

You might be thinking that it only happens to newbies but once you start analyzing, you will be surprised how common this mistake is. Of course a better way to use files shall involve a finally block:

```
FileInputStream in = null;
try{
   in = new FileInputStream("leaking.file");
   // use the stream
} finally {
   try {
          if(in != null){
                in.close();
          }
   } catch (Exception e) {
          e.printStackTrace();
   }
}
```

If your application is fairly short and simple as in the example above, it would be very easy to spot such problems. However, productive applications are never simple. They may consist of thousands of classes with millions of lines of code and then the problem is nearly impossible to find by just looking at the code.

Finding the leak

Problems, like this one, are best solved with Java I/O analysis. With the help of the JPicus I/O analysis framework you can analyze your application and fix such problems in advance, before they go to customers. Enough talking. Let's see how do we get the problem solved.

Note: If you are new to JPicus, go to its home page (see the References section), follow the download and installation instructions and read the short "Getting started" section.

Let's run the above application with the JPicus agent attached and take a snapshot. Double-click on the snapshot to open its "Handles" view and look at the "Closed at" column.

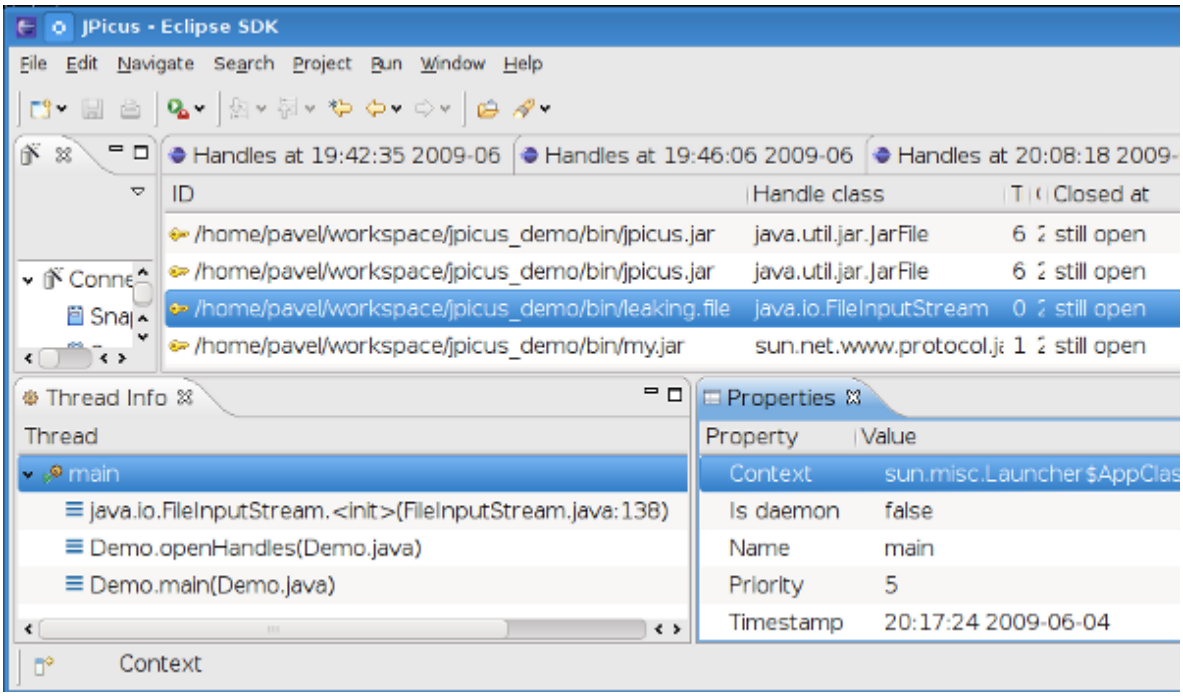

Figure 1 – a leaking file handle

You will see that for some handles the value is "still open". This means that the handle hadn't been closed at the time the snapshot was taken. You should be able to spot the leaking file handle from the example above in seconds. Now, if you select the handle and look down to the "Thread Info" view, you will see one thread - the "Opening thread". Selecting and expanding it reveals its stack trace and populates the "Properties view" with all the information available about this thread. Having this at your disposal, you can confidently identify the place in source code where this file was opened and replace the single line with a try-finally block in order to ensure that the handle is closed when not needed anymore.

When we get the things right with the finally block and repeat the analysis, one more thread shall be present in the thread info view - the "Closing thread". It shows where/when the handle was closed.

The information about the closing thread could be very useful if there is a leak because of improper synchronization. For example, imagine that you have opened 100 files and they were supposed to be closed after processing. It could happen that for some reason, some of them will remain open. In this case, you can click on one of the handles that were properly closed and identify where in code the handle is supposed to be closed. Then you can use this information in order to figure out why the processing of the other files did not reach section of code.

More examples of leaks

There are of course other ways to leak a handle the most common of which are:

- Crowded finally blocks
  - If an exception (including unchecked ones) is thrown within a finally block, before the close method of your handle is executed the handle won't be closed and will leak.
  - Keep an eye on the finally blocks and don't place there two possible sources of exceptions.
- Parsing XML files withing a ZIP of JAR archive
  - `If you are using code like: DocumentBuilder.parse("jar:file:///C:/file.jar!data.xml"); most Java Virtual Machines will cache the JarFile object and the handle will not be closed`

- Dont use this mechanism for more than a few files or in a large application, especially if you want to delete or move the files later on (while the application is running)

- Launching external processes
  - If you are using code like: Runtime.getRuntime().exec("external_process") and you don't explicitly close all the three streams (in, out and err) and call process.destroy() the handles (file descriptors) will leak.
  - Make sure that you obtain all the streams and call their close method like this:

```
Process process = null;
InputStream err = null;
InputStream in = null;
OutputStream err = null;
try{
   process = Runtime.getRuntime().exec("javac MyClass.java");
   err = process.getErrorStream();
   in = process.getInputStream();
   out = process.getOutputStream();
} finally {
   If(err != null){
         try{
               err.close();
         } catch (Exception e){
               e.printStackTrace();
         }
   }
   // do the same for the rest of the streams

   if(process != null) {
         process.destroy();
   }
}
```

Exhausted Handles

Even if you close your handles properly you might still not be utilizing their life efficiently. Such problems are even harder to investigate than leaking handles if you are using common techniques like debugging, profiling or code reviews. Again the answer here is I/O analysis. Let's have a look at another example:

```
ObjectInputStream in = null;
try{
   in = new ObjectInputStream( new FileInputStream("/home/pavel/my.ser"));
   MyObject obj = (MyObject) in.readObject();
   process(obj);


} catch(Exception e){
   e.printStackTrace();
} finally {
   if(in != null){
         try {
               in.close();
         } catch (IOException e) {
               e.printStackTrace();
         }
   }
}
```

What we are doing here is reading a serialized object out of a file, deserializing it and doing some processing. Now imagine that the process method occupies 95% of the overall processing time (e.g. you are inserting this data into a remote database). The problem here is that after we read the object in memory, the file input stream is not closed althought it not needed anymore. In a highly loaded environment this could easily lead to exhausted handlles. Let's run the application with JPicus attached and take a snapshot:

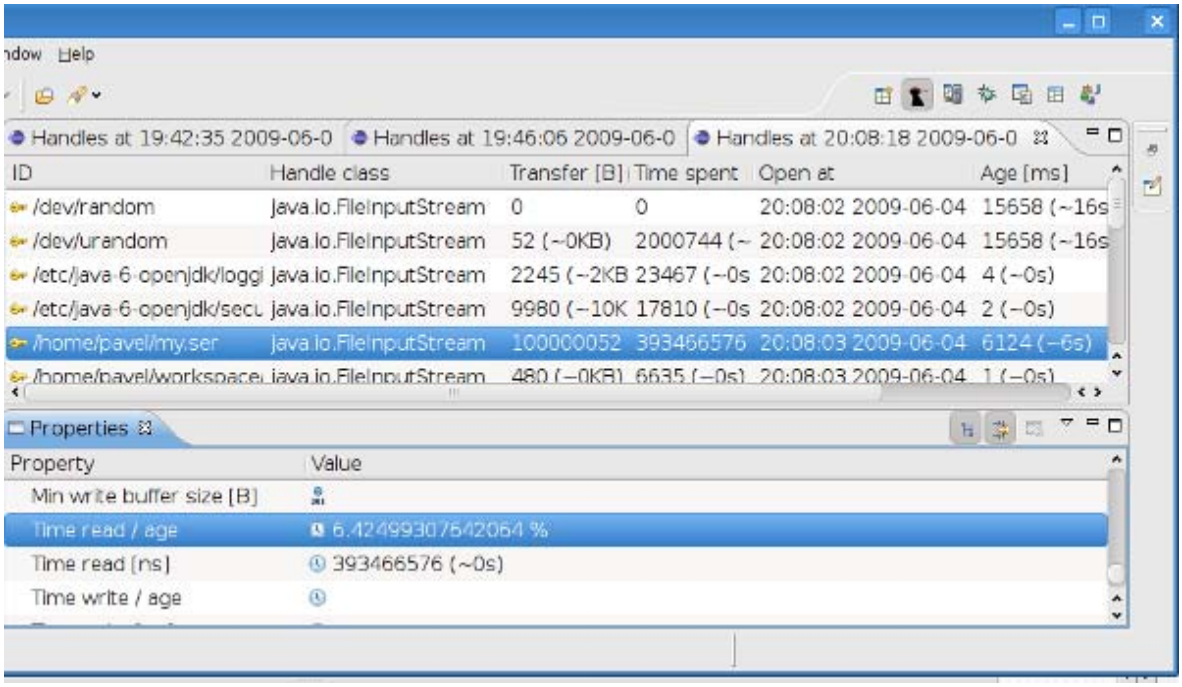

Figure 2 – inefficiently used file handle

A very straightforwared way to discover handles with suboptimal utilization is to have a look at one of their composite metrics, "**time read/age"** or **"time write/age"** respectively. This metric gives you the ratio of the time spent for reading or writing to the age of the handle. It is calculated in percents and presented in the advanced properties view. Handles that have this metric below 5% could turn out very good candidates for optimization.

Note: In order to see the time read/age in the properties view you have to enable the advanced properties by clicking the second button of the properties view.

Other sources of exhausted handles

- Applications that rely on heavy file indexing (like Derby DB)
  - If you have such a kind of an application it is probably worth spending some time in I/O analysis to see how it uses file handles and eventually tune it so that it matches your environment.
- Modular systems with lots of components
  - If you are using the Equinox OSGi runtime or some other modular software and you have lots of modules on top of it, it might lead to exhausted file handles. For example, Equinox does not close the bundle JAR files by default.
  - In order to limit the Equinox usage of file handles use the following Equinox initialization parameter: **osgi.bundlefile.limit=X** , where X is the value that you see fit for your working environment

Summary

The "Too many open files" problem is non-trivial to solve without Java I/O analysis. Once you are in such a situation and you start searching the web for a solution you will be surprised how many people have encountered this problem and how much time it took them to solve it. Now, with the help of JPicus, problems like this could be solved faster, better and with more confidence.

Related Content

JPicus wiki

For more information, visit the Java homepage.

Copyright