JNI Fault Tolerance Using Java ProcessBuilder

Yew Kwang Hooi and Alan Oxley

***Abstract*—An application can be crippled by the memory leakage of one of its components. Unfortunately, access to the source code of a referenced component, for rectification, is often not feasible. This paper presents our experience of using multi-processing as a strategy to contain the problem. We demonstrate the use of Java ProcessBuilder to protect applications from unstable native code accessed via the Java Native Interface. The technique discussed can help in designing applications that provide better fault tolerance without costing much memory utilization.**

***Index Terms*—Computer crashes, Java Virtual Machine, multi-processing, multi-threading.**

I. Introduction

Some applications have tasks that refer to native code. Native code can be found in, for example, Dynamic-Link Library (DLL) files. Native code is often written in a language such as C or C++. Since native code is specifically written for the host platform the performance is unquestionably superior to that of an application written in a portable language such as Java. Invoking native code extends some host-specific features and avoids unnecessary reinvention of functions that are already available [1].

Unfortunately, invoking native code has some disadvantages that may cripple the parent application. Therefore, it is imperative to implement fault-tolerance in an application having a native code invocation. This paper describes our experience of using Java ProcessBuilder to protect Java applications from errors caused by native code.

The following sections discuss the architecture of the Java Virtual Machine (JVM), stability issues of the Java Native Interface (JNI), memory leaks, multiprocessing and multithreading designs, the experimental setup and finally, the advantages and disadvantages of the technique that we propose to address the problem.

II. Java Virtual Machine (JVM)

A Java application runs on JVM, a stack-based machine that emulates a virtual processor and provides a layer of abstraction on which Java byte code runs. JVM provides platform portability and manages the native method stack.

JVM throws an OutOfMemoryError exception when a Java program is consuming more memory than is available. To address a large memory requirement, the memory size of JVM can be enlarged from the default 2MB capacity, at loading time. To address an unexpected increase at runtime, hedge memory can be pre-reserved and programmatically released when the exception is thrown [2]. However, this technique requires an overestimation of memory usage and this may be a waste of resource.

During execution, a runtime instance of Java application code is generated as a process with the main method being the initial running thread. A thread may execute one or more tasks in a sequential or concurrent manner. Another process, a new sub-process or a new thread may be created and added as necessary. A new sub-process requires the allocation of resources (CPU time, memory, files, I/O devices) over and above that of the parent process. In Java, an operating system process can be created using ProcessBuilder. Alternatively, a new thread may be spawned to run a new task. Use of new threads, i.e. multi-threading, is sometimes preferred because resources are being shared, resulting in better data exchange and faster context switching [3].

III. Native Code Invocation

Java Native Interface (JNI) wraps native code so that the latter can be used like a Java object [4]. As depicted in Fig. 1, the functionalities of a native stack are not directly accessed but accessed through the Java method stack that functions as a mediator. However, there are several disadvantages of invoking native code from a Java application [5]. These disadvantages will now be described

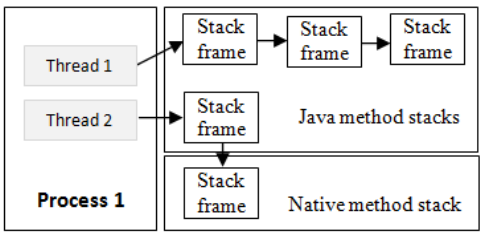

Fig. 1. Multi-threaded invocation of native method stacks.

A. Poor stability

Careless usage of JNI may cause the application to perform poorly leading to instability [6].

B. Reduced portability

JNI reduces application portability because the native code is not portable and may not be available on another host [6];

C. Lacking garbage collection mechanism

JNI and most native code do not have a garbage collection mechanism for automatic recycling of unused dynamically allocated memory;

Manuscript received September 16th, 2011;September 29, 2011.

Yew Kwang Hooi is with Dept. of Computer and Information Sciences, Uni. Teknologi PETRONAS, Bandar Seri Iskandar, 31750 Tronoh, Perak, Malaysia (yewkwanghooi@petronas.com.my).

Alan Oxley is with Dept. of Computer and Information Sciences, Uni. Teknologi PETRONAS, Bandar Seri Iskandar, 31750 Tronoh, Perak, Malaysia (alanoxley@petronas.com.my).

D. Lacking exception controls

JNI does not catch and control exceptions or errors generated by the native code. An undetected memory leak may grow in size and ultimately cripple any application. Hence, JNI may disrupt application stability.

IV. Memory Leaks

A process that is causing a memory leak may produce a saw tooth pattern of memory utilization [7]. Left unattended, the memory leak may eventually cripple the application. The host system usually finds itself incapable of handling the silent memory leak. Following are reasons for the occurrence of a memory leak [8]:

A. Data cancer

Due to negligence in cleaning up unused Java references.

B. Incomplete disposal of unused memory

Due to negligence in disposing of unused memory.

C. Bad finalizer

Wrong implementation of the finalizer causes many pending objects to be left.

When a shortage of memory occurs, as a recovery measure the host system may randomly, or selectively, remove processes in order to free up some of the memory. Selective removal assumes that the largest process is responsible for the leak, whether or not it is the actual cause of the leak.

A preventive measure, unlike recovery, does not clean-up after a process has failed but, rather, prevents failure in the first place. A good mechanism for preventing memory leaks involves hosts pre-allocating a memory limit to each process in order to restrict dynamic memory usage to a single block of memory during application initialization [9].

V. Processes and Threads

Processes and threads are the basic units of execution. A process is an active entity spawned from a program during runtime, whose instructions are being executed sequentially in the CPU until completion [3]. It has a self-contained execution environment, i.e. a private set of basic run-time resources and memory space. Processes are generally not allowed to access one another's storage despite sharing underlying computational resources (CPUs, memory, I/O channels). “A machine crash caused by one process often kills all other processes” and brings the program to an immature termination [10].

An application must have at least one or more cooperating processes. Cooperating processes communicate through Inter-Process Communication (IPC), i.e. pipes and sockets. Reasons for providing an environment that allows process cooperation are as follows:

- Information sharing of the same piece of information.
- Computation speedup by breaking a task to several subtasks.
- Modularity by dividing system functions into separate processes or threads.
- To allow multiple task processing.

Java applications contain only a single process at startup. Depending on its design, the main process may create several sub-processes during its runtime, see Fig. 2. For example, a new process can be forked using a ProcessBuilder object. The path of the new process, i.e. an external byte code to be executed, can be passed as an argument to the ProcessBuilder object during its instantiation.

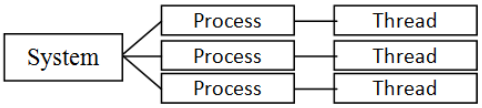

Fig. 2. Multi-processing.

When a sub-process is created, resources (CPU time, memory, files, I/O devices) are allocated from the operating system or the parent process. Each process or sub-process may contain one or more jobs. Each job is a code segment (algorithm) plus data. Some of the jobs residing in the same process must be performed sequentially whilst some are best done concurrently.

Multi-threading is the use of two or more threads within a process, see Fig. 3. It is a common technique to hide latency by switching execution from one thread to another in order to let the CPU perform useful work while waiting for the pending requests to be processed in the main memory. A new thread requires fewer resources to create than a new sub-process and provides better context switching [3]. Threads belonging to the same process share a code section, a data section and operating system resources such as memory and files. Sharing of resources facilitates data exchange and enhances performance. However, multithreading requires a few tradeoffs:

- Sharing of resources eliminates total autonomy. Memory pool has to be shared among threads.
- Unsynchronized threads may be potentially problematic due to subtle and non-deterministic interactions. Cheap scheduling policies lead to thread competition and possible starvation. Hence, reliability of a multithreaded program can only be determined partially.
- Memory-based synchronization prevents thread starvation but incurs programming complexity and greater chances of a programming error [10].

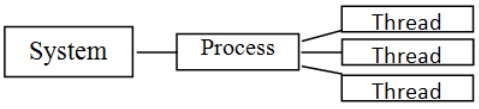

Fig. 3. Multi-threading.

VI. Benchmarking Memory

Benchmarking is a way of providing consistency among evaluations for comparison purposes. Kazi [11] states that the lack of a standardized set of Java benchmark programs makes it difficult to evaluate the performance of various execution techniques. Java benchmarks such as CaffeineMark 3.0 (an ‘instructions per second’ benchmark), Jmark (a ‘multi-threading performance’ benchmark), Volanomark (a ‘messages transferred per second’ benchmark) and Symantec are designed to test specific features of a JVM implementation, not the performance of a JVM as a whole.

Static checkers such as Calvin, developed by Flanagan et al. [12], is a tool to catch elusive timing-dependent bugs in multi-threading systems, including synchronization error and violation of data invariants.

Despite the tools, none seemed to fit our benchmarking requirements. Hence, we devised our own measurement using a Java System class static method, the NetBeans Profiler and the Windows Task Manager performance indicator.

VII. Methodology

Our work entailed conducting a series of experiments. A Java desktop application was written and tested on a host computer. The computer has a single processor, an Intel Core 2 Duo 2.99 GHz, and has1.93 GB RAM of primary memory. The JVM is Java 2 Platform Standard Edition version 1.5.0 JVM and the operating system is Windows XP v2002 SP3.

The application contains two tasks. The first task contains a native DLL file referenced via a JNI wrapper. In this task, the DLL file is used to access a Universal Serial Bus device, retrieve a string value and pass it to the main method. The second task uses standard Java Swing packages to create, assemble and display Graphical User Interface (GUI) containers, widgets, events, handlers and images.

The following tests were conducted independently:

a. Run both tasks sequentially in the main method.

b. Run both tasks concurrently using multi-threading.

c. Run both tasks concurrently using multi-processing, with the first task running in a forked process implemented by ProcessBuilder, see Fig. 4.

```
//Task 1:
String[] command ={"java","-jar","jniWrapper.jar"};
ProcessBuilder pb = new ProcessBuilder(command);
Process p = pb.start();
//Begin task 2 here
//Join task 1 and task 2 here
```

Fig. 4. ProcessBuilder for multiprocessing in Java.

Both tasks involve consuming a considerable amount of memory and having an extensive loading time. For each of the above tests, both loading time and runtime stability were assessed.

To measure the first aspect, i.e. loading time, time stamps were inserted at strategic points in the code; see Fig. 5 and Fig. 6. An elapsed time was determined by finding the difference in the time stamps. To evaluate the second aspect, i.e. stability, the application was left running for 12 hours or until a crash occurred.

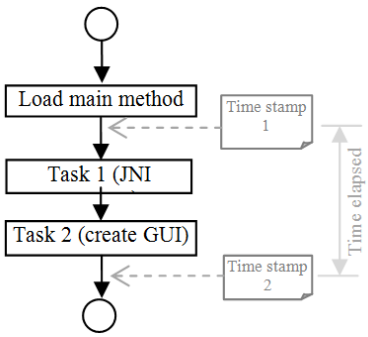

Fig. 5. Measuring elapsed time in sequential tasks.

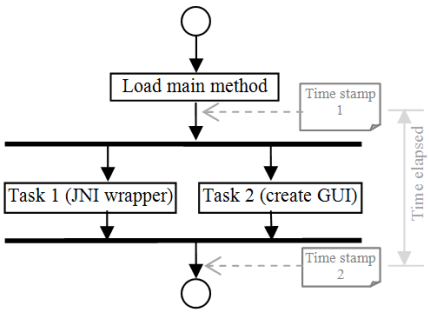

Fig. 6. Measuring elapsed time in concurrent tasks.

VIII. Results and Discussion

A. Application performance and stability.

The tests were carefully repeated several times and the findings are summarized in Table I. Running task 1 and task 2 individually served as control experiments.

TABLE I: Summary of Findings

Implementation	Mean loading time	Crash	Running time
Task 1 only	3.6 sec	Yes	10 mins
Task 2 only	0.8 sec	No	> 12 hours
Sequential	>4 sec	Yes	<4 mins
Multi-threading	3.2 sec	Yes	<1 mins
Multi-processing	3.3 sec	No	>12 hours

In the sequential implementation, the findings indicate that the application is slow, unstable and crash-prone. To verify which of the two tasks was responsible for the crash, we ran them separately. The result indicates the task using the JNI method to be the cause. It violated access of a problematic code frame. Inspection of its profile indicates a possible memory leak, see Fig. 7. A saw tooth memory utilization pattern is a common indicator of a memory leak. Fig. 8 depicts the pattern of normal heap memory consumption, which was observed in the other task, i.e. instantiation and loading of GUIs.

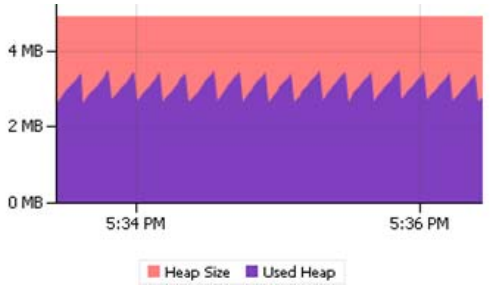

Fig. 7. Saw tooth utilization of heap memory in task 1.

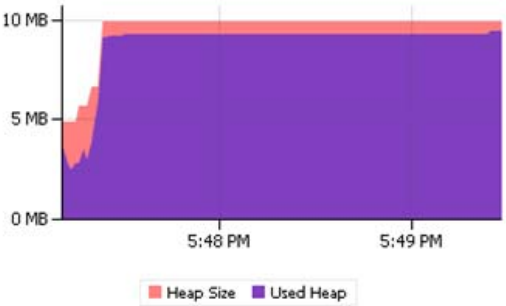

Fig. 8. Normal heap memory performance in task 2.

However, a saw tooth pattern can also be the result of an active garbage collection [7]. Further inspection indicates the absence of the garbage collection activity in task 1; see plot for "Relative Time Spent in GC" in Fig. 9. This confirms that the saw tooth pattern is not caused by garbage collection but by a possible memory leak. Consequently, the surviving generations of objects, indicated by the "Surviving Generations" plot, are increasing and may eventually consume all of the allocated memory. Garbage collection in task 2, as indicated by the short curve on the X axis in Fig. 10, ensures that the number of surviving generations of objects is kept at a constant.

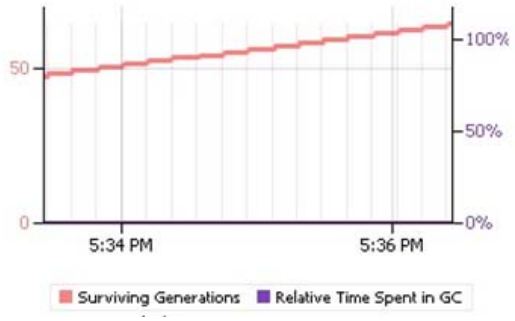

Fig. 9. Surviving objects without garbage collection.

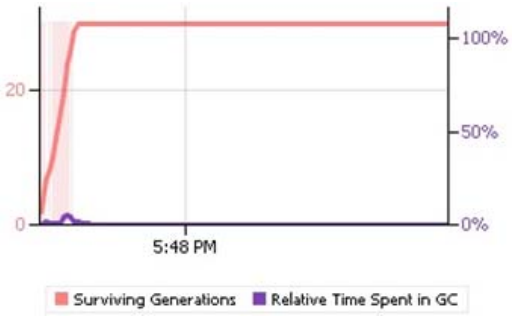

Fig. 10. Surviving objects with garbage collection.

Unfortunately, accessing native code for the rectification of a memory leak is not always possible. Therefore, we investigated the possible use of multi-threading or multi-processing in order to isolate the problematic task from the healthy one.

Using multi-threading, the loading time is reduced by 28% but an unexpected result is that the program becomes even more prone to crashing, as evidenced by a shorter running time. Memory consumption within JVM revealed a relatively sluggish garbage collection and high memory consumption.

Using multi-processing produces a significant improvement. When a new process is started, a new application state is initialized by copying values from the current application, including standard streams and 'running user' [13]. We have successfully shown that multi-processing is capable of isolating unstable native code by having separate memory stacks in a new process; see Fig. 11. Should the native code crash in its own native memory stack, it will not propagate into the Java method stack memory. The limitation however, is that since separate processes do not share memory blocks, communication between tasks is not direct but through message passing using input/output streams. Hence, this may not be feasible for achieving a high performance if both tasks need to communicate very frequently or to share large amounts of data.

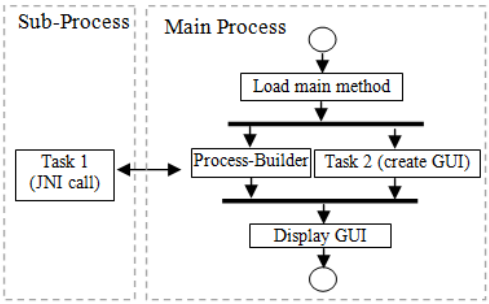

Fig. 11. Proposed multi-processing using Java ProcessBuilder.

B. Effect on the physical memory of the host

In multithreading, the used heap keeps growing in size until it reaches 10MB, see Fig. 12. In multiprocessing, however, the size of the used heap remains below 10MB, at about 8MB; see Fig. 13.

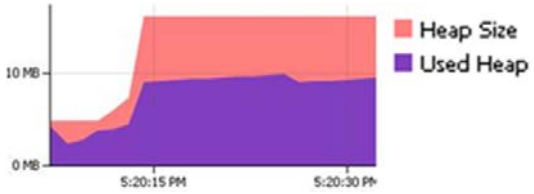

Fig. 12. Multithreading of both task 1 and task 2.

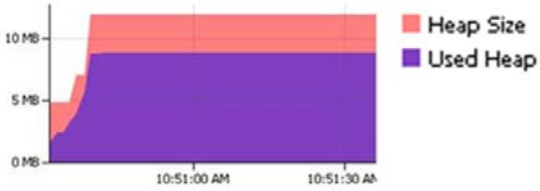

Fig. 13. Multiprocessing of both task 1 and task 2.

Although multi-processing does not seem to require more heap memory than multi-threading, there is a possibility that more memory has been allocated outside of the JVM for running the native code. The profiling utility used does not give information on host memory consumption. Thus, a system tool such as Windows Task Manager was used to inspect the amount of physical memory consumed before and after running the experiments.

The results in Table II indicate that the use of ProcessBuilder does not incur more physical memory than multi-threading. Surprisingly, the amount of memory utilized when running the tasks concurrently is not much different from running the task individually. This is probably due to the small memory footprint of the JNI task, making the difference not significant. Furthermore, usage of the Task Manager may be prone to inaccuracy due to there being many other background programs running and so it is not able to exclusively measure the application.

The results were obtained from tests conducted on Windows XP SP2. Results from tests conducted on Windows Vista and Windows 7 have, however, been deliberately omitted. This is because the application tested on both platforms did not crash immediately, although similar patterns of memory consumption and loading behaviors were

observed. As the intent of this paper is to show that ProcessBuilder is a good strategy to prevent instability leading to a crash, we chose to display test results from the least stable platform.

These findings are useful because sometimes an application may need to access a native library function on the host computer and this activity might be unstable, causing memory leakage and thus bringing down the application. As with 'exceptions throwing', the errors caused by a referenced library should be managed so that if necessary the application can either rollback the process to a previous state or gracefully shut itself down.

Multi-processing requires the operating system of the host computer to allocate resources that are separate from the bulk of the application, which is beneficial to the application because any memory leakage or other problem is then isolated from the main part of the application. In Java, this can be done using the ProcessBuilder class which can be easily coded using Threads and Runnables. Any library file, not necessarily wrapped in JNI, can be run as a task. The drawback is the lack of communication between concurrent tasks in separate processes, which is important to applications with plenty of message passing, synchronization and coordination.

TABLE II: PHYSICAL MEMORY CONSUMPTION ON HOST

No.	Scenario	% memory consumed
1	GUI only	3.59%
2	JNI only	2.13%
3	Sequential	3.75%
4	Multithreading	3.69%
5	Multiprocessing	3.57%

IX. CONCLUSION

Our experience shows that failing native code can be tolerated in a Java application if the task can be isolated as a separate sub-process. A carefully written Java program may not be immune from buggy native code unless rectification has been undertaken on the native source code. We introduce a more convenient solution, the use of Java ProcessBuilder to isolate unstable native code.

ACKNOWLEDGMENT

The authors would like to thank the management of Universiti Teknologi PETRONAS for all forms of support.

REFERENCES

[1] S. Liang. *Role of the JNI in The Java Native Interface*. Prentice Hall, 1999. Available: http://java.sun.com/docs/books/jni/html/intro.html#1811

[2] J. T. Boyland, "Handling Out of Memory Errors," presented at the Proceedings of ECOOP'05 Workshop on Exception Handling in Object-Oriented Systems : Developing Systems that Handle Exceptions, Glasgow, 2005.

[3] Silberschatz, Galvin, Gagne "*Operating System Concepts with Java, 6th edition*." San Francisco, CA: John Wiley and Sons, 2004, p. 144.

[4] B. Venners, *Inside the Java 2 Virtual Machine*, 2nd ed. Columbus, OH: Computing McGraw-Hill, 1999.

[5] (2010, 9th January). *Java Native Interface*. Available: http:// en.wikipedia.org/wiki/Java_Native _Interface #cite_note-role-0

[6] M. Dawson, G. Johnson, A. Low (2009, July 7, 2010). *Best practices for using the Java Native Interface*. Available: http://www.ibm.com/developerworks/java/library/j-jni/

[7] (2011, April 2nd). Memory Leak. Available: http://en.wikipedia.org/wiki/Memory_leak

[8] (November 29, 2010). *Java Memory Leak*. Available: http://jb2works.com/memoryleak /index.html

[9] N. Hillary, *Measuring Performance for Real-Time Systems*. Denver, Colorado: Freescale Semiconductor, 2005.

[10] Lea D., "*Concurrent Programming in Java: Design Principles and Patterns*" Cambridge, MA: Addison-Wesley, 1996, p. 23.

[11] Kazi I.H., Chen H.H., Stanley B., Lilja D.J., Techniques for obtaining high performance in Java programs. ACM Comput. Surv. 32, (2000), pp. 213-240.

[12] Flanagan C., Freund S.N., Qadeer S., Seshia, S.A.: Modular verification of multithreaded programs. Theor. Comput. Sci. vol. 338, (2005), pp. 153-183.

[13] D. Balfanz and L. Gong, "Experience with Secure Multi-Processing in Java," presented at the Proceedings of the The 18th International Conference on Distributed Computing Systems, 1998.

Yew Kwang Hooi graduated in 2006 with a MEng in Computer Science from Cornell University, New York. In his earlier years, he had worked at IBM Malaysia as a product engineer and had worked on projects at the Federal Bank and for a Malaysian banking portal. He led a software tool development for electrical safety review for Group Technology Services, Petroleum Nasional Berhad for 2 years. He has conducted a few programming workshops for the local state government. At present, he is a lecturer in the Information and Communication Technology programme at Faculty of Science and Technology, Universiti Teknologi PETRONAS, Malaysia. His research interest includes software engineering, computer graphics and data mining. He is a certified programmer for Java platform, SE6. This author is a member of IEEE Computer Society and ACM Member.

Alan Oxley is a professor at Universiti Teknologi PETRONAS in Perak, Malaysia. He has received the following degrees from UK universities: a B.Sc. from City University, London; an M.Sc. from Cranfield Institute of Technology; a Ph.D. from Lancaster University; a PGCTLHE from London Guildhall University. His research interests are wide-ranging. He is currently looking into a fundamental problem in bioinformatics.