Una comparación del desempeño para acceder programáticamente recursos del sistema en Linux

A comparison of performance to access programatically system resources in Linux

John Willian Branch. Ph.D. & Sergio Armando Gutiérrez. I.S.
Escuela de Sistemas, Facultad de Minas, Universidad Nacional de Colombia
jwbranch@unal.edu.co, saguti@unal.edu.co

Recibido para revisión 10 de agosto de 2010, aceptado 03 de enero de 2011, versión final 02 de febrero de 2011

***Resumen*— Este artículo presenta un análisis comparativo de tres aproximaciones que se pueden emplear para acceder a recursos del sistema bajo el sistema operativo Linux. Luego de una revisión de las principales características de cada aproximación, se realiza un experimento donde el comportamiento de estos es evaluado. Finalmente, los resultados son comparados y analizados.**

***Palabras Clave*— Desempeño de llamadas al Sistema, Linux, Java, JNI, Python.**

***Abstract*— This paper presents a comparative Analysis of three approaches which can be used to access computer system resources. After a review of the main features of both of the approaches, an experiment is performed where behavior of them is evaluated. Finally, results are compared an analyzed.**

***Keywords*— System call performance, Linux, JNI, Python.**

I. INTRODUCTION

In monitoring of system resources, the performance of the monitor component is quite important as under particular circumstances, if monitoring is not properly performed, it might cause degradation or lead to incorrect data.

Oftenly, monitoring tasks is performed without taking good care of their performance or their impact on the system.

This paper presents the analysis of three methods to access information about filesystems on a Linux system, from the perspective of the time they take to be executed, and the impact of them on the system related to System Calls they invoke when executed.

For the analysis herein explained, concepts and tools presented in Dienelt [1] are applied, by using Tools which are available in the standard operating system.

This paper is organized as follows: Section 1 will introduce the approaches to access the resource in the system. Section 2 will describe the experiment which was performed to analyze the behavior of method, and the scenario where it was performed. Section 3 will present the conclusions of the analysis and future work to be performed on this matter.

II. THEORETICAL FRAMEWORK

This section will introduce the approaches which were used to access information of an Operating System Resource.

Basically, the three approaches employ API (Application Programming Interfaces) which allow to request information about the space in a filesystem of a computer running Linux operating system.

2.1 Pure JAVA Approach.

JAVA programming language was developed on 1995 by James Gosling and nowadays has become the second most important programming language in industry [2].

JAVA has five main design principles [3]:

• It should be "simple, object oriented, and familiar": JAVA requires that programming style be object oriented, although it also allows some features of procedural programming in the code. It is familiar as its syntax is very similar to C and C++ programming languages, and it is in a certain way easy to use.

Revista Avances en Sistemas e Informática, Vol.8 No.1, marzo de 2011 - Medellín. ISSN 1657-7663

• It should be "robust and secure": JAVA has several features which increase robustness, stability and security. Virtual Machine, the virtual processor where JAVA code runs is an abstraction layer which isolates in certain way the applications from the host hardware, increasing security by imposing restrictions regarding system resources access. Also, the way JAVA handles memory, and the way memory is referenced from code avoids stability issues which are typical in languages as C or C++.

• It should be "architecture neutral and portable": This is one of the features which has contributed to make JAVA a very important tool in application development. Code is written to run on Virtual Machine, and Virtual Machine is the same, despite of the hardware platform where it runs. It is an abstraction layer which isolates the hardware details from application code.

• It should execute with "high performance": Concepts like bytecodes, and Garbage Collection allow JAVA applications to perform in a very acceptable way. Multithreading and mechanisms for interprocess communicaction allow to applications take advantage of multiple processors, boosting the performance for every kind of application.

• It should be "interpreted, threaded, and dynamic": JAVA can be thought as an intermediate step between interpreted and compiled language. The compiler generates bytecodes to be run on Virtual Machine, in an interpreted way. The virtual machine is implemented in such a way that components within it can run in paralallel, because of their nature of independent threads. In certain way, entities within JAVA applications can be thought as really live entities, which born, reproduce and die during application life.

For the purpose of the feature under study on this paper, pure JAVA can be consideres as limited in certain way. Because of the principles above exposed, JAVA does not provide extensive APIs to access system resources at low level. Traditionally standard releases of JAVA Development Kit (JDK) -latest is 1.6.X-, have limited interfaces to access properties of system components as devices or filesystems. New releases as JDK 1.7.X [4] include a wider set of API's to access system at lower level.

2.2 JAVA Native Interface

JAVA Native Interface (JNI) is a framework which allows to JAVA programms interact with native applications, that is to say, code written in C, C++ or Assembler which is specific to hardware platform where JAVA Virtual Machine is running [5]. This can be considered as a good mechanism which extends JAVA applications, so that the be able to access using lower level interfaces the elements of the platform where it runs.

Although, it might seem a transgresion of the design principles of JAVA, it is worth to mention that JNI requires a very strict protocol to allow the interaction of JAVA code with Native Code, although, under certain conditions, and if it is not well programed, might destabilize the entire JVM.

There are two main implications related to the use of JNI:

• Applications depending on JNI are no longer fully portable. Although JAVA part of the code is still portable, the native code will need to be at least recompiled, or even ported when application is being migrated.

• The native code in general is not type safe neither has mechanism to protect memory access and memory referentiation. Mechanisms as pointers in C and C++, despite of their utility, are the main cause of unstability of applications. So, additional care has to be taken when integrating native code with JAVA through JNI as the errors in native code affect and impact the behavior of Virtual Machine.

Because of these issues above mentioned, there are some recommendations regarding when to use JNI, and how to use it to take advantage of its features:

• In situations where it is required JAVA perform tasks which are host dependent, and is not desirable to delegate these tasks to another process.

• Whether a native library is required, and it is not desirable to have the overhead of copying library, JNI can be used to access the library.

• To reduce the need to span multiple process to execute tasks.

• In cases where is desirable to get higher performance, by implementing a smal part of the code in native language, for example, to take advantage of specialized hardware (Encryption processors, Accelerator Video Cards).

It is a design recomendation when using JNI, separate the classes which execute native code, from the classes which are pure JAVA; use it in as few classes as possible, and in the most specific possible way.

2.3 Python

Python is a general purpose language, implemented on 1991 by Guido van Rossum [6]. It is a generic language, extremely portable and efficient enough for multiple applications. It has the flexibility of Perl, associated with the numerical power and ease to use of MATLAB, but available as an open source environment. The source code is generally small when compared to compiled languages by several reasons: high-level data types and operations, dynamic typing, automatic memory management, and command blocks delimited by identation [7, 8].

Many different institutions and technology vendors make extensive use of Python for a wide variety of applications ranging from hardware testing, web searching, peer to peer networking, cryptography, mobile applications, and others.

Python is usually applied for Systems Programming, given its wide support for usual Operating Systems, and the easeness to access system resources. Also for programming GUI interfaces, Internet Scripting, Integration and interfacing, Database access, and numeric and scientific processing [9].

Also features as Object orientation, freeness to use, easeness to use, its capability to be mixed with other languages, and the clearness of its syntax are causing python becomes one of the ten most used and important programming languages [2].

III. EXPERIMENTS AND RESULTS

This section will describe the tests performed to compare the behavior of the three above explained approaches to request information of system resources.

Two experiments were performed. On the first one, three programs, each one written in Pure Java, JNI and Python were executed five consecutive times, and measured by means of the strace utility, which is built in with Linux Operating System [1]. On the other experiment, the programs were run during one minute, being executed every minute.

Table 1 summarizes the specifications of hardware and software which was involved in this test. Tables 2, 3 and 4 illustrate the average of measurements with strace for the case of a single run. Tables 5, 6 and 7 show the results for repeatitive execution within a loop.

Table 1: Specifications of test hardware and software

Processor	Intel(R) Core(TM)2 Duo CPU T7250 @ 2.00GHz
Memory	2 GBytes
Kernel Version	2.6.32-22-generic (i686)
JAVA Version	OpenJDK version 1.6.0_18
Python Version	Python 2.6.5

Table 2: Pure JAVA Results

Total Seconds	0.06
Slowest System Call	futex
Most Called System Call	open

Table 3: JNI Results

Total Seconds	0.01
Slowest System Call	futex
Most Called System Call	open

Table 4: Python Results

Total Seconds	0.00088
Slowest System Call	futex
Most Called System Call	open

From the results can be observed that the approach which presents the lowest execution time is Python. This can be explained from the fact that python does not require the futex system call, which implies an active waiting, which adds a considerable delay to execution.

Table 5: Pure JAVA Results on loop execution

Total Seconds	0.06160
Slowest System Call	futex
Most Called System Call	open

Table 6: JNI Results on loop execution

Total Seconds	0.02
Slowest System Call	futex
Most Called System Call	open

Table 7: Python Results on loop execution

Total Seconds	0.00
Slowest System Call	futex
Most Called System Call	open

When performing loop execution, Python still exhibits a better behaviour than other two schemes. It shows that python can be considered as a good stating point for the development of monitoring tools which query information at high level from the system resources.

IV. CONCLUSIONS AND FUTURE WORK

In this work, a comparison among pure JAVA, JNI and Python when querying information from a system resource as a filesystem has been performed. After reviewing results, a

first conclusion can be obtained, the good behavior of Python, which exhibits lowest execution times and even reduced times in the lowest system calls.

Further work which can be performed is comparing the results on other test beds, querying another resources, and assessing other metrics as memory consumption, interrupts and other.

BIBLIOGRAPHY

[1] S. Dienelt, "Profiling linux system call activity," Master's thesis, Chemnitz University of Technology, 2006.
[2] TIOBE, TIOBE Programming Community Index, http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html.
[3] Sun Microsystems INC, The Java Language Environment, http://java.sun.com/docs/white/langenv/Intro.doc2.html.
[4] Sun Microsystems INC, Java Development Kit release 7, http://java.sun.com/javase/7/webnotes/index.html.
[5] Java Native Interface Specification, http://java.sun.com/docs/books/jni/html/jniTOC.html.
[6] General Python FAQ, http://www.python.org/doc/faq/general/.
[7] R. Lotufo, R. Machado, and A. Saude, A. Silva, "Toolbox of image processing for numerical python," in Proceedings of XIV Brazilian Symposium on Computer Graphics and Image Processing, 2001, 2001.
[8] R. Lotufo, R. C. Machado, A. Saude, and A. G. Silva, "Toolbox of image processing using the python language," in Proceedings. 2003 International Conference on Image Processing, 2003. ICIP 2003., 2003.
[9] M. Lutz, Learning Python. O'Reilly, 2007.