Tracing ext3 file system operations in the QEMU emulator

V.M. Stepanov <vladislav.stepanov@ispras.ru>
P.M. Dovgalyuk <pavel.dovgaluk@ispras.ru>
D.N. Poletaev <poletaev@ispras.ru>
Yaroslav-the-Wise Novgorod State University,
11 Lasarevskaya Street, Velikiy Novgorod, Russia, 173000

Abstract. The paper proposes an approach to monitoring file operations through capturing virtual disk accesses in the emulator. This method allows obtaining information about file operations in the OS-agnostic manner but requires a separate implementation for each file system. An important problem for implementing this approach is the correct handling of changes in the file system. Operating systems that cache write requests can perform operations in any order. The authors have created a method for detecting read, write, create, delete and rename operations, and a module for QEMU, which monitors operations in the ext3 file system. The advantage of this method over others is that it does not interfere with the operation of the OS and does not depend on it. It is assumed that the QEMU module for file systems other than ext2/3 can be implemented using the methods described in this article.

Keywords: virtual machines; file systems; monitoring; QEMU; introspection

DOI: 10.15514/ISPRAS-2016-30(5)-6

For citation: Stepanov V.M., Dovgalyuk P.M., Poletaev D.N. Tracing ext3 file system operations in the QEMU emulator. Trudy ISP RAN/Proc. ISP RAS, vol. 30, issue 5, 2018. pp. 101-108. DOI: 10.15514/ISPRAS-2018-30(5)-6

1. Introduction

The task of monitoring file operations is relevant when debugging the OS and its file system drivers, as well as researching the behavior of systems with an unknown internal organization, particularly performing the security audit of the information processed by such systems. The essence of the task is to display the actions and the names of the files with which the operations are performed.

Current solutions for file system monitoring are typically based on using the tools of the operating system and tracing system calls. These solutions differ depending on the operating system, and some exotic OSes might not have the appropriate tools for this task.

The approach described in this article does not require any knowledge about the operating system used. The information about file system operations is obtained through capturing the disk requests of the virtual machine. The implementation that has been created is based on the QEMU emulator [1]. By modifying the source code of the project, functionality has been added to monitor and log the file operations of the system.

A data read or write query contains the disk sector number and the number of bytes to read or write. The QEMU module identifies the file names based on the sector in the query, the virtual disk information and the knowledge about the structure of the file system. Every file system type has its own distinctive internal organization different from others and thus requires its own implementation of the module. As an example, a monitoring tool for the ext3 file system [2] has been implemented, which is one of the file systems used in Linux-based operating systems.

As a result of this project, a module was created to monitor file operations of any guest OS, but only if it uses an ext3 file system. It is expected that the ideas used in this implementation can be applied to other file systems as well.

2. Overview of existing solutions

First, the authors would like to review several tools for file system monitoring.

Inotify is a Linux kernel subsystem intended for monitoring file system events [3]. This mechanism can be used to monitor such file operations as reading, creating, deleting, changing files, etc., by subscribing to events. The application creates an inotify object and informs the kernel about the files needed. The kernel responds by sending notifications, which can be received by the application by reading the file. Users can monitor the activity of the file system by using command-line utilities from the inotify-tools library.

Other operating systems have similar mechanisms. For example, Windows uses FileSystemWatcher [4]. FreeBSD and Mac OS X allow monitoring changes using kqueue [5].

QEMU-Based Framework for Non-intrusive Virtual Machine Instrumentation and Introspection [6] is a system that uses a binary application interface to analyze the state of the virtual machine. The system includes a file monitoring plugin, which receives information about file operations by capturing the corresponding system functions. Since these system calls are different for different operating systems, a specific plugin is created for each operating system. Currently, file monitoring is implemented for Windows and Linux. Unlike the preceding mechanisms, this tool allows monitoring the file operations of the virtual machine without interfering with the guest operating system processes.

The proposed approach, like the plugin described above, does not require modifying the operating system. The difference is that the implementation of this approach does not have any dependencies on the operating system. Instead, it depends on the file system.

One of the possible use cases of this project implies monitoring file operations in exotic file systems where information about system calls is not present or which do not have system calls in their traditional sense because the whole system operates in a privileged mode. The information about what files are being used in such an operating system can be useful for analysis.

3. Ext3 file system

The authors will now briefly describe the structure of the ext3 file system, which the tool is intended for. The space of the file system is divided into fixed-size blocks. For the purposes of optimization, the blocks are combined into groups. Each group has a description block, bit masks, and an inode table (fig. 1).

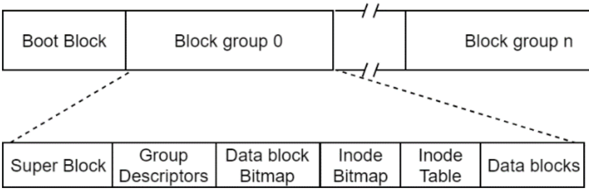

Fig. 1. Ext3 file system structure

An inode is a structure which contains the addresses of all blocks of the file, as well as its attributes, such as the file type and access permissions. The name of the file is contained in a separate structure – the parent directory. The directory contains a table, in which each of the entries represents a child file and includes the name and number of the respective inode.

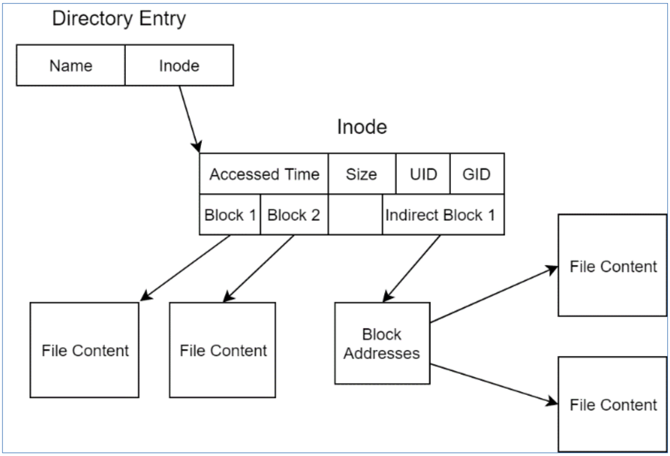

Fig. 2. General information structure of the ext3 file system: directories, inodes, and data blocks

The physical location of the file is represented as data block numbers in the inode. If any inode does not have sufficient space to store all the addresses of the data blocks, references to additional address blocks are used. Address block entries can refer to other address blocks, and so on. In this case, such address blocks are called double and triple indirect blocks. The interconnections of the described file structures are depicted in fig. 2.

When opening a file, the OS uses the path name to access the data. The directory system converts the ASCII name into information needed to find the data. However, for the purposes of monitoring file operations, the authors are interested in the reverse process. A method is needed that will make it possible, by using a specific sector number, to obtain the name of the corresponding file.

4. Possible ways of finding the file name based on its sector number

The task to find the full file name based on one of its sector numbers has several possible solutions.

The first way implies that every disk operation should be accompanied by iterating through all the files and the addresses of their data blocks until the file with the particular sector is found. The time to complete such iteration is, in the worst case, linearly dependent on the space used in the partition.

Implementing this solution showed that for the 6.0 GB disk with 4.0 GB used, in an ext3 file system, one such disk query may take up to 12 seconds. This time measurement was performed on a computer configuration using an AMD FX-8370E processor and 16 GB of RAM. Thus, it may be concluded that to achieve high performance, the number of full file system iterations should be minimized.

Another way to do this implies creating special data structures with the aim to achieve higher searching speed. A directory tree with file names and an associative array with fast search functionality allow completing this task in logarithmic time. It is only required for the keys of the associated array to be the block numbers, and for the values to be the names of the respective files in the directory tree.

Creating these fast search structures is performed on capturing the first query to the partition. The question of how these structures should change while capturing new operations in the file system has several answers.

The first option is: the structures can remain unchanged. In this case, file accesses performed before the operating system has been loaded can also be traced. However, changing the size and location of these files makes the output data about file operations irrelevant or incomplete.

The second option is: the structures can be rebuilt from scratch with some determined periodicity. In this case, operations with new and changed files can be traced. However, some situations are possible when an operation with a recently created or enlarged file takes place, but the fast search structures have not yet been updated. In this case, such an operation can be left untraced.

The third option is: the structures can be rebuilt after writing to index descriptors, address blocks, and directories. This allows the fast search structures to always correctly reflect the current file system state. However, this method leads to a significant increase in time needed to process all the operations that change the file system state, which can negatively affect the performance of the guest OS.

The fourth option implies that when operations that change the file system occur, the existing fast search structures also change. With this method, fast processing of any disk accesses and monitoring of the current information about file operations can be achieved. The drawback of this solution is its implementation complexity.

Operations that change the file system include creation, deletion, expansion, truncation, renaming and moving of files. The mechanism to recognize these operations is based on detecting the structure to which data is written and then comparing old data with new. For example, if an operation results in adding a new entry to a directory, it indicates that a new file was created, or that an old file has been moved to this directory. For each type of operations, a specific processing mechanism exists, which performs required changes to the directory tree and the associated array.

5. The problem of unspecified order of queries to the disk

There is a problem which results in some file operations not being detected using the described methods. The problem is that most OSes do not perform writing to the disk immediately but delay it for a period of time. At the same time, the order in which the pending writing operations will eventually be performed can vary. In some cases, the information is registered in structures even though no information about their existence has been written to the disk yet.

The time diagram in fig. 3 demonstrates one of the possible sequences of operations when writing data to a new file.

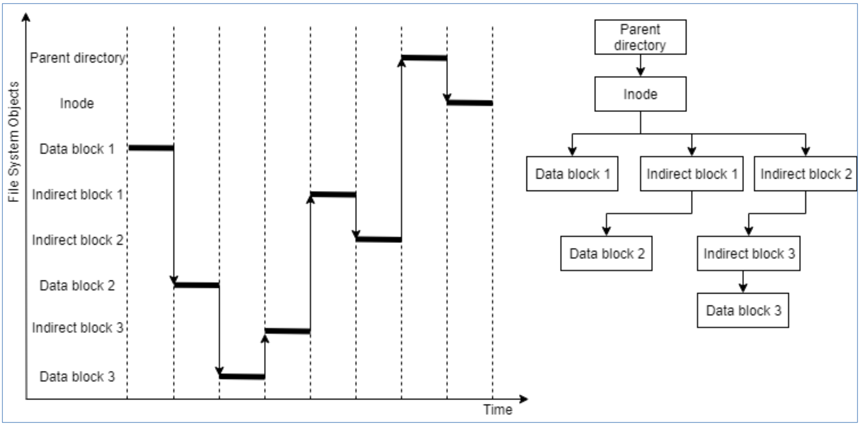

Fig. 3. Time diagram of writing data in case of new file creation

The solution to this problem is to store the information about unidentified operations for some time. When adding addresses to an inode and indirect blocks, previously unidentified operations are checked. If the address of the new block matches the destination of any of the earlier operations, this operation is processed.

A separate case occurs when creating files. In this situation, writing to the parent directory and the inode can be performed in any order. If, at the moment of writing to the directory, writing to the inode has already taken place, then this file operation is processed. Otherwise, the processing will be postponed until writing to the inode.

One more case is file moving. Sometimes the file information is added to the new directory first and is removed from the old one only after that. In this case, the file is moved in the directory tree while the deletion is ignored.

6. Testing

The implemented module creates its additional structures once during its operation, and then changes them in accordance with the new file system state. To verify that the module operates as expected, a series of tests were conducted using different guest OS images. The tests involved opening applications, navigating from folder to folder, as well as reading, creating, changing, deleting, renaming and moving files. Various situations were checked when the order of operations is ambiguous.

The following is an example of the module operation. In the guest OS (in this case, Debian), the command “dd if=test of=test1” is performed. This command copies the data in file “test”, 1 Kb in size, into file “test1”. The entries added to the log file are presented in fig. 4.

```
read   6926160      16384   /bin/dd
read   6926192      32768   /bin/dd
read   6926256      12288   /bin/dd
read   10263952     4096    /home/debian/test
...
create /home/debian/test1
write  2640360      4096    /home/debian
write  10490400     4096    /home/debian/test1
```

Fig. 4. Fragment of a log file generated by the module

Log entries contain information about the operation type, the sector number, the number of bytes affected and the file name. In this case, first, the executable file “dd” from the directory “bin” is read, and then the file “test” is read. Approximately 20 seconds after the input of the command, pending operations of writing to the disk are performed, and the log file is updated with new events. Then, the file “test1” is created, which is populated with data from “test”.

The testing was performed using Linux, FreeBSD, Windows 10 and KolibriOS operating systems [7]. The tests show that the module successfully registers file operations and processes the file system changes. At the same time, no lags due to the module monitoring were observed.

KolibriOS was chosen for testing as an example of an exotic operating system. This is a miniature OS, with its core and most of its programs written in assembly language. While testing, it was found that writing operations in this OS are not cached but performed immediately. Consequently, the problem of the order of operations being ambiguous is irrelevant for KolibriOS.

Thus, it was verified that the module functions correctly with each of the tested OSes.

7. Conclusion

In this paper, an approach to file operations monitoring has been described. This approach allows analyzing operations with operating system files and applications in a virtual machine. The implemented module works with the ext3 file system. It is intended for capturing virtual disk accesses in the guest system and writing the operation type and file name into a log. In contrast to internal file system monitoring tools, such as inotify, the created QEMU module can monitor file operations without interfering with the operation of the guest OS. In addition, the module does not depend on system calls, which allows it to work with any OS. While implementing the module, it has also been made possible to achieve a high speed of file operations processing. To do this, QEMU creates and maintains special structures: binary search trees and directory trees. The solutions described in the article can be applied to develop monitoring instruments in other file systems, including FAT32, NTFS, ext4.

References

[1]. Bellard, F. QEMU, a fast and portable dynamic translator. In Proceedings of the USENIX Annual Technical Conference, 2005, pp. 41–46.

[2]. Brian Carrier, File System Forensic Analysis. Addison-Wesley Professional, 2005.

[3]. Koen Vervloesem. Inotify: Watch your filesystem. Linux format, № LXF140, 2011.

[4]. FileSystemWatcher. https://msdn.microsoft.com/en-us/library/system.io.filesystemwatcher(v=vs.110).aspx

[5]. Jonathan Lemon. Kqueue - A Generic and Scalable Event Notification Facility, Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference, 2001, p.141-153

[6]. P. Dovgalyuk, N. Fursova, I. Vasiliev, V. Makarov. 2017. QEMU-based framework for non-intrusive virtual machine instrumentation and introspection. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017), pp. 944-948. https://dx.doi.org/10.1145/3106237.3122817

[7]. Artem Jerdev. Kolibri-A: a lightweight 32-bit OS for AMD platforms, Postgraduate Conference for Computing: Applications and Theory (PCCAT 2011), pp. 20-22.

Отслеживание операций с файловой системой ext3 в эмуляторе QMU

В.М. Степанов <vladislav.stepanov@ispras.ru>
П.М. Довгалюк <pavel.dovgaluk@ispras.ru>
Д. Н. Полетаев <poletaev@ispras.ru>
Новгородский государственный университет имени Ярослава Мудрого,
173000, Россия, г. Великий Новгород, ул. Лазаревская, дом 11

Аннотация. В работе рассматривается подход к отслеживанию файловых операций с помощью перехвата запросов к виртуальному диску в эмуляторе. Такой способ позволяет получать информацию о файловых операциях независимо от гостевой ОС, однако требует отдельной реализации для каждой файловой системы. Важной проблемой для реализации данного подхода является корректная обработка изменений в файловой системе. С операционными системами, которые имеют свойство кешировать операции записи, возникают осложнения, так как операции записи могут выполняться в произвольном порядке. Для примера подхода был создан модуль эмулятора QEMU, отслеживающий операции с файловой системой ext3. Преимущество данного инструмента перед другими состоит в отсутствии вмешательства в работу ОС, а также отсутствии зависимости от ОС. Благодаря этому возможно использование на таких экзотических ОС, с которыми не работают другие инструменты мониторинга файловых операций. Предполагается, что модуль QEMU для файловых систем, отличных от ext2/3, может быть реализован с использованием методов, подобных описанным в статье.

Ключевые слова: виртуальные машины; файловые системы; мониторинг; QEMU; интроспекция

DOI: 10.15514/ISPRAS-2018-30(5)-6

Для цитирования: Степанов В.М., Довгалюк П.М., Полетаев Д.Н.. Отслеживание операций с файловой системой ext3 в эмуляторе QMU. Труды ИСП РАН, том 30, вып. 5, 2018 г., стр. 101-108 (на английском языке). DOI: 10.15514/ISPRAS-2018-30(5)-6

Список литературы

[1]. Bellard, F. QEMU, a fast and portable dynamic translator. In Proceedings of the USENIX Annual Technical Conference, 2005, pp. 41–46.

[2]. Brian Carrier, File System Forensic Analysis. Addison-Wesley Professional, 2005.

[3]. Koen Vervloesem. Inotify: Watch your filesystem. Linux format, № LXF140, 2011.

[4]. FileSystemWatcher. https://msdn.microsoft.com/en-us/library/system.io.filesystemwatcher(v=vs.110).aspx

[5]. Jonathan Lemon. Kqueue - A Generic and Scalable Event Notification Facility, Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference, 2001, p.141-153

[6]. P. Dovgalyuk, N. Fursova, I. Vasiliev, V. Makarov. 2017. QEMU-based framework for non-intrusive virtual machine instrumentation and introspection. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017), pp. 944-948. https://dx.doi.org/10.1145/3106237.3122817

[7]. Artem Jerdev. Kolibri-A: a lightweight 32-bit OS for AMD platforms, Postgraduate Conference for Computing: Applications and Theory (PCCAT 2011), pp. 20-22.110