ISSN: 2321-7782 (Online)
Volume 2, Issue 3, March 2014
International Journal of Advance Research in Computer Science and Management Studies
Research Article / Paper / Case Study
Available online at: www.ijarcsms.com

Block-Wise Data Recovery for Virtual Hard Disk

Rohnak K. Motwani[1]
Dept. of Information Technology
MITCOE
Pune – India

Rishabh K. Singh[2]
Dept. of Information Technology
MITCOE
Pune – India

Abhijit Awachar[3]
Dept. of Information Technology
MITCOE
Pune – India

Saranga Bhutada[4]
Assistant Professor
Dept. of Information Technology
MITCOE
Pune – India

Abstract: ***Whenever we want to store some data for future use, we store it on the hard disk. Hard disk drive is a data storage device used for storing and retrieving digital information. Data is stored on the hard disk in 4096 bytes of block containing 8 512 byte sectors each.***

Just like physical hard drives, VHD files are at risk for corruption, accidental deletion, virus attacks and other causes of data loss. As such, the ability to recover data from a VHD image can be very valuable.

In case data stored on hard disk gets corrupt, the information cannot be retrieved and the whole file system gets corrupted. To get access to data, it needs to be retrieved from backup. With this, whole file system needs to be replaced to get access to original data. In case of large volume size, this technique proves to be inefficient.

So instead of requesting whole volume from the backup, replacing only the corrupted blocks or files would be efficient. This will lead to better management of memory and recovery time.

I. INTRODUCTION

A. Need

With the help of driver development kernel mode C programming, we are proposing a way to recover only the corrupted blocks of the .vhd file instead of replacing the whole file system. This will be an efficient technique as less memory space will be wasted.

B. Application

To provide all the users with an effective way of recovering only corrupted data block-wise which will prove to be an efficient technique since memory won't be wasted as much.

C. Basic concept

1) Filter Manger:

The filter manager is a kernel-mode driver that conforms to the legacy file system filter model and exposes functionality commonly required in file system filter drivers. By taking advantage of this functionality, third-party developers can write mini filter drivers, which are simpler to develop than legacy file system filter drivers, thus shortening the development process while producing higher-quality, more robust drivers.

2) File System Mini Drivers:

The Filter Manager was meant to create a simple mechanism for drivers to filter file system operations: file system mini filter drivers. File system mini filter driver are located between the I/O manager and the base file system, not between the file system and the storage driver(s) like legacy file system filter drivers. File system mini filter drivers are simpler than legacy drivers and hence less prone to bugs.

3) Input Output Request Packets (IRPs):

I/O request packets (IRPs) are kernel mode structures that are used by Windows Driver Model (WDM) and Windows NT device drivers to communicate with each other and with the operating system. They are data structures that describe I/O requests, and can be equally well thought of as "I/O request descriptors" or similar. Rather than passing a large number of small arguments (such as buffer address, buffer size, I/O function type, etc.) to a driver, all of these parameters are passed via a single pointer to this persistent data structure. The IRP with all of its parameters can be put on a queue if the I/O request cannot be performed immediately. I/O completion is reported back to the I/O manager by passing its address to a routine for that purpose, IoCompleteRequest. The IRP may be repurposed as a special kernel APC object if such is required to report completion of the I/O to the requesting thread. IRPs are typically created by the I/O Manager in response to I/O requests from user mode. However, IRPs are sometimes created by the plug-and-play manager, power manager, and other system components, and can also be created by drivers and then passed to other drivers.

4) The Callback Mechanisms:

The old mechanism of filtering file system operations (between the filesystem and storage driver(s)) required handling IRPs and the creation/handling of device objects. On the other hand, file system minifilter drivers that utilize the Filter Manager utilize a callback mechanism. This callback mechanism specifies what IRPs you are interested in filtering. The structure for doing this is extremely simple.

```
const FLT_OPERATION_REGISTRATION FilterCallbacks[] = {
    {IRP_MJ_CREATE,
    0,
    PreCreate,
    PostCreate,
    NULL},
    {IRP_MJ_WRITE,
    0,
    PreWrite,
    NULL,
    NULL},
    {IRP_MJ_CLOSE,
    0,
    PreClose,
    PostClose,
    NULL},
```



```
    {IRP_MJ_OPERATION_END}
};
```

This structure is passed to the Filter Manager (with some other registration information) in a call toFltRegisterFilter. These callback functions (PreCreate, PostClose, etc.) then need to be defined in the driver and the filter manager ensures that the appropriate functions are called when it receives the IRPs you specified you wanted callbacks for. The pre-callbacks are for IRPs going from the I/O manager to the base filesystem. The function signature for pre-callbacks is:

```
typedef FLT_PREOP_CALLBACK_STATUS
  (*PFLT_PRE_OPERATION_CALLBACK) (
    IN OUT PFLT_CALLBACK_DATA Data,
    IN PCFLT_RELATED_OBJECTS FltObjects,
    OUT PVOID *CompletionContext
    );
```

Post-callbacks are for IRPs going in the opposite direction, from the base filesystem to the I/O manager and their function signature is:

```
typedef FLT_POSTOP_CALLBACK_STATUS
  (*PFLT_POST_OPERATION_CALLBACK) (
    IN OUT PFLT_CALLBACK_DATA Data,
    IN PCFLT_RELATED_OBJECTS FltObjects,
    IN PVOID CompletionContext,
    IN FLT_POST_OPERATION_FLAGS Flags
    );
```

There are a couple parameter differences between the pre- and post-callbacks, but the general idea is the same. The Data parameter specifies information about an I/O operation. Basically it represents the IRP information but in a simple to use container. The FltObjects parameter provides pointers to objects related to the operation including the volume and file object. The CompletionContext parameter is extremely helpful for passing information between the callbacks. Any pre-callback can set CompletionContext and that pointer will be passed in to the post-callback.

The CompletionContext mechanism and context support are my two most used components of the Filter Manager. While CompletionContext allows you to attach information to a specific IRP operation, the Filter Manager's context support allows you to attach information to a volume, instance, file, stream, or stream handle (support for all of these are not yet completed, so check the latest information in MSDN to see what is currently supported). While being able to transfer information to a post-callback is useful, sometimes higher-level tracking is required. For instance, a filter driver wanting to track all operations across the lifetime of a particular file open would use stream handles (which operates at the file object level).

5) Advantages of filter manager model:

The filter manager model offers the following advantages over the existing legacy filter driver model:

- **Better control over filter load order.** Unlike a legacy filter driver, a minifilter driver can be loaded at any time and attached at the appropriate location as determined by its altitude.
- **Ability to unload while system is running.** Unlike a legacy filter driver, which cannot be unloaded while the system is running, a minifilter driver can be unloaded at any time, and it can prevent itself from being unloaded if necessary.

The filter manager synchronizes safe removal of all minifilter driver attachments, and it handles operations that complete after the minifilter driver is unloaded.

- **Ability to process only necessary operations.** The filter manager uses a callback model in which a minifilter driver can choose which types of I/O operations (IRP-based, fast I/O, or FSFilter) to filter. The minifilter driver receives only I/O requests for which it has registered callback routines. A minifilter driver can register a unique preoperation or postoperation callback routine, or both, and it can ignore certain types of operations, such as paging I/O and cached I/O.
- **More efficient use of kernel stack.** The filter manager is optimized to reduce the amount of kernel stack it uses, and the callback model greatly reduces the impact of minifilter drivers on the stack. The filter manager reduces the impact of recursive I/O by supporting filter-initiated I/O that is seen only by lower drivers in the stack.
- **Less redundant code.** The filter manager reduces the amount of code required for a minifilter driver in a number of ways, such as providing infrastructure for name generation and caching file names for use by more than one minifilter driver. The filter manager attaches to volumes and notifies minifilter drivers when a volume is available. The filter manager is optimized to support multiprocessor systems, which makes locking both more efficient and less prone to error.
- **Reduced complexity.** The filter manager simplifies filtering I/O requests by providing support routines for common functionality, such as naming, context management, communication between user mode and kernel mode, and masking differences between file systems. The filter manager handles certain tasks on behalf of minifilter drivers, such as pending IRPs and enumerating and attaching to file system stacks.
- **Easier addition of new operations.** Because minifilter drivers register only for the I/O operations they will handle, support for new operations can be added to the filter manager without breaking existing minifilter drivers.
- **Better support for multiple platforms.** A minifilter driver can run on any version of Windows that supports the filter manager. If a minifilter driver registers for an I/O operation that isn't available at runtime, the filter manager simply doesn't call the minifilter driver for that operation. A minifilter driver can determine programmatically whether functions are available, and filter manager structures are designed to be extensible.
- **Better support for user-mode applications.** The filter manager provides common functionality for user-mode services and control programs that work with minifilter drivers. The filter manager user-mode library, Filterlib.dll, enables communication between a user-mode service or control program and a minifilter driver. Filterlib.dll also provides interfaces for management tools.

II. Aims and Objectives

We will develop a software system to create a block-wise data recovery from the backup, instead of requesting for the whole VHD file to be replaced when the data gets corrupted. We will request from the backup to replace only the corrupted block. This technique will prove to be efficient and recovery time will be reduced to a great extent.

III. Literature Survey

We studied different reference books related to file system, kernel architecture, block memory management, Windows Server 2012.

We referred websites like Wikipedia, StackOverflow.com, msdn.microsoft.com and many other similar websites.

A number of errors can occur when data stored on hard disk gets corrupt, the information cannot be retrieved and the whole file system gets corrupted. In order to access the data, it needs to be retrieved from backup. With this, whole file system needs to be replaced to get access to original data.

Data Recovery:

Data recovery is the process of replacing data from damaged, failed, corrupted, or inaccessible secondary storage media when it cannot be accessed normally. Often the data is being replaced from storage media such as internal or external hard disk drives, solid-state drives, USB flash drive, storage tapes, CDs, DVDs, RAID, and other electronics. Recovery may be required due to physical damage to the storage device or logical damage to the file system that prevents it from being mounted by the host operating system.

The most common "Data Recovery" scenario involves an operating system (OS) failure (typically on a single-disk, single-partition, single-OS system), in which case the goal is simply to copy all wanted files to another disk. This can be easily accomplished using a Live CD, many of which provide a means to mount the system drive and backup disks or removable media, and to move the files from the system disk to the backup media with a file manager or optical disc authoring software. Such cases can often be mitigated by disk partitioning and consistently storing valuable data files (or copies of them) on a different partition from the replaceable OS system files.

Another scenario involves a disk-level failure, such as a compromised file system or disk partition, or a hard disk failure. In any of these cases, the data cannot be easily read. Depending on the situation, solutions involve repairing the file system, partition table or master boot record, or hard disk recovery techniques ranging from software-based recovery of corrupted data, hardware-software based recovery of damaged service areas (also known as the hard drive's "firmware"), to hardware replacement on a physically damaged disk. If hard disk recovery is necessary, the disk itself has typically failed permanently, and the focus is rather on a one-time recovery, salvaging whatever data can be read.

In a third scenario, files have been "deleted" from a storage medium. Typically, the contents of deleted files are not removed immediately from the drive; instead, references to them in the directory structure are removed, and the space they occupy is made available for later overwriting. In the meantime, the original file contents remain, often in -a number of disconnected fragments, and may be recoverable.

IV. Overall Description

On the current windows server operating systems such as Windows Server 2012 or newer version of hypervisors the virtual hard disks for virtual machines are stored in .vhd file format on the server. The .vhd files can be manipulated like any other files, and consist of headers and footers.

Whenever data is written to a virtual hard disk, it is physically written to the .vhd file. This data is normally written in blocks of size 4096 bytes, for the .vhd files stored on the NTFS file system.

Our design approach involves writing a file system mini filter driver, this driver is attached to the filter manager on the host. This driver catches the writes for the .vhd file.

Whenever the write request is detected for the .vhd file in the pre-callback function of the driver, the respective checksums and bitmap (explained later) structures are updated. The data offset and length information for the write operation and the respective write buffer are available in the FLT_CALLBACK_DATA parameter of the function.

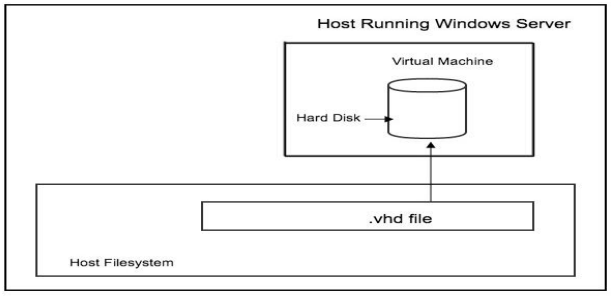

Fig.1 Overview

Checksum:

Writes to the .vhd file contain offsets which are perfectly aligned multiples of 4096 i.e the block size. Checksum needs to be maintained for each block and a write operation is usually spanned across multiple blocks. We have used the CRC32 (Cyclic Redundancy Check) algorithm to maintain the checksum for each block. The stored checksum is then used to detect corrupt blocks later.

Bitmap:

The following figure shows the need for the bitmap structure:

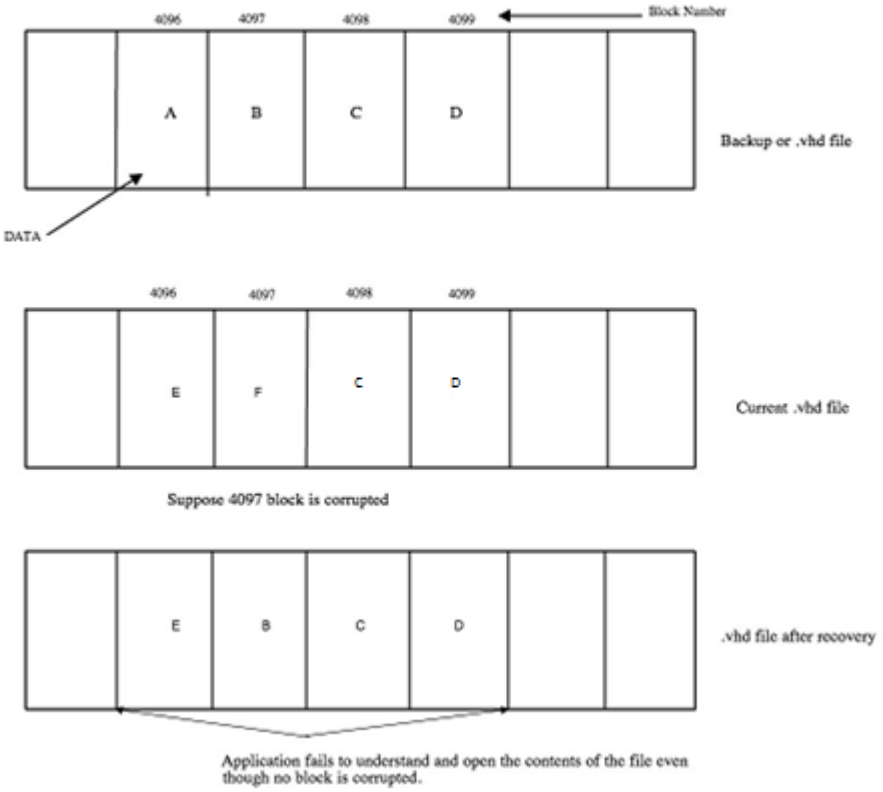

Fig. 2 Need for bitmap structure

Consider the hypothetical situation shown above,

At the time the backup was taken blocks numbered 4096, 4097, 4098 and 4099 contain data A, B, C and D respectively. Let us assume that these blocks form a file, which makes sense to an application. Sometime after the backup the file was modified by the application and now the blocks numbered 4096, 4097, 4098 and 4099 contain data E, F, C and D respectively. Now suppose the block 4097 becomes corrupt and is retrieved from backup. Now the blocks numbered 4096, 4097, 4098 and 4099 contain data E, B, C and D respectively.

The problem is that the application either understands data in the order A, B, C, D or E, F, C, D. The application fails to understand data in the order E, B, C, D.

Hence a bitmap structure is maintained which represents whether the block was modified after the last backup operation. A bit is used to represent a single block, 1 for *was modified* and 0 for *not modified*. Whenever a corrupted block is detected, the bitmap structure is checked to see if the block was modified after the backup operation. If block was modified, all the blocks modified after the backup operation are retrieved. Otherwise only the corrupted block is retrieved.

V. Deployment Overview

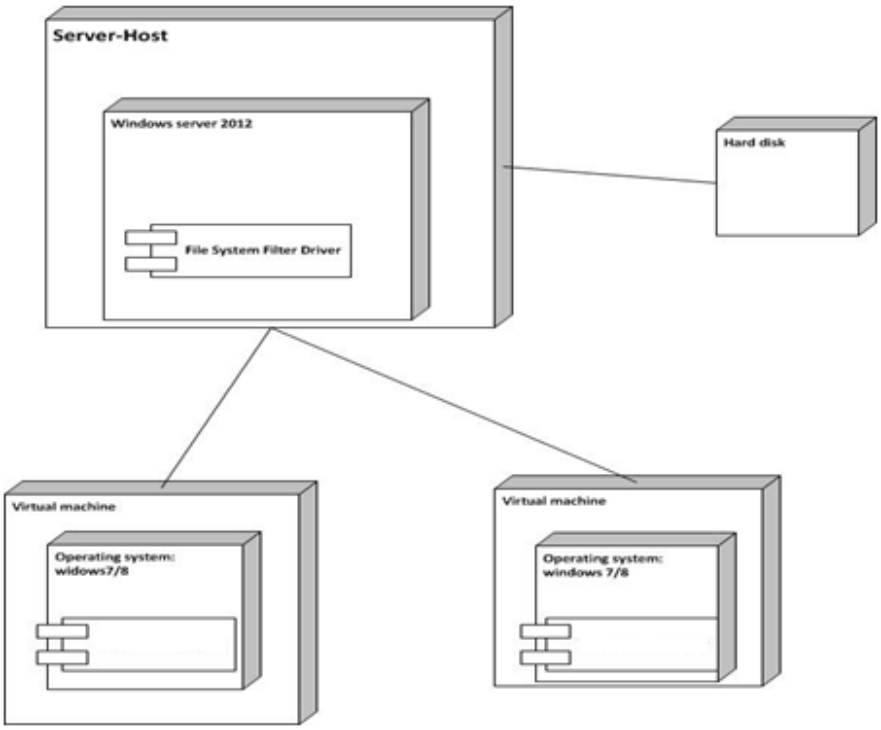

Fig. 3 Deployment View

VI. Sample Architectural Overview

The .vhd file stored on host machine is mounted on virtual machine (VM) that acts as virtual hard disk (VHD) for VM. Whenever data is to be stored on VHD, it is ultimately written on the .vhd file stored on the host. The file system mini filter driver on the host catches the writes for the .vhd file and updates the checksum and bitmap structures.

Fig. 4 Architecture Diagram

A utility is run on demand which reads the .vhd file block-wise and computes the checksum for each block. These checksums are then compared with the checksums calculated by the driver. Whenever the checksums do not match, the block is marked as eligible candidate for recovery and later recovered.

VII. Conclusion

In this paper we have given the overall idea of how we could manage the recovering of corrupted data block wise so that proper memory management takes place. We have given an overall idea about how we can go about achieving this effective way of memory management and have reduced the recovery time.

Acknowledgement

We take this opportunity to thank our project guide Prof. S.N.Bhutada and Head of the Department Dr.Anil S. Hiwale for their valuable guidance and for providing all the necessary facilities, which were indispensable in the completion of this project report. We are also thankful to all the staff members of the Department of Information Technology of MIT College of Engineering, Pune for their valuable time, support, comments, suggestions and persuasion. We would also like to thank the institute for providing the required facilities, Internet access and important books.

References

1. Programming the Windows Driver Model 2nd Edition-Walter Oney
2. Windows 2000 device driver book-Art Baker
3. http://en.wikipedia.org/wiki/Windows_Server_2012
4. http://blogs.technet.com/b/heyscriptingguy/archive/2008/08/13/how-can-i-find-files-metadata.aspx
5. http://msdn.microsoft.com/en-us/library/windows/desktop/aa364944%28v=vs.85%29.aspx
6. http://msdn.microsoft.com/en-us/library/windows/hardware/ff545762%28v=vs.85%29.aspx
7. http://msdn.microsoft.com/en-us/library/windows/hardware/ff538937(v=vs.85).aspx
8. http://blogs.msdn.com/b/erick/archive/2006/03/27/562257.aspx
9. http://social.msdn.microsoft.com/Forums/windowsdesktop/en-US/26a2faf6-b0b0-470f-9ff5-92b7d48df181/dynamically-addremove-partitions-in-upper-volume-filter-driver

AUTHOR(S) PROFILE

Rohnak K. Motwani, is presently pursuing BE-Information technology (4 years) from MIT College Of Engineering, Pune, India. His area of interest includes operating systems, artificial intelligence, computer networking, database management systems etc.

Rishabh K. Singh, is presently pursuing BE-Information technology (4 years) from MIT College Of Engineering, Pune, India. His area of interest includes operating systems,multimedia system, database handling, software architecture etc.

Abhijit Awachar, is presently pursuing BE-Information technology (4 years) from MIT College Of Engineering, Pune, India. His area of interest includes computer networking and security,operating systems, computer graphics etc.

Mrs. Saranga Bhutada, is working as an assistant professor at MIT College Of Engineering.She has completed her Masters in Information technology. Her area of interest for research includes data structures and operating system.

