CSE 524 Project Report

File system virtualization inside Windows system using mini-filter drivers.

Nikhil Pujari

SUNY SB ID: 106667284

Email: npujari@cs.sunysb.edu

Abstract:

The file virtualization system has been implemented as a mini-filter driver, which redirects the File I/O requests to a VM private workspace. The mini-filter is envisaged to be a part of a larger OS level virtualization system, which organizes virtual machines a process groups isolated from one another. The mini-filter seeks to replace a previous system call interception based copy-on-write mechanism to mirror the host's file system. The writes are redirected to VM's private workspace and hence the host file system is isolated from changes made by processes running inside the VM. The guest files are created and maintained as sparse files and a reparse points are associated with them which record the regions of the files which have been written to by the processes inside VMs. The file create/opens are intercepted and file handles from corresponding files from VM's private workspace i.e. guest files are returned to the processes. The reads belonging to the regions of a file which were unmodified are redirected to the host/original file while the others are passed on to the guest file. Contexts are used to keep track of dirty/modified regions and to store original/host file handles.

CSE 523 work:

In my CSE523 project work, I had implemented a file and registry operations logging mini-filter. The mini-filter filtered the following IRPs: IRP_MJ_READ/file reads, IRP_MJ_CREATE/file create and open, IRP_MJ_WRITE/file write, IRP_MJ_CLOSE/file close. The registry read/write operations were filtered and logged.

The **DriverEntry** routine is called when the mini-filter is loaded which registers the callbacks for the respective IRPs and registry operations. File I/O filtering starts when the filter is attached to a drive. For logging operations, we only need to register pre-operation callbacks for the respective IRPs. **CmRegisterCallback** function is used to register a callback for registry operations. The parameters such as Filename, Read/Write length, offset are retrieved from the IRP and recorded in the log file.

Architecture and Implementation:

Mini-filter overview:

A driver that is inserted between the Windows NT I/O system and the base file system driver is referred to as a file system filter driver. NT driver stack works with I/O Request Packets (IRPs) which are requests to do specific file system operations such as open, read, write, close etc. The NT I/O Manager sits at the top of the file system driver stack. It accepts system calls from processes, or function calls from other kernel components, translates handles into file objects, generates IRPs and hands them over to the topmost member of the driver stack. After the IRP completes the I/O manager completes the system or function call, or does other actions in case of asynchronous requests.

Windows XP SP2 introduced a new file system filter driver architecture known as the Filter Manager. In this file system filter drivers are written as mini-filter drivers which are managed by a Microsoft supplied Filter Manager. The Filter Manager itself is a legacy file system filter driver. IRPs are silently handled by the Filter Manager and callbacks registered by mini-filters are invoked by it. The CallBackData structure passed to the pre-operation and post-operation callbacks represents the IRP. It exports a relatively less complicated interface for development of mini-filters. Also, a lot of support routines are provided for name lookups, caching, cross-filter and user mode communication, buffer management, I/O cancels etc. Context management is easier and efficient compared to legacy filters.

VM private workspace:

VMs private workspaces are directories in the root folder named by the VMs ID. For example, VM1, VM2 would have their private workspaces as C:\VM1, C:\VM2 and so on.

Sparse files and Reparse points:

I employ the following facilities provided by the Windows NTFS file system for the implementation of this virtualization system: Sparse Files and Reparse Points. A sparse file is a file that does not have physical disk space allocated for the entire file. Parts of the file that are not allocated are logically filled with zeroes. A file may be marked as sparse and extended without reserving disk space for the extension. A write to an unallocated region causes disk space to be allocated.
A file or directory can contain a Reparse point, which is a collection of user-defined data. The format of this data is understood by an application or filter driver which sets it for a particular file or directory and operates on it. When a reparse point is set, a unique reparse tag is associated with it, which identifies it.

The guest files created in VM's private workspace are converted to sparse files as soon as they are created. This is done by calling the function, **FltFsControlFile** on the guest file handle with the control code **FSCTL_SET_SPARSE**. Then the file's size is made equal to the original host file by using the function **FltSetInformationFile**, with **FileInformationClass** type equal to **FileEndOfFileInformation** and the associated **FILE_END_OF_FILE_INFORMATION** structure containing the original file's size. Only the writes happening subsequently on the guest file would occupy disk space.
The reparse point data needed for the operation of our filter contains a list of dirty regions in the form of offsets and lengths. The reparse point is set using the function **FltTagFile**, along with a unique tag and GUID, which identify the reparse points associated with our mini-filter, when a file is created first in the VM's private workspace. On subsequent opens, the reparse data buffer is read using the function **FltFsControlFile**, which is the file IOCTL function, with the control code **FSCTL_GET_REPARSE_POINT**.

Context Management:

One of the features provided by the Filter Manager is the ability to track and manage contexts on behalf of the mini-filters. Contexts are reference counted data structures used by mini-filters to track the state of various objects. A context is a piece of dynamically allocated memory allocated by calling the function **FltAllocateContext**. Contexts can be attached to volumes, mini-filter instances, files, streams and stream handles. At registration time in the DriverEntry callback, the types of the contexts, their size and the routine used to clean up the context are defined. Each time is a context is referenced using the respective **FltGetXXXXContext**, its reference count is increased. Each context reference must be matched by corresponding **FltReleaseContext** function to decrease the reference count whenever the context has been used.

Two types of contexts are used in this mini-filter, stream and stream handle. Stream contexts represent contexts associated with all the open file objects across a single data stream. Stream handle contexts are associated with single open, i.e. a single file object.

The stream handle context is used to store the handle to the host/original file and is associated with the file handle of the guest file. Whenever an operation is needed to be redirected to the host/original file its handle is retrieved from the associated stream handle context and used. Stream context is used to store the pointers to the “dirty” and “faulted” lists (description of these lists is in the following sections), since these need to be referred by actions on all the open file handles for a particular file. The dirty and faulted lists are protected by **ERESOURCE** locks.

Handling of create/open requests:

When a process requests a create/open of a file, a sparse file with the same name is created/opened in the VM’s private workspace. It is done in the following way. In the pre-operation call back for **IRP_MJ_CREATE** the filename is retrieved. The **FLT_FILE_NAME_INFORMATION** structure with the retrieved filename is assigned to the completion context of the pre-operation call back. A completion context is a parameter passed to both the pre-operation and post-operation callback. It is a mechanism for passing arbitrary context data from pre-operation to post-operation callback. The reason for retrieving name in the pre-operation callback is that it is always invoked at a **Passive IRQL**, and the post-operation callback can be called at an **IRQL DPC_LEVEL**, while the function **FltGetFileNameInformation** can only be called at an IRQL less than that.

Then the target filename of the create operation in the field **Iopb->TargetFileObject->FileName** is changed to the guest name (e.g. c:\test.txt to c:\vm1\test.txt) and the CallbackData is marked as dirty by calling **FLT_SET_CALLBACK_DATA_DIRTY**() to make that change effective. In this way the create operation is redirected to the guest file and its handle is returned to the requesting process.

In the post-operation callback for the **IRP_MJ_CREATE**, the original file is opened with the original flags/parameters using the completion context passed from the pre-operation callback. Then a stream handle context is allocated and this handle to the original/host file is stored in it. This context is then attached to the file handle of the file being created/opened. This is the filehandle to which the read requests would be redirected as needed. The reason that allocation and setting of the stream handle context is done in the post-operation callback of the **IRP_MJ_CREATE** is that the function **FltSetStreamHandleContext** cannot be called on an unopened file object.

Handling read/write requests:

A copy-on-write approach to file virtualization would entail copying the host file to the corresponding guest file upon first write and redirect all subsequent read-write requests to the guest file. I avoid copy-on-write in the following way. I maintain a map of the "dirty" regions of the file, where dirty implies the writes done by the processes in the VMs. The map is maintained in the form of a list whose elements contain the offset of and length of the dirtied part of the file.

In the pre-operation call back of **IRP_MJ_WRITE**, the offset and length of the write are extracted. Writes are executed on the file handle of the guest file which was returned to the process at file create/open, which is essentially a pass-through operation. If the write is successful, the corresponding part of the file is marked as dirty by appending the corresponding offset and length to the dirty list. If the region is contiguous to another dirty region the length of that region is increased or the offset changed accordingly.

In the **IRP_MJ_READ** pre-operation callback, the offset and length of the operation are retrieved. Then the dirty list is scanned and checked whether the region to be read is dirty. If the read spans multiple regions, it is split into multiple reads and they are satisfied from original file handle and guest file handle.

The mechanism for the actual redirection of reads is implemented as follows. If the read is from dirty region, it is allowed to pass through, and in this case the filter acts as a simple pass through filter. This ensures that processes in each VM read from their own files, and not from then host/original file, and see the modifications done by them and not from processes from other VMs. If the read belongs to a clean region, the buffer is retrieved from the IRP/callback data passed to the **IRP_MJ_READ** pre-operation callback using the function **FltDecodeParameters**. **FltDecodeParameters** returns pointers to the memory descriptor list (MDL) address, buffer pointer, buffer length, and desired access parameters for an I/O operation. This saves minifilter drivers from having a switch statement to find the position of these parameters in helper routines that access the MDL address, buffer pointer, buffer length, and desired access for multiple operation types. In case of Direct I/O the data to be read is copied to user space buffers directly and IRP encodes it in the form of a MDL. To retrieve the address of the buffer described by the MDL, **MmGetSystemAddressForMdlSafe** needs to be used which locks the user buffer in memory, maps those pages in system memory and returns them. In case of buffered I/O, the **ReadBuffer** parameter from the I/O parameter block, which a part of the CallbackData passed to the pre-operation callback, needs to be used. **FltDecodeParameters** handles these switch cases internally and returns the appropriate buffer address. After retrieving the buffer, it still needs to be checked that it can be written to. Since it is a read operation, the data read is to be written in the buffer hence the writablility check. This check is performed using the **ProbeForWrite** function, which accepts the pointer to the buffer and its length. This function needs to be enclosed in a try/catch block, as an exception would be thrown if it is not writable, especially in case the user buffer is used directly.

Once the buffer and the length are retrieved, the read is satisfied by calling **ZwReadFile** from inside the pre-operation callback for **IRP_MJ_READ** on the original/host file handle, which is retrieved from the stream handle context. If the read is successful, then the IRP is completed here, by returning the status **FLT_PREOP_COMPLETE**. The lower level drivers never see the original IRP. In this way the redirection is achieved by intercepting the original IRP, generating a new IRP on the guest file with the same parameters, and completing the original IRP.

<u>Handling of Asynchronous reads/writes:</u>

If the original read/writes IRPs are asynchronous then the corresponding IRPs generated for the guest files from the mini-filter are also completed asynchronously. In that case the IRPs are generated using **FltReadFile/FltWriteFile** instead of **ZwReadFile/ZwWriteFile**. These routines take as input pointers to **PFLT_COMPLETED_ASYNC_IO_CALLBACK-**typed callback routines to call when the corresponding read/write operation is complete. The **IRP_MJ_READ/IRP_MJ_WRITE** pre-operation callbacks return a status of **FLT_PREOP_PENDING**. This is to notify the Filter Manager that the mini-filter will complete the IRP by calling **FltCompletePendedPreOperation**(), which in this case is called from the **PFLT_COMPLETED_ASYNC_IO_CALLBACK**-typed callback routines passed to the **FltReadFile/FltWriteFile** of the original IRP is suspended by the Filter Manager and lower level drivers never see this IRP.

If it is paging I/O then **FltReadFile** cannot be used since it can only be used at an **IRQL PASSIVE_LEVEL**. In that case the function used is **FltPerformAsynchronousIo**. A callback structure is allocated and filled with appropriate parameters which consist of the guest file object, obtained from the guest file handle using the function **ObReferenceObjectByHandle**. This function accepts a similar callback routine pointer as **FltReadFile** to complete the asynchronous operation and the IRP.

<u>Handling of memory mapped files:</u>

Redirection of reads/writes on memory mapped files requires a different mechanism. When a user (or the system cache manager) maps a file and touches a page for the first time, it will generate a page fault which will be translated by the NT IO system into a read request (**IRP_MJ_READ**) which would intercepted by our mini-filter. These read requests are paging I/O and are specially marked as such. They can be detected by checking for the flags **IRP_PAGING_IO** or **IRP_SYNCHRONOUS_PAGING_IO** in the **Iopb->IrpFlags** member of the CallbackData structure passed to the **IRP_MJ_READ** pre-operation callback. Once this data has been read, the virtual memory system will map the appropriate page, and future accesses (either reads or writes) will not generate any action that can be intercepted by our mini-filter. Other users who map and touch the same portion of the same file will be provided with a mapping to the same page, but the virtual memory system will not generate new IRPs since it already has the data. The virtual memory system can unmap and throw away any clean (unwritten) pages without the interception by the mini-filter and it can generate writes for dirtied pages asynchronously at its own pace.

Until the dirtied pages have been written to the disk we cannot mark them as dirty in our dirty list. This may potentially lead to incoherency between memory mapped reads and normal reads. Treating the data which has been page faulted in as clean will make the subsequent **IRP_MJ_READ**s to be passed to the original/host file and hence data changed by memory mapped writes will not be read. Treating them as dirty will pass the request to the guest file. In that case if the page was not written to and hence discarded by the virtual memory manager, it will generate another page faulted read.

The normal reads and page faulted reads need to be distinguished and handled separately. Another list similar to the dirty is created for this purpose, called the “faulted” list. When a page is first read by a page faulted read, which is detected by checking for **IRP_PAGING_IO** or

IRP_SYNCHRONOUS_PAGING_IO flags, it is added to the “faulted’ list. Subsequent normal **IRP_MJ_READ**s will be allowed to pass through to the guest file. If the page was written to/dirtied, it would either be written to the disk by the virtual memory manager before discarding it or it would still be in the cache. Either way the data read would be correct and would reflect the latest written data.

If this read generates another page faulted read, it means that the page was not written to and was discarded. Then this read can be redirected to the original file. The process is summarized as below.

1) When a page faulted read comes, check if the region exists in the dirty list.

2) If yes, pass the read to the guest file i.e. pass through.

3) If no, redirect the read to the original/host file. Add the region to the faulted list, if it already isn’t there.

4) If a normal read comes, check if the region belongs to the dirty list. If yes then pass the read to the guest file i.e. pass through.

5) If no, check if the region belongs to the faulted list.

6) If yes, then pass the read to the guest file i.e. pass through. If no, then redirect the read to the original/host file.

In step 6, if the page was clean and was discarded, this will generate a page fault and hence paging I/O i.e. a page faulted read, which would be intercepted and handled as above. These steps ensure the coherency between normal reads and memory mapped reads to a memory mapped region of the file. The redirection of read in step 3 ensures that the page cache associated with the guest file is filled with data from host file. When this region is written to and hence dirtied, the paging I/O writes generated by the virtual memory manager go to the guest file on the disk.

Handling of file close:

In the **IRP_MJ_CLOSE** pre-operation callback, firstly the original/host file handle is retrieved from the associated stream handle context. **FltClose** is called on it to close the file. Then the dirty list associated with the guest file is written to its reparse point using the function **FltTagFile** and the unique GUID and Tag values.

An alternative approach also could be followed, which would allow us to get rid of dirty lists and redirections. If the size of the dirty regions almost equals the size of the original/host file then the rest of the data can be copied, in the pre-operation callback for **IRP_MJ_READ**, and the dirty list can be removed and replaced with a flag value indicating that the file has been completely replaced/rewritten in the guest. All subsequent operations on that file can be directly passed on the guest file.

Handling of directory operations:

The combination of **IRP_MJ_DIRECTORY_CONTROL** as a major code and **IRP_MN_QUERY_DIRECTORY** is used to issue IRPs for querying directories. In the pre-operation callback of **IRP_MJ_DIRECTORY_CONTROL** we check if the minor code for the operation is **IRP_MN_QUERY_DIRECTORY** from the **Iopb->MinorFunction** member of the CallBackData structure. To handle this, the function **ZwQueryDirectoryFile** is executed on both the host directory and the corresponding guest directory. The **FileInformationClass** type of the original IRP is used in the function and buffers are allocated for the corresponding directory information to be returned.

Typically the **FileInformationClass** is **FileDirectoryInformation** which returns a list of **FILE_DIRECTORY_INFORMATION** structures for each file in the directory. Each structure contains the file information such a create time, accessed time, end of file, file attributes, file name etc. When these lists for both the host and guest directories are obtained they are merged to form a union of both the lists and this list is returned. A list of deleted or renamed files is maintained per VM. This can be done by intercepting **IRP_MJ_SET_INFORMATION** which has the **FileInformationClass** parameter equal to **FileDispositionInformation** and the corresponding **FILE_DISPOSITION_INFORMATION** structure contains a boolean value of TRUE. After both the lists are merged, the entries for deleted files can be removed from the union list.

Fast I/O handling:

Typically I/O requests are generated by the I/O manager in the form of IRPs. However, if the data is cached by the cache manager, the data can be directly obtained from system cache instead of through the file system. This method of I/O is called Fast I/O. However in this mini-filter the reliance of the redirection operations is based on filtering and completing IRPs in the mini-filter callbacks, hence Fast I/O requests are denied, when they are intercepted. When Fast I/O is denied, the I/O manager simply regenerates the requests using the normal IRP method. This is done by detecting whether an I/O operation is of a Fast I/O type using the macro **FLT_IS_FASTIO_OPERATION** on the CallBackdata structure. If yes then the pre-operation callbacks simply return the status of **FLT_PREOP_DISALLOW_FASTIO**.

Conclusion:

The advantages of the approach followed are reduction is used disk space and system cache memory space. Only additional space required, if any, is by the writes made by processes from the VMs. Since the guest files are sparse files, they do not actually occupy the amount of space indicated by their size. This approach also avoids any copy-on-write or copy-on-open-for-write. The downside of the approach followed is the increased complexity for handling memory mapped files and time consumed in dirty and faulted list traversals while processing reads. The list traversals can be made more efficient at the cost of increased complexity by maintaining them as balanced range trees or B+ trees. Also for redirections additional IRPs are generated from the mini-filter which adds to the overhead.

A copy-on-close mechanism could be added to the mini-filter depending upon the percentage of host/original file overwritten. Copy-on-close would eliminate all costs for maintaining the dirty and faulted lists and redirection of reads.

References:

1) Microsoft Filter Driver Developer Guide
2) MSDN Library and DDK(Device Driver Kit).
3) Single Instance Storage in Windows 2000 by William J. Bolosky, Scott Corbin, David Goebel, and John R. Douceur . Microsoft Research, *Balder Technology Group, Inc.
4) Windows NT File System Internals by Rajeev Nagar
5) OSRONLINE.com NTFSD (NT File System Development) mailing list.