Behind the Scenes of Xcitium's Kernel API Virtualization

ZeroDwell Containment

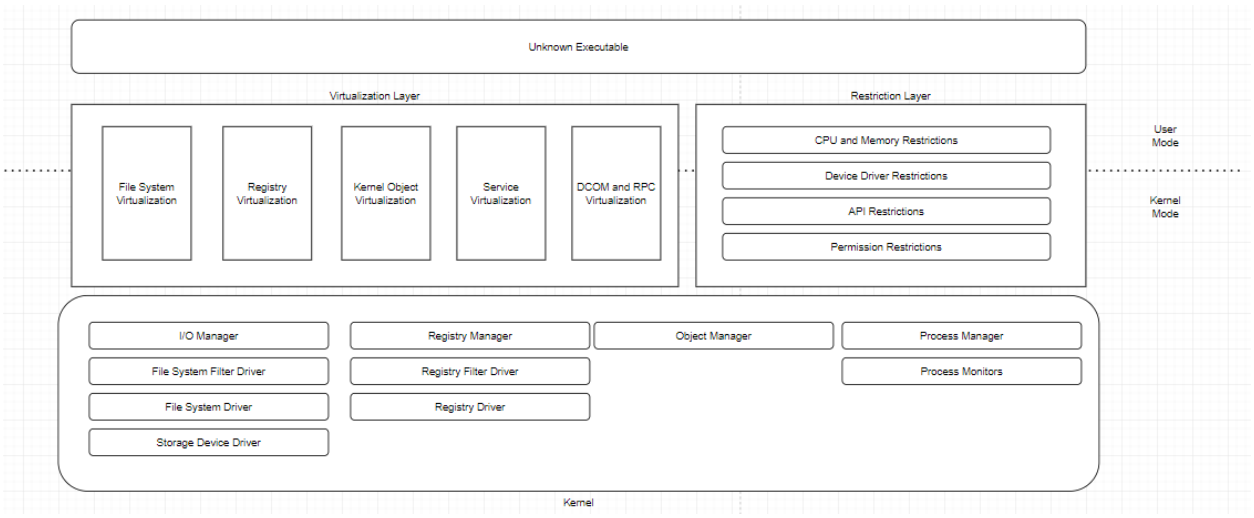
Behind the Scenes of Xcitium's Patented "Kernel API Virtualization"

When it comes to securing your enterprise endpoints, it's important to have a foundational understanding that there are three types of files: the good, the bad and the unknown. Approaches such as Antivirus (both vendor-branded "next gen" and legacy detection-based), Blacklisting, and Whitelisting handle the known good and the bad files – but what about the unknown files?

Regardless of the "next gen" nature and effectiveness of any new pre-execution, detection-based solution, there will always be a certain number of unknown files, executables, and code that by default are allowed to run on the host if not deemed malicious. The problem is that detection-based solutions will never detect 100% of what is malicious, or 100% known to be good. Unknown files may be perfectly harmless and required for system functionality, or they may be dangerous zero-day threats or APTs that cause mega breaches and damage. Your cyber security solution must be able to detect the difference to both prevent breaches and enable productivity.

Xcitium's Solution: Kernel API Virtualization

A key component of Xcitium technology is Kernel API Virtualization, or ZeroDwell Containment. This patented virtualization feature defeats zero-day attacks with no impact to the end user experience, and does so better than any other security technology on the market today. Xcitium's solution uses a combination of kernel API virtualization, whitelisting, machine learning, behavior analysis, and advanced static and dynamic threat cloud analysis (Xcitium Verdict Cloud) to accurately and quickly deliver a 100% trusted verdict for unknown files and processes. Pre-execution, our technology authenticates every executable and process that requests runtime privileges, and if not 100% known-good or known-bad, the file or object is deemed unknown, and ushered inside a secure, virtual environment that does not allow WRITE access to system resources or user data. This provides total protection against zero-day threats, proactively prevents damage, and has no impact on end-user experience or workflows. Whether the unknown files are malicious or safe, our technology is architected so they run and perform in auto-containment just as well as they would on the actual host system. However, they cannot damage or infect the systems because they cannot access the underlying system. This allows safe applications the freedom to run as needed while denying malicious applications the system access they need to deliver their payloads.

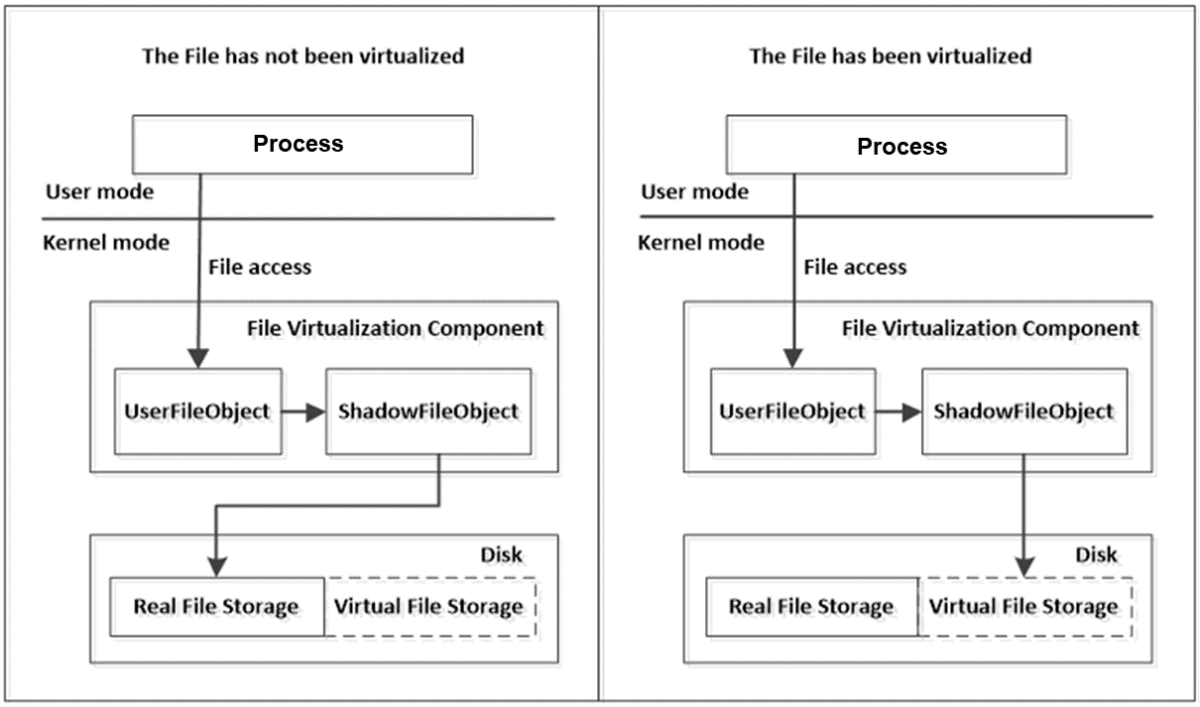


This protection is achieved by introducing a virtalization layer between processes running an unknown executable with Kernel functions. We have introduced 5 main virtualization components that filter any relevant Kernel calls or callbacks:

- File System
- Registry
- Kernel Object
- Service
- DCOM/RPC

These are the main virtualization components that run both user and kernel mode, handle necessary interrupts, and implement all necessary filter drivers to fulfill requests in virtualization (which is not a sandbox).

File System Virtualization is a good example for understanding this. File System virtualization is an abstraction layer between a File System and the client programs that access those files. It provides a logical view of the files. By using redirection techniques, a client program's access to a physical file is redirected to a virtual file, which prevents malicious programs modifying system files, and this also isolates operations to the file. The client program doesn't need to be concerned with the details of file virtualization; it is completely transparent.

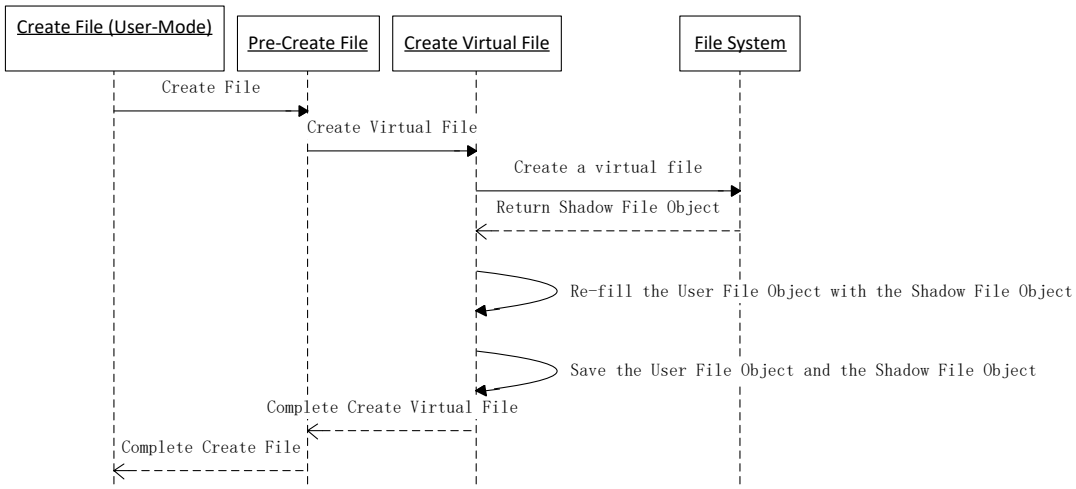


This type of virtualization can only be done at kernel level via file system filter drivers, where we can capture all relevant events, modify them, or redirect them as necessary. The routines we have captured and virtualized are: file creation, file read, file write and file change. Let's take file creation event as an example and summarize what routines should be captured and their interactions.

PreCreateFile routine executes when any process wants to access a Kernel for file create operations, and according to input parameters, there are two cases: create a new file or open an existing file.

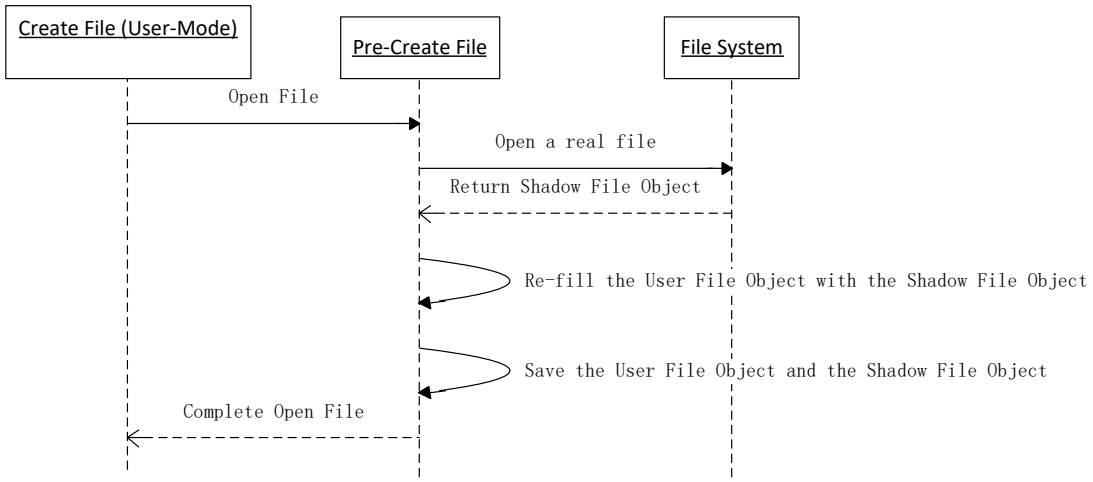
Create File

The following figure shows the interaction of file creation virtualization:

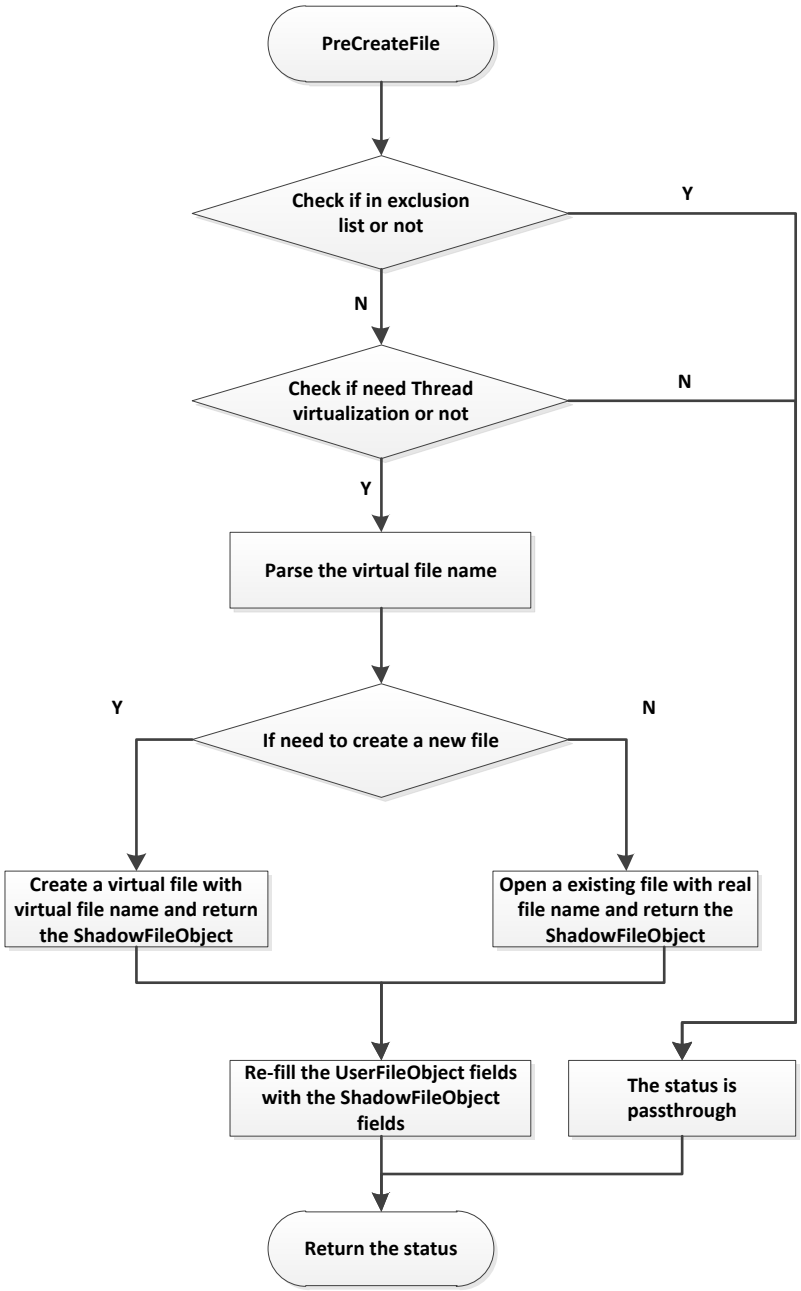


Open Existing File

The following figure shows the interaction of file opening virtualization:



The following figure shows the main flow of the create file operation:



As shown in the figure, the **PreCreateFile** routine summarizes the following list:

1. Check if the file path is in the **exclusion list** or not. If it is in the list, bypass. Otherwise go to step 2.
2. Check if the thread which creates or opens the file needs to be virtualized. If it needn't be virtualized, bypass. Otherwise go to step 3.
3. Retrieve the full path information for the accessed file, and parse the virtual file name.
4. Verify that the operation is to create a new file or open an existing file. If it is to create a new file, go to step 5. Otherwise go to step 6.

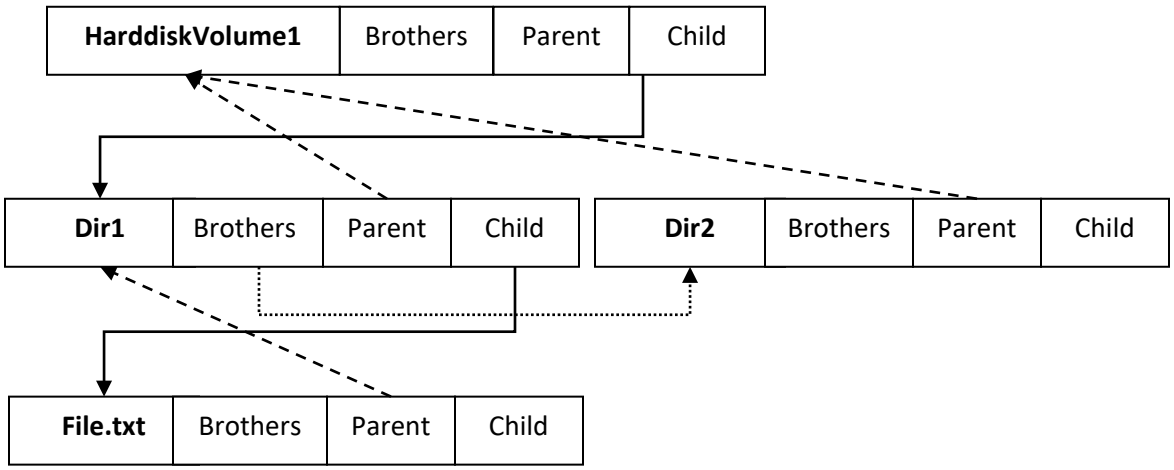
5. Create a virtual file in the **virtual file storage**, and return the ShadowFileObject that points to the virtual file. Re-fill the UserFileObject fields with the ShadowFileObject fields.
6. Open an existing file, and return the ShadowFileObject which points to the real file. Re-fill the UserFileObject fields with the ShadowFileObject fields.

Virtual File Maintenance

The file virtualization component uses a **virtual file tree** in memory to maintain the virtual files. We use this tree to cache virtual files and record states of virtual files, like deletion, renaming, etc. Following is the definition of the **virtual file tree node**:

```
typedef struct _SB_VIRTUAL_FILE_TREE_NODE
{
     UNICODE_STRING                          Name;
     ULONG                                   Flags;
          SB_VIRTUAL_FILE_TREE_NODE *        Parent;
          SB_VIRTUAL_FILE_TREE_NODE *        Child;
          PRTL_SPLAY_LINKS                   Brothers;
          ……
} SB_VIRTUAL_FILE_TREE_NODE, *PSB_VIRTUAL_FILE_TREE_NODE;
```

SB_VIRTUAL_FILE_TREE_NODE structure describes the directory (file) structure of virtual files. The following figure shows the virtual files on the **virtual file tree** ("\HarddiskVolume1\Dir1\File.txt" and "\HarddiskVolume1\Dir2").



The **virtual file tree** node is created when a process creates or opens a file in **PreCreateFile** routine. **If it creates a new file, the** file virtualization component **redirects the operation into the virtual file storage** (create a new virtual file) and then marks the Flags field as FV_FLAGS_VIRTUALIZED. If it opens an existing file, the file virtualization component marks the Flags field **as FV_FLAGS_NOT_VIRTUALIZED.**

When a process intends to delete (or rename) a file on disk, if the file has been virtualized before, the virtual file is deleted (or renamed) instead of the real one. After the operation is complete, the **virtual file tree** will be updated. On the contrary, if the file has not been virtualized before, it means the process

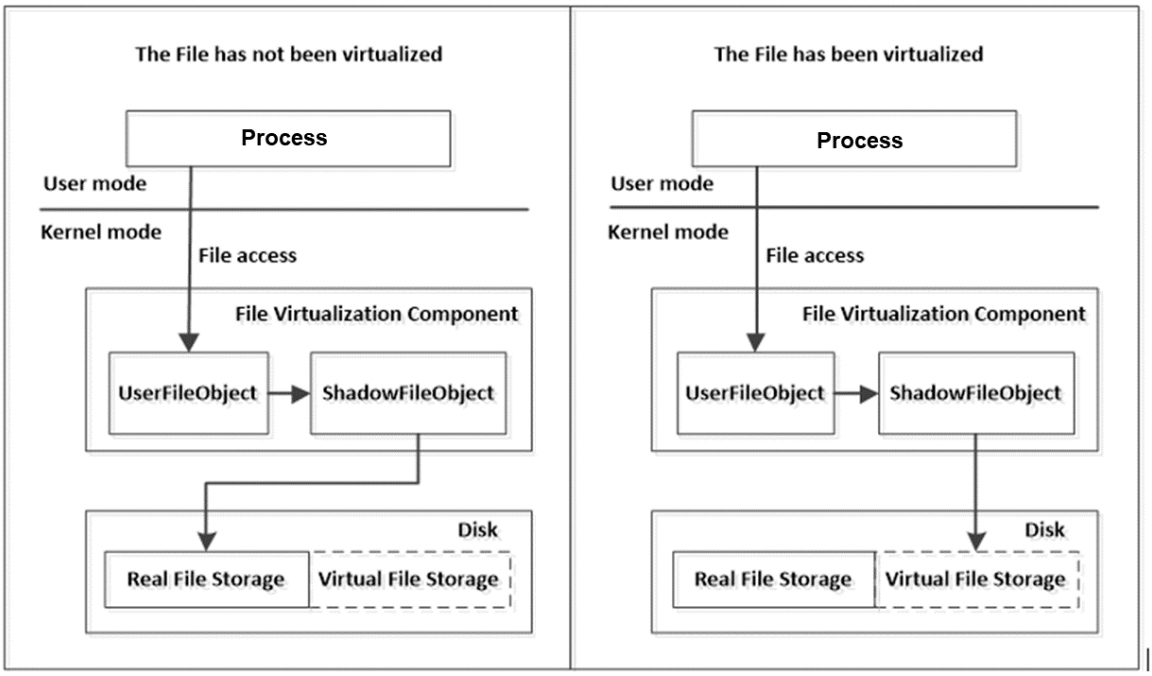
wants to delete (or rename) a real file on disk, so no files are actually deleted (or renamed). We only mark the Flags field **as FV_FLAGS_DELETED on the virtual file tree node.**

When a process wants to enumerate a directory to get files in the directory, we hide the files marked for deletion.

After all the processes that access the same file name exit, the **virtual file tree** node instance is freed. Note that the nodes that have been marked for deletion could not be freed in order to mark the deleted files in the future.

File System Virtualization:

File System virtualization is an abstraction layer between the File System and the client programs that access those files. It provides a logical view of the files. By using redirection techniques, some client programs access to the physical file is redirected to a virtual file, which prevents malicious programs modifying system files and isolates the operations to the files. These client programs do not need to be concerned with the details of file virtualization; it is completely transparent.



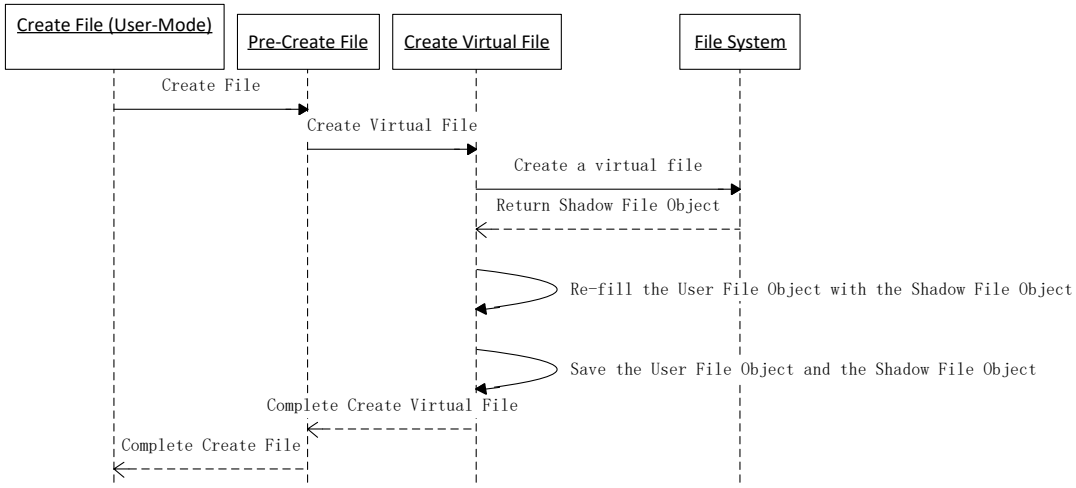
Again, this type of virtualization can only be done at the kernel level via file system filter drivers, where we can capture all relevant events, modify them, or redirect them as necessary. The routines we have captured and virtualized are: file creation, file read, file write, change. First, let's summarize what routines should be captured and their interactions.

PreCreateFile

In **PreCreateFile** routine, according to input parameters, there are two cases: create a new file or open an existing file.

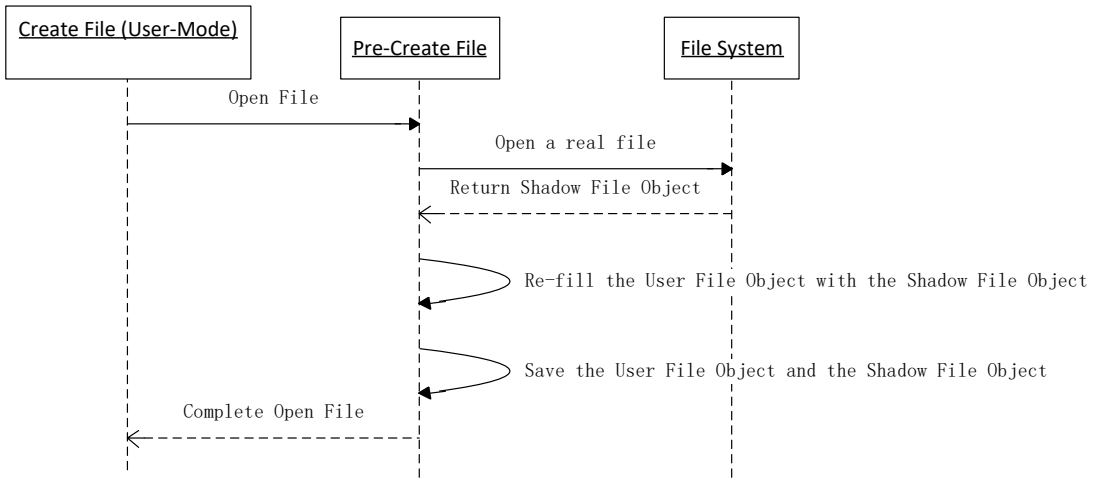
Create File

The following figure shows the interaction on file creation virtualization:

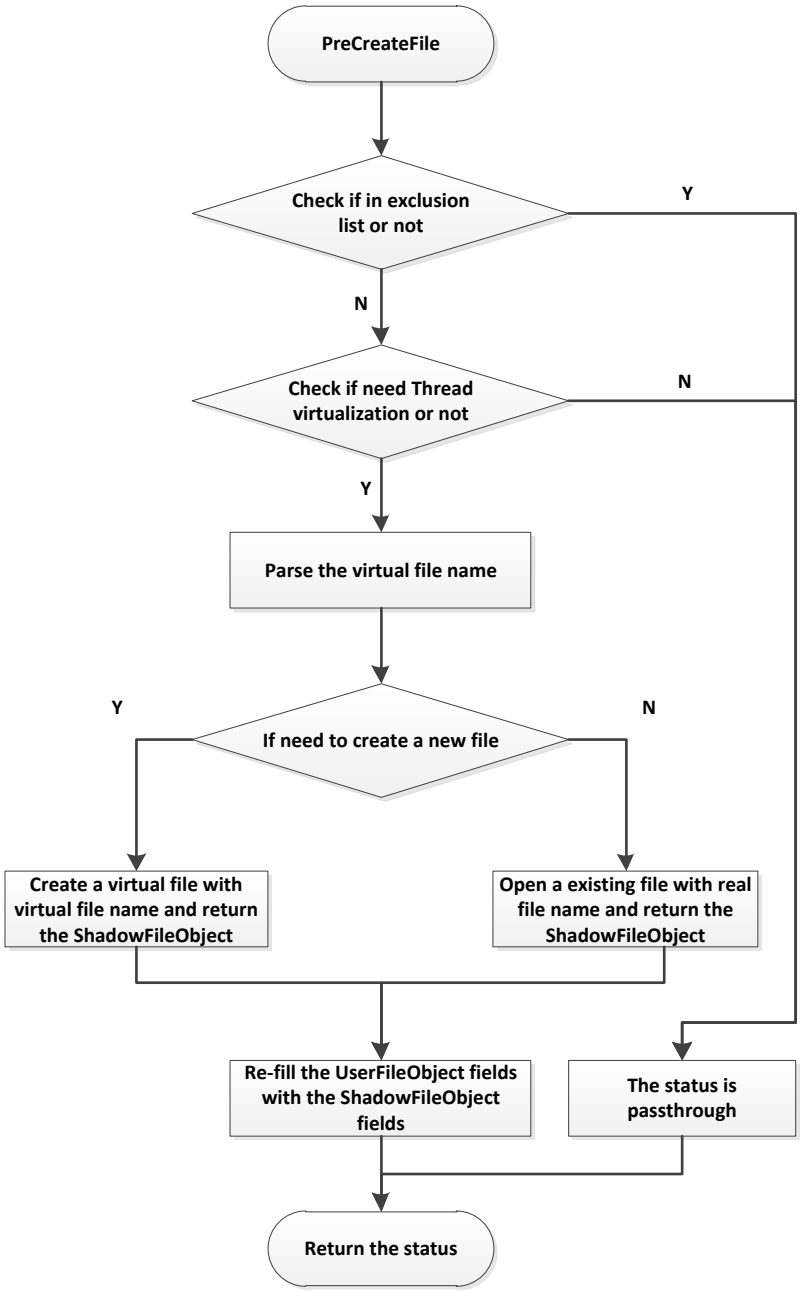


Open File

The following figure shows the interaction on file opening virtualization:



The following figure shows the main flow of the create file operation:



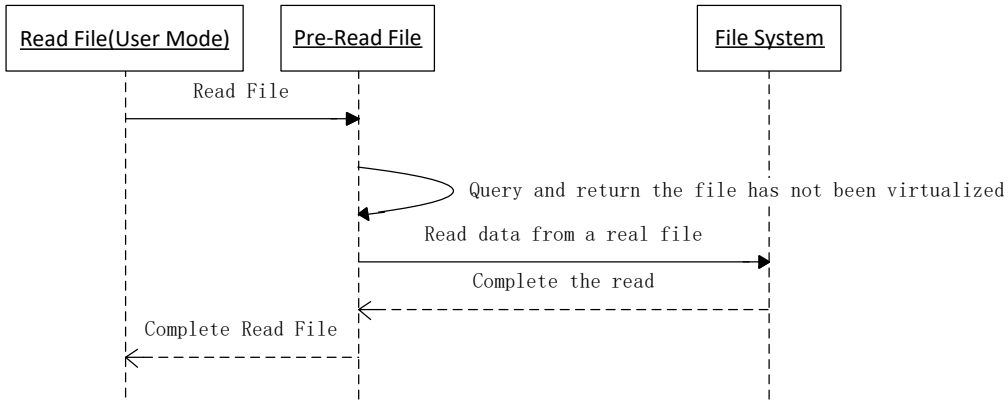
As shown in the figure, the **PreCreateFile** routine summarizes in the following list:

7. Check if the file path is in the **exclusion list** or not. If it is in the list, bypass. Otherwise go to step 2.
8. Check if the thread which creates or opens the file needs to be virtualized. If it needn't to be virtualized, bypass. Otherwise go to step 3.
9. Retrieve the full path information for the accessed file, and parse the virtual file name.
10. Check the operation is to create a new file or opens an existing file. If it is to create a new file, go to step 5. Otherwise go to step 6.
11. Create a virtual file in the **virtual file storage**, and return the ShadowFileObject which points to the virtual file. Re-fill the UserFileObject fields with the ShadowFileObject fields.

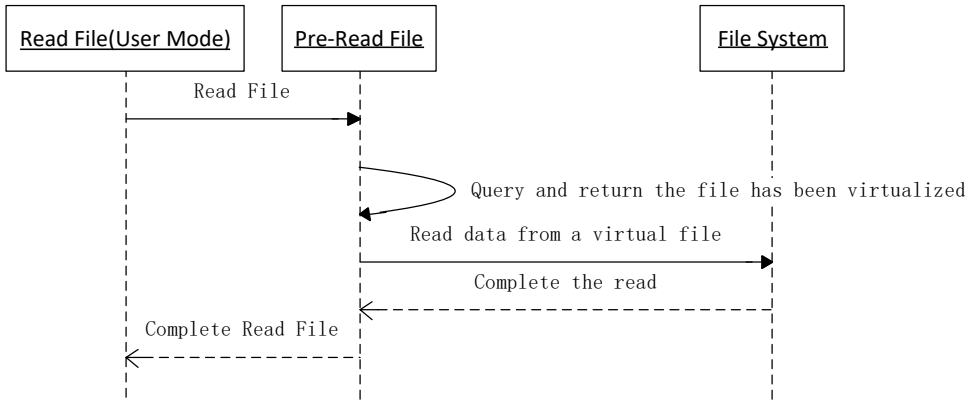
12. Open an existing file, and return the ShadowFileObject which points to the real file. Re-fill the UserFileObject fields with the ShadowFileObject fields.

PreReadFile

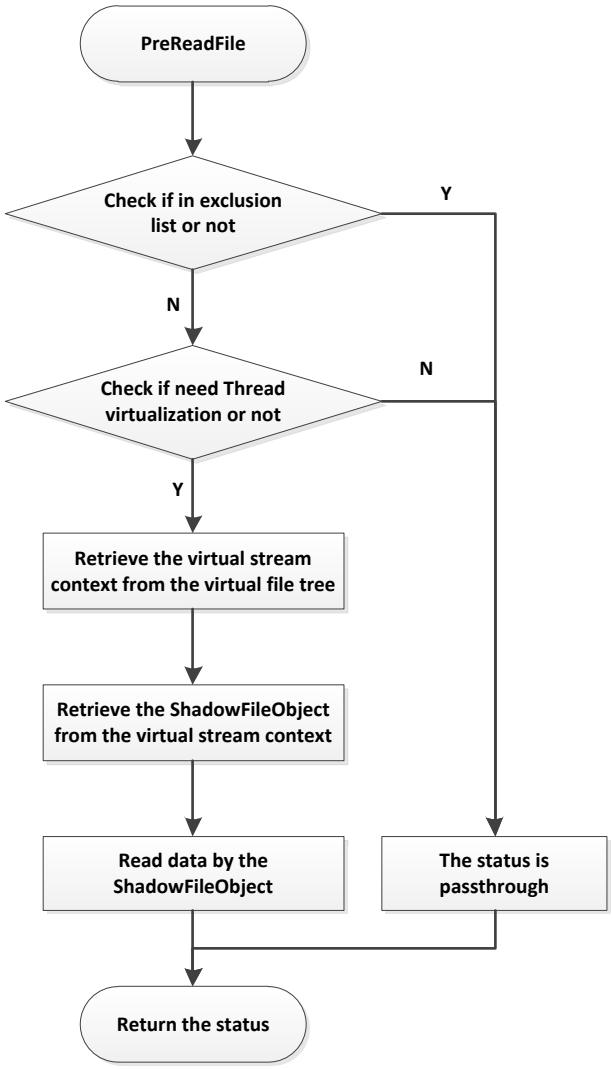
The following figure shows the interaction on file reading virtualization (The file has not been virtualized).



The following figure shows the interaction on file reading virtualization (the file has been virtualized).



The following figure shows the main flow of the read file operation:

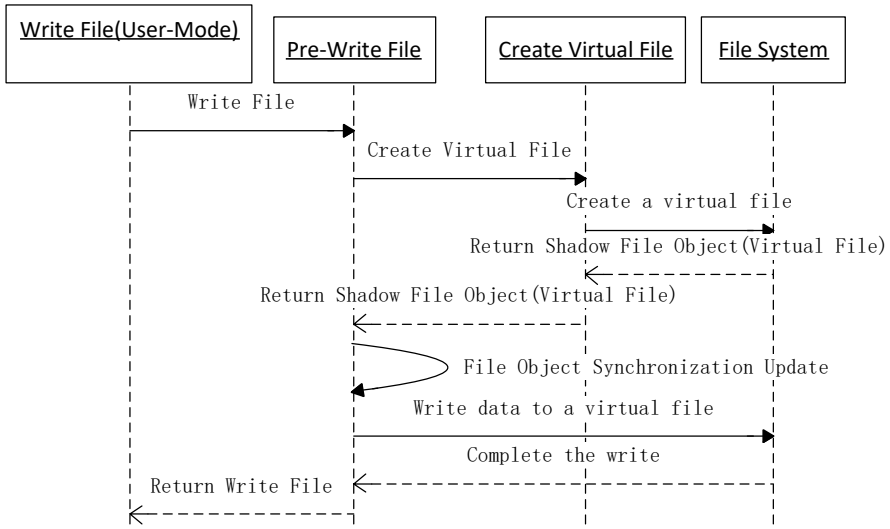


As shown in the figure, the **PreReadFile** routine summarizes in the following list:

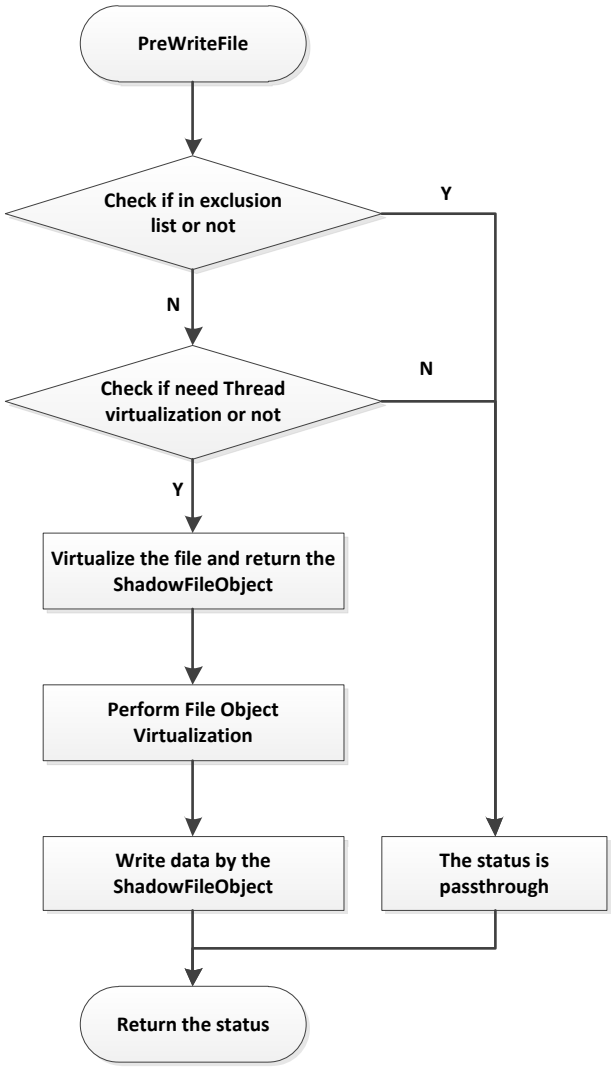
1. Check if the file path is in the **exclusion list** or not. If it is in the list, bypass. Otherwise go to step 2.
2. Check if the thread which reads from the file needs to be virtualized. If it needn't to be virtualized, bypass. Otherwise go to step 3.
3. Retrieve the SB_VIRTUAL_STREAM_CONTEXT instance from the **virtual file tree**.
4. Retrieve the ShadowFileObject from the SB_VIRTUAL_STREAM_CONTEXT instance.If the file has been virtualized, the ShadowFileObject points to the virtual file. Otherwise it points to the real file.
5. Read data by the ShadowFileObject.

PreWriteFile

The following figure shows the interaction on file writing virtualization.



The following figure shows the main flow of the write file operation:



As shown in the figure, the **PreWriteFile** routine summarizes in the following list:

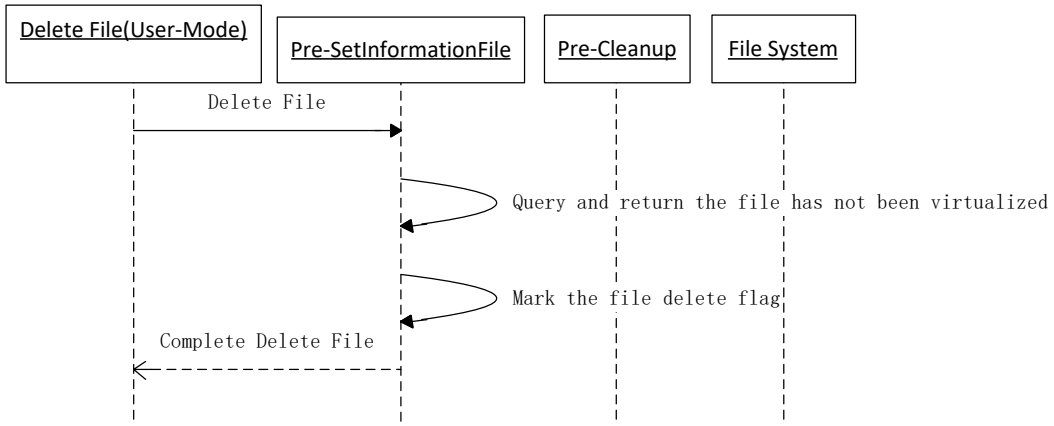
1. Check if the file path is in the **exclusion list** or not. If it is in the list, bypass. Otherwise go to step 2.
2. Check if the thread which writes to the file needs to be virtualized. If it needn't to be virtualized, bypass. Otherwise go to step 3.
3. Create a virtual file in the **virtual file storage** and return the ShadowFileObject. The ShadowFileObject points to the virtual file.
4. Perform file object virtualization. The file objects which have been opened before re-points to the virtual file.
5. Write data by the ShadowFileObject.

PreSetInformationFile

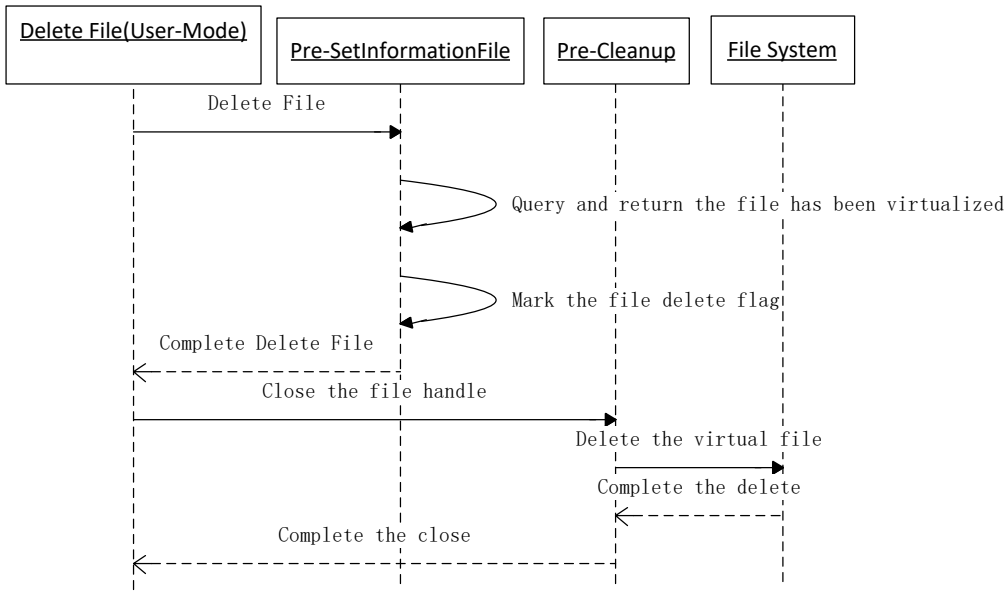
In **PreSetInformationFile** routine, according to input parameters, there are two cases: delete a file or rename a file.

File Deletion

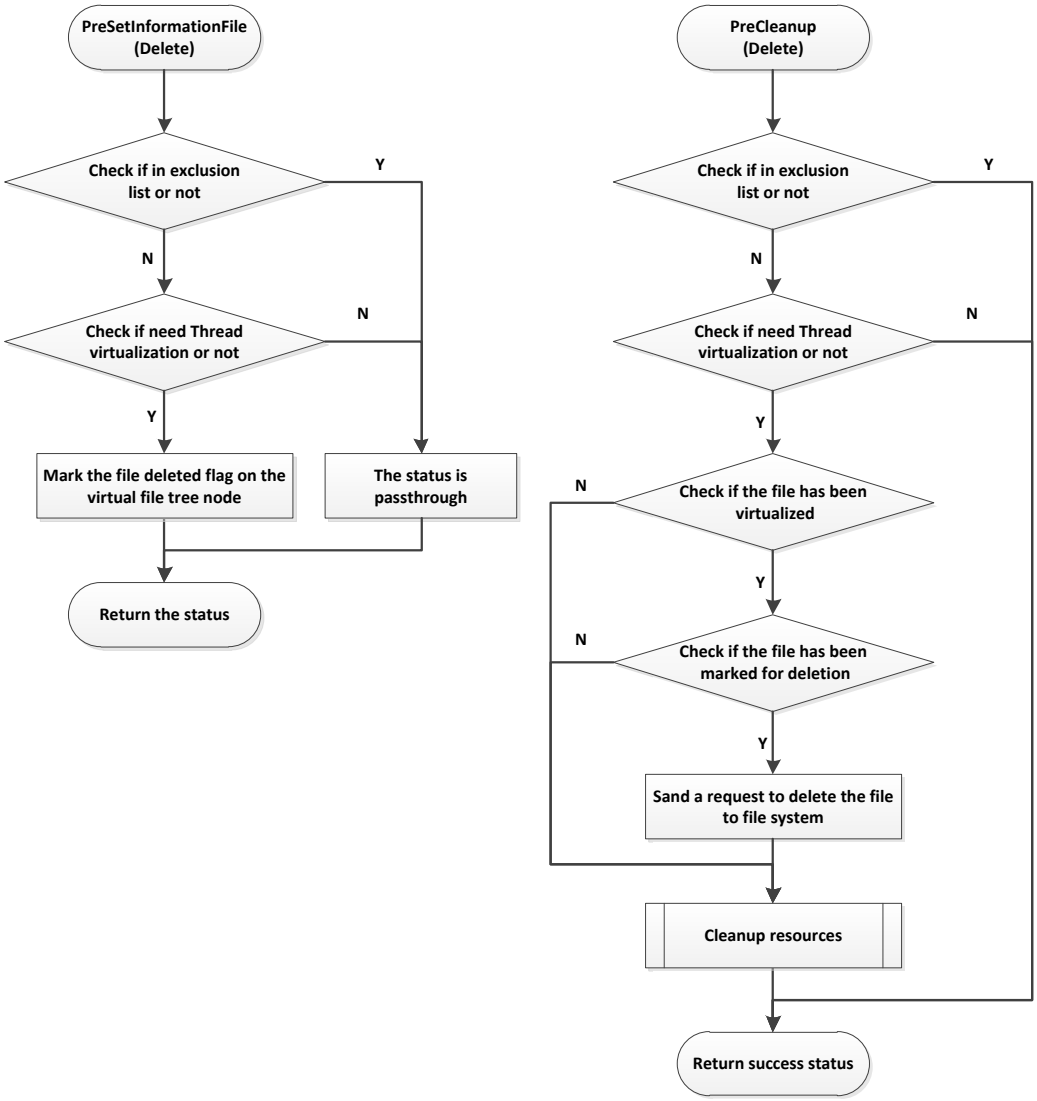
The following figure shows the interaction on file deletion virtualization (the file has not been virtualized).



The following figure shows the interaction on file deletion virtualization (The file has been virtualized).



The following figure shows the main flow of the delete file operation:

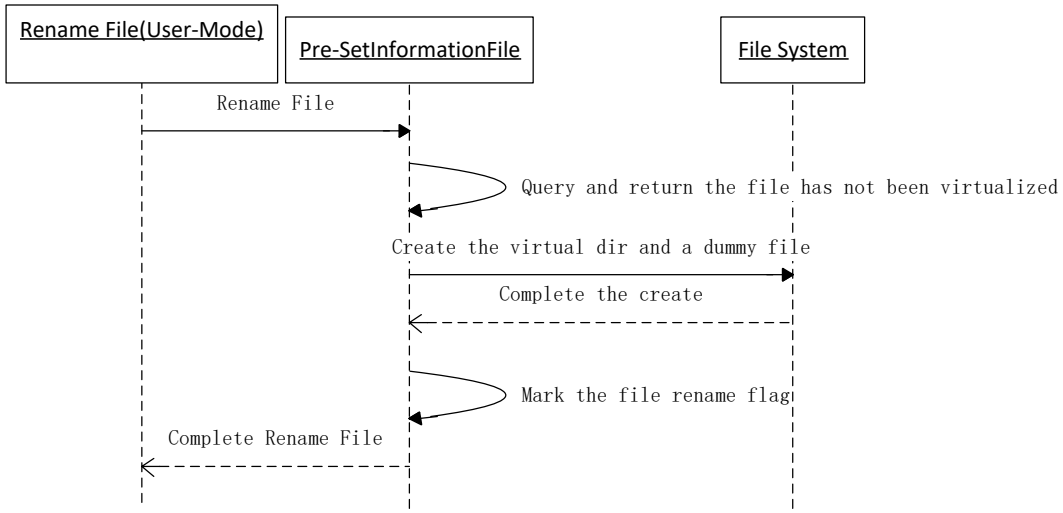


As shown in the figure, the **PreSetInformationFile(Delete)** routine summarizes in the following list:

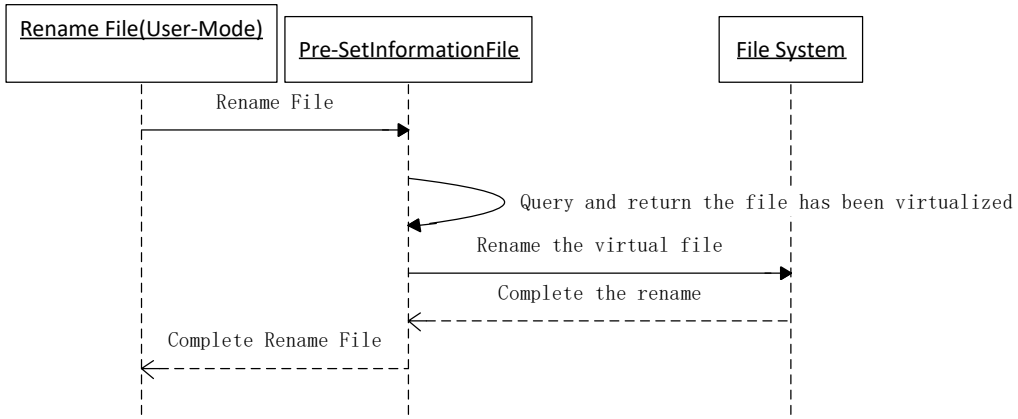
1. Check if the file path is in the **exclusion list** or not. If it is in the list, bypass. Otherwise go to step 2.
2. Check if the thread which deletes the file needs to be virtualized. If it needn't to be virtualized, bypass. Otherwise go to step 3.
3. Mark the file delete flag on the **virtual file tree** node.
4. In **PreCleanup** operation, if the file has been virtualized and has been marked for deletion, then sand a request to delete the file to file system.

File Renaming

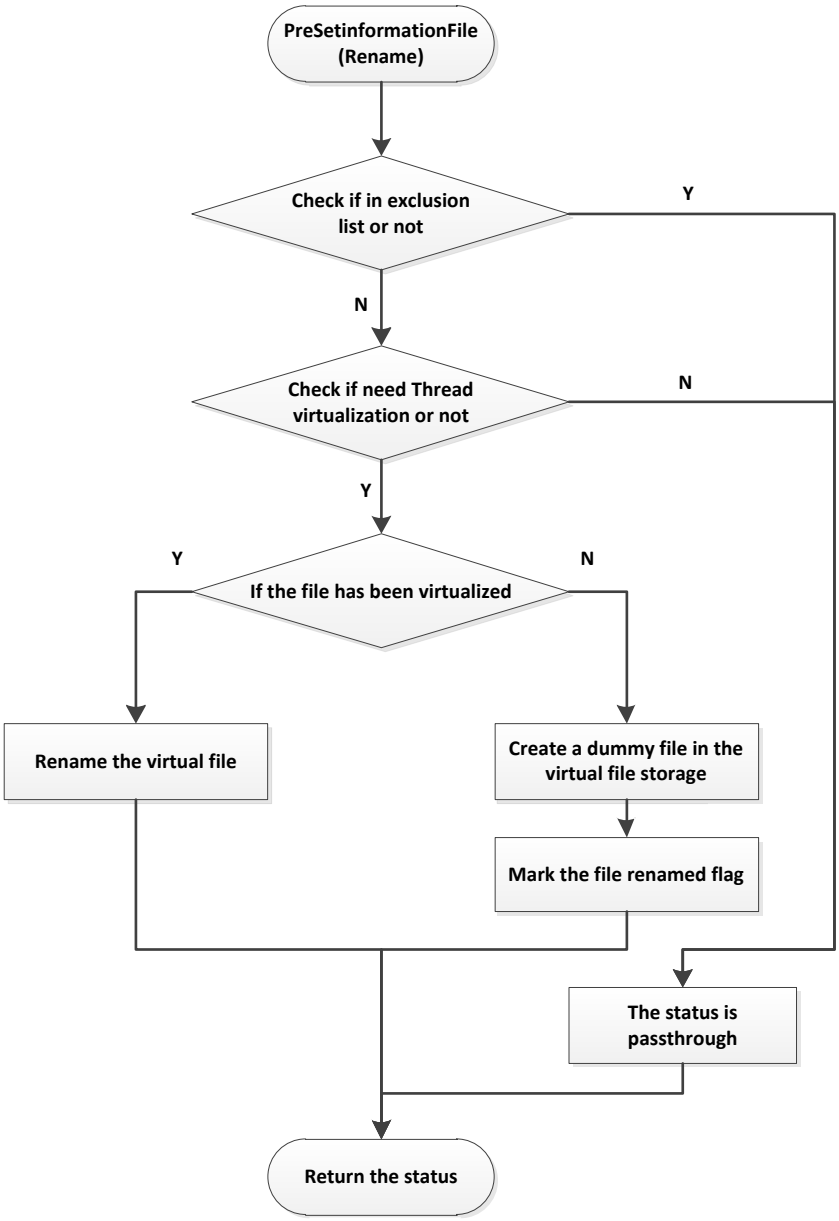
The following figure shows the interaction on file renaming virtualization (the file has not been virtualized).



The following figure shows the interaction on file renaming virtualization (The file has been virtualized).



The following figure shows the main flow of the rename file operation:

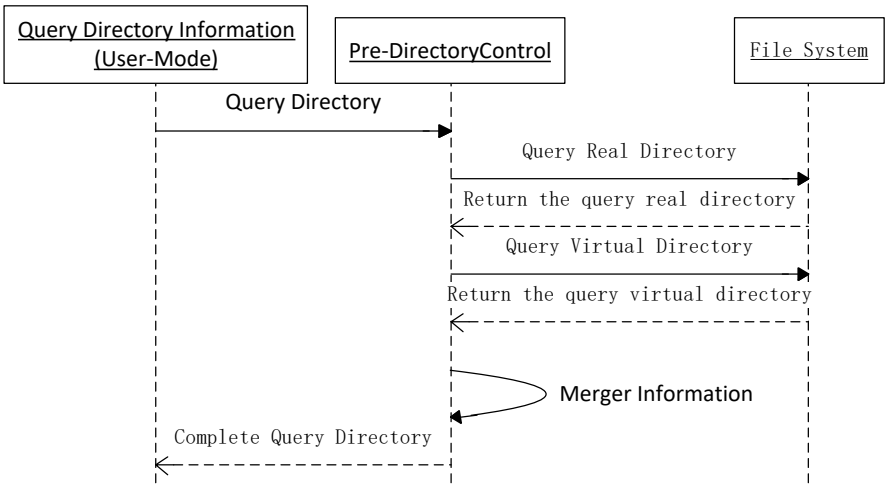


As shown in the figure, the **PreSetInformationFile(Rename)** routine summarizes in the following list:

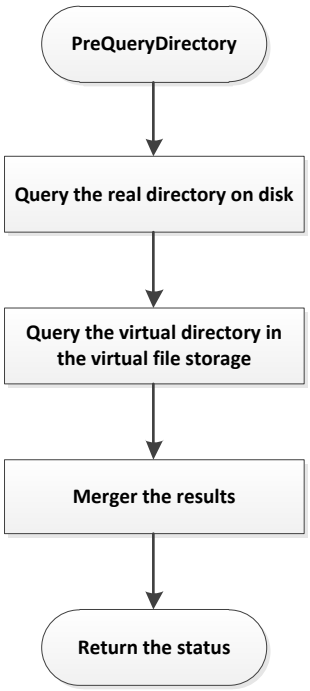
1. Check if the file path is in the **exclusion list** or not. If it is in the list, bypass. Otherwise go to step 2.
2. Check if the thread which renames the file needs to be virtualized. If it needn't to be virtualized, bypass. Otherwise go to step 3.
3. If the file has been virtualized, rename the virtual file. Otherwise create a dummy file in the **virtual file storage**. The dummy file is not a virtual file. It has the same attributes and size with the real file. It is only a placeholder for the **PreDirectoryControl** operation. The dummy file can improve the performance for the rename operation because it doesn't copy data from the real file and only set the file attributes and size to the dummy file.

PreDirectoryControl

The following figure shows the interaction on directory control virtualization:



The following figure shows the main flow of the directory control operation:

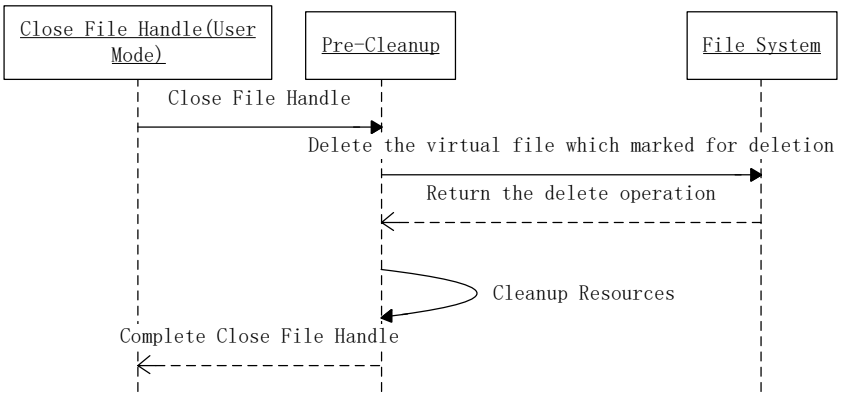


As shown in the figure, the **PreDirectoryControl** routine summarizes in the following list:

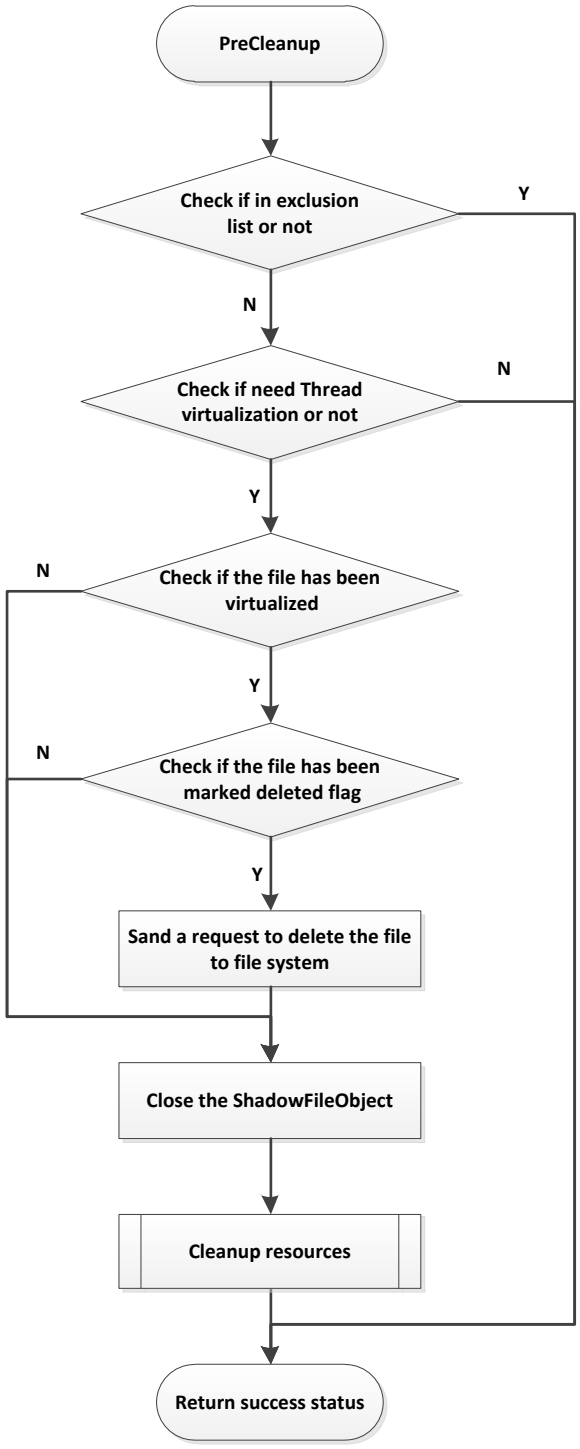
1. Query the real file information in the real directory on disk.
2. Query the virtual file information in the **virtual file storage**.
3. Merger the query results (include dummy files and exclude duplicate files).

PreCleanup

The following figure shows the interaction on file cleanup virtualization.



The following figure shows the main flow of the cleanup operation:



As shown in the figure, the **PreCleanup** routine summarizes in the following list:

1. Check if the file path is in the **exclusion list** or not. If it is in the list, bypass. Otherwise go to step 2.
2. Check if the thread which operates cleanup needs to be virtualized. If it needn't to be virtualized, bypass. Otherwise go to step 3.

3. If the file has been virtualized and has been marked for deletion, then sand a request to delete the file to file system.
4. Close the ShadowFileObject which is created in the **PreCreateFile** operation.

Cleanup other resources.

Registry Virtualization:

Registry virtualization is handled similarly to File System Virtualization on the Filter Driver level. We capture all necessary registry events.

Xcitium, formerly known as Comodo Security Solutions, is used by more than 3,000 organizational customers & partners around the globe. Xcitium was founded with one simple goal – to put an end to cyber breaches. Our patented Xcitium Essentials ZeroDwell technology uses Kernel-level API virtualization to isolate and remove threats like zero-day malware & ransomware before they cause any damage to any endpoints. ZeroDwell is the cornerstone of Xcitium's endpoint suite which includes pre-emptive endpoint containment, endpoint detection & response (EDR), managed detection & response (MDR), and managed extended detection and response (M/XDR). Since inception, Xcitium has a track record of zero breaches when fully configured.

AWARDS & RECOGNITION

OUR CUSTOMERS

SALES

US: 646-569-9114

CA: 613-686-3060

EMAIL

sales@xcitium.com

support@xcitium.com

VISIT

200 Broadacres Drive,
Bloomfield, NJ 07003
United States