In the following I’m providing some consideration how the filesystem event monitoring facility *fanotify* can be used.

*fanotify* is a file system event monitoring facility that can be used to implement hierarchical storage management applications. Compared to dmapi, *fanotify* provides much less functionality:

* The different type of events that can be monitored is much less compared to that what is possible with dmapi.
* There is no mandatory locking of file system objects. Since also Linux does not provide a useful way to perform mandatory locking I do not see a possible way to do that.
* There does not exist a concept of managed regions. It is possible to enable event monitoring for a whole filesystem or also for specific file system objects. What I have seen so far this enablement is not persistent. If you e.g. migrate a file and enable event monitoring for that file this event monitoring only is performed as long your data management application is running. The only useful way from my perspective is to perform event monitoring for a whole file system and store migration states within extended attributes.
* There are no invisible read or write operations (dm\_read\_invis(), dm\_write\_invis()) available that do not modify the time stamp of a file system object. Time stamps must be reset accordingly after a migration or recall operation happened.
* There does not exist a similar concept like invisible dmapi-attributes. If extended attributes are used instead these attributes might be visible to a user.

The following events can be monitored as described by the *fanotify* man-page:

|  |  |
| --- | --- |
| FAN\_ACCESS | A file or a directory (but see BUGS) was accessed (read). |
| FAN\_OPEN | A file or a directory was opened. |
| FAN\_MODIFY | A file was modified. |
| FAN\_CLOSE\_WRITE | A file that was opened for writing (O\_WRONLY or O\_RDWR) was closed. |
| FAN\_CLOSE\_NOWRITE | A file or directory that was opened read-only (O\_RDONLY) was closed. |
| FAN\_Q\_OVERFLOW | The event queue exceeded the limit of 16384 entries.  This limit can be overridden by specifying the FAN\_UNLIMITED\_QUEUE flag when calling fanotify\_init(2). |
| FAN\_ACCESS\_PERM | An application wants to read a file or directory, for example using read(2) or readdir(2).  The reader must write a response (as described below) that determines whether the permission to access the filesystem object shall be granted. |
| FAN\_OPEN\_PERM | An application wants to open a file or directory.  The reader must write a response that determines whether the permission to open the filesystem object shall be granted. |

Only the last two events are synchronous, the other events will not block the calling application. E.g. read calls are possible to block but write calls are not. There are no events provided when file system mounts or unmounts occur. This might be necessary.

Monitoring read and write events synchronously would be the best in respect to a less complex design. For usual HSM implementations there exist three different types of migration states:

* migrated: the data is on tape (if tape is the target tier to migrate to) and no data on disk.
* premigrated (also called co-resident): the same data is on tape and disk. The benefit of this state is that a file in this state is still readable but also quickly to migrate since there is no data transfer necessary.
* resident: the data only is on disk.

A differentiation between read and write events is important to determine the target when/after a file is recalled (migrated data is written back to disk): if a file only is read the target state is preferably premigrated while if the file is written it needs to be resident.

Regarding space management applications like Spectrum Protect HSM or Spectrum Archive EE: if a migrated file is read or written these calls get blocked as long as the data is transferred back to disk (synchronous event). Regarding *fanotify* this write event is asynchronous. When receiving the event the write already happened. There is no way to before transfer back the data. Only dispositioning read and write events would be not sufficient.

Since a whole filesystem is monitored events are generated also for those files that are resident. Using dmapi for a data management application it is possible to set managed regions for files that a migrated and premigrated. If a file is recalled to premigrated or resident state and the managed region is changed accordingly, no further read or write event will occur for this. For *fanotify* I do not see a way to not being notified for all read and write calls. If a filesystem is heavily in use, there might be too many of them. Therefore, to enable monitoring for read and write calls might not be appropriate.

Instead of being notified for read and write calls in my proposal I disposition the following events:

|  |  |
| --- | --- |
| FAN\_OPEN\_PERM: | Using this event to block open calls to transfer back the data. |
| FAN\_CLOSE\_WRITE: | Using this event to move the file state to resident. |
| FAN\_CLOSE\_NOWRITE: | Using this event to move file state to premigrated. |

The open event need to be responded. There are two possibilities:

* FAN\_ALLOW, or
* FAN\_DENY

The latter one causes an error within the calling application.

There are disadvantages to depend on the open event:

* If a file is opened it is cannot be expected that data is read or written thereafter. An application might perform other operations than to read or write to a file. In the current proposal data is recalled whenever an open call is issued on a migrated file. Doing so unnecessary recalls might happen.
* If a file has been opened for write it cannot be expected that data is written thereafter. An application might open a migrated file for write but only read data. In this case, such a file would be transferred to resident state where a premigrated state would be appropriate.

Furthermore, if data has been recalled after an open event has been received thereafter it is not clear in which state to move that file until a corresponding close event has been received. It will be necessary to introduce a transition state for that purpose which only indicates that a file is not migrated anymore. The same topic is relevant after opening a premigrated file. After that it is not known if the file will be read – and therefore the premigrated state can be kept – or if the file will be written and therefore moved to the resident state.

Therefore, there are further migration states need to be introduced beside migrated, premigrated, and resident:

* migrating: there is a data transfer from disk to tape in operation.
* recalling: there is a data transfer from tape to disk in operation.
* not-migrated: a migrated file where it’s data already has been transferred back or a premigrated file has been opened and a close event not happened so far.

The plan is to store the migration states within an extended attribute of a corresponding file. This attribute might be visible to a user but I do not see another way to do so (there is nothing available like invisible dmapi attributes). The following identifiers will be used for the migration states:

* MIGR (migrated)
* IMIG (migrating)
* NMIG (not-migrated)
* IREC (recalling)
* PMIG (premigrated)

For the resident case this attribute is removed.

If a file should be migrated it needs to be verified that it has not been opened before by another application. If an application opens a file in resident state, OpenLTFS will receive an open event and respond to that immediately to let the application continue. It can happen that the application keeps this file open for a while without reading or writing to it. Migration should be not allowed during this time. If a migration happens during this timeframe the application will be able write to a empty stub file (file after migration). This write call cannot be interfered.

OpenLTFS needs to store the number of file accesses (open events) in a small table containing the inode number and the number of open file descriptors on that inode. A migration only can happen if this number is zero.

Within the flow chart below this access information is indicated by the following object:



The following flow chart is showing the possible state changes after an open call (start with FAN\_OPEN\_PERM) happened:

