Trash Can/Undelete for Lustre

- 1. Introduction
  - 1.1. Design and Implementation
    - 1.1.1. Configuration for the Trash Can
    - 1.1.2. Delete a file into the Trash Can
    - 1.1.3. Delete a directory into the Trash Can
    - 1.1.4. List "undeleted" files within a Trash Can
    - 1.1.5. Deleting a file or directory in the Trash Can
    - 1.1.6. Empty a Trash Can:
    - 1.1.7. Restore a file from the Trash Can
  - 1.2. Space and Quota Accounting
  - 1.3. Clean up files from the Trash Can
  - 1.4. Per-User Trash Can
  - 1.5. Per-Tenant Trash Can
  - 1.6. Repeated deletion of same filename
  - 1.7. Avoid preserving temporary files
  - 1.8. JobID of process deleting a file
  - 1.9. Backup and Restore of MDT with Trash Can
  - 1.10. Trash Can and fscrypt Files and Directories
- 2. Virtual .Trash Directory Support
  - 2.1. Virtual .Trash directory
  - 2.2. .Trash pFID name lookup
  - 2.3. .Trash striped directory
  - 2.4. .Trash directory migration
  - 2.5. Orphans in Trash Can
- 3. Reclaim TCU Space
  - 3.1. ltrash_purge Tool (LU-19598)
  - 3.2. TCU Reclaim Upcall (LU-19950)

1. Introduction

If files are accidentally or maliciously deleted from a file system, the user data may be permanently lost. The Trash Can is a useful feature in file systems that acts as a temporary holding area, allowing users to store deleted files for a short time before they are permanently deleted. It provides a mechanism to restore or retrieve deleted files if needed, and automatically deletes the files once they become too old or the filesystem is too full.

When the Trash Can feature is enabled, when a user deletes a file from a file system, it is not immediately deleted but moved to the Trash Can. Deleted files and directories are temporarily stored in the Trash Can. Files and directories in the Trash Can may be restored or retrieved individually or in bulk if they are still available. The Trash Can may be manually emptied, or once the filesystem is nearly full the system will automatically empty files from the Trash, taking into account which users and projects are consuming the most space.

The Trash Can should including the following functionalities:

- Files should be added to the Trash can during normal usage (e.g. `rm` or `rmdir` utility or `unlink()` and `rmdir()` syscalls, or if a file is renamed onto another one).
- There should be a per-UID space for files in the Trash Can, so that administrative tools can easily find files for each user
- Restore a file in the Trash Can. This will restore a file or directory to its original path. The corresponding User, Group, and Project quota account should also be increased again.
- List "undeleted" files in the Trash Can.
- After a file is deleted and moved into Trash Can, the User, Group, and Project quota for this file should be accounted and reduced accordingly.
- A file in the Trash Can is not visible in the namespace of the file system.
- A file or directory in the Trash Can is marked with a `TRASH` flag, so that tools like `lipe_find3` can optionally skip/find deleted files.
- A file in the Trash Can marked with the `TRASH` flag cannot be read by a user or application, to prevent applications continuing to read these files.
- Permanently remove a file in the Trash Can. This will remove the file from the Trash can and destroy the OST objects to free the used space. The file is now unrecoverable.
- Empty the Trash Can. This will remove all files in the Trash Can.
- Files should be deleted from Trash Can after a specified retention period, such as 7 days.
- Files should be deleted from Trash Can when an OST approaches a capacity threshold (over 80% for HDD, or 90% for SSD) to avoid performance impact or the risk of running out of space if a large number of files are written at once.
- Have a tunable parameter to enable/disable Trash Can feature on a entire file system.
- A administrator can enable/disable Trash Can feature on a specified file or directory by setting the `NOTRASH` flag on the file.

Deleted files can no longer be restored from the Trash Can when:

- A file (or directory) is deleted from the Trash Can. In other words it have been deleted twice. The first deletion only moves the file to the Trash Can. The second deletion actually removes the file from the file system.
- The Trash Can is emptied of all of its contents.

1.1. Design and Implementation

The design for the Trash Can feature in Lustre is relatively straight forward.

On the server side, the MDS implements the basic functionalities such as moving the "deleted" files into the Trash Can, and the interface how to traverse them. On the client side, a basic utility tools to interact with the Trash Can ("`lfs trash set|clear|list|delete|restore FILE|DIR`"), are available for debugging and low-level operations on the Trash Can including:

- Set or clear the `TRASH` flag on a given file or directory (this is implemented with the `FS_UNRM_FL` inode attribute so that deleted files can be identified with `lsattr` in the filesystem).
- list files in the Trash Can on a given MDT
- Permanently delete a file or directory within the Trash Can on a given MDT
- Empty the Trash Can on a given MDT
- Restore a file within the Trash Can on a give MDT

These low-level commands will generally not be used by the end user, since the virtual `.Trash` directory will be more readily accessible to users, including those in restricted subdirectory mounts in a tenant namespace.

This implementation only moves the regular files or subdirectories into the Trash Can upon its last unlink. This means that deleted hard links will not be preserved by default, in order to minimize the overhead of managing the Trash Can.

Deleted subdirectories are only preserved if they have deleted files currently in the Trash Can. Empty subdirectories are not currently preserved since they can be trivially recreated, though this could potentially be implemented with a tunable parameter if there was a need for it.

The implementation borrows ideas from orphan and volatile files in Lustre, which normally stores deleted files in the "`ROOT/PENDING`" directory on each MDT. After the initial setup and mount, each MDT creates a "`ROOT/.lustre/Trash/MDT`*`xxxx`*" directory as a Trash Can to store deleted files, if it does not already exist.

1.1.1. Configuration for the Trash Can

- An administrator can enable/disable Trash Can feature globally on a specified MDT via: `lctl set_param mdd.*.trash_can_enable`
- The UID/GID/PROJID of files in the Trash Can are configured globally via `mdd.*.trash_can_uid`, `mdd.*.trash_can_gid`, and `mdd.*.trash_can_projid`, see ***Space and Quota Accounting*** below for details

1.1.2. Delete a file into the Trash Can

When a file or empty subdirectory is deleted (last link in namespace is removed) a number of steps are performed for the file. Some of them are "one time only" for the user or directory, while others are done for each file

- Create a subdirectory in the `.lustre/Trash/MDT`*`xxxx`* directory for the user to hold all of their deleted files. This is done only once per UID in the filesystem. See ***Per-User Trash Can*** below for details.
- Create a *stub* subdirectory for the parent directory where the file is being deleted.
  - The *stub* directory is named with the FID of the parent directory (***pFID***) where the file is being deleted from. The stub directory will have its own FID (***stFID***) that is unrelated to *pFID*, but needs to be unique to ensure this directory can be accessed by the client independently of the parent directory.
  - To reduce repeated lookups of the `ROOT/.lustre/Trash/MDT`*`xxxx/UID/pFID`* directory when deleting many files from a parent directory, the original parent object may cache the *stFID* object (by FID or pointer?) since the mapping to the stub directory will remain unchanged for the lifetime of the parent.
  - It would be desirable to copy the parent directory name from its `trusted.link` xattr to the stub, so that it is possible to generate a user-friendly name for the stub directory when it is shown in the `.Trash/` subdirectory. It is likely best if the `trusted.link` xattr contain the FID of the *`UID/`* directory as the parent FID rather than make an exact copy of `trusted.link`, so that LFSCK does not try to re-link the stub into the parent directory.
- The deleted file will be moved into the stub directory. As a part of this transaction, several changes will be applied to the deleted inode:
  - the inode will be marked with the `FS_UNRM_FL` attribute to indicate that it was deleted. This allows it to be identified as "in the Trash" by scanning utilities, regardless of the UID/GID/PROJID assigned.
  - the UID/GID/PROJID will be changed to the `trash_can_uid`, `trash_can_gid`, and `trash_can_projid` to remove it from the quota accounting of the original user. See ***Space and Quota Accounting*** below for details.
  - a "`trusted.unrm`" XATTR is added on the deleted file. The XATTR contains the following information:

```
struct ll_trash_xattr {
   __u32 ltx_flags;      /* for future usage */
   __u32 ltx_uid;        /* original UID of the file, used to restore on unrm */
   __u32 ltx_gid;        /* original GID of the file, used to restore on unrm */
   __u32 ltx_projid;     /* original PROJID of the file, used to restore on unrm */
   __u64 ltx_timestamp;  /* time the file moved into the Trash Can, or we could use ctime here? */
};
```

Where `ltx_uid`/`ltx_gid`/`ltx_projid` are the original UID/GID/PROJID of the deleted file, mainly used for the restore operation. `ltx_timestamp` is the time that the file was moved into the Trash Can. It is used to determine whether the file is expired for the specified retention period and thus should be purged from the Trash Can. It may be to use the inode ctime for this purpose instead of storing a separate timestamp to reduce the size of the xattr.

1.1.3. Delete a directory into the Trash Can

When a directory is deleted into the Trash Can, it is desirable to preserve the directory hierarchy of the original directory tree, so that accidental "`rm -rf`" (or equivalent) does not result in millions of files or directories in the top level of the `.lustre/Trash/MDT`*`xxxx/UID`* directory. The directory deletion will perform the following actions:

- As with regular file deletion, a *stub* directory is created with the name of the FID of the directory's parent (***ppFID***).
- The directory's own FID is looked up in the `.lustre/Trash/MDTxxxx/UID/` directory to determine if a *stub* directory with the `pFID` name already exists or not.
- The xattrs on the deleting directory (e.g ACLs, selinux, etc.) are copied over to the pFID stub
- If the deleting directory's pFID stub exists then it is renamed to match the name of the directory and moved into the new ppFID stub directory of its parent.

By renaming and moving each pFID stub directory into the corresponding ppFID stub directory, this preserves the hierarchy of a deleted directory tree. The "`rm -rf`" process is processing files and subdirectories in a "bottom up" manner, so it needs to build up the deleted directory hierarchy incrementally as files are deleted.

There is no single "delete directory tree" command in POSIX, since that may normally take a very long time to complete while processing billions of files. With Trash Can it may be desirable to offer such an interface, since the whole directory tree could be moved into the Trash Can in a single operation. (it would necessitate background operations to annotate files with the `FS_UNRM_FL` attribute, store the original UID/GID/PROJID into the `trusted.unrm` xattr, and change the UID/GID/PROJID into the `trash_can_*` equivalents. This would still be more efficient than deleting (renaming) thousands or millions of individual files and subdirectories.

1.1.4. List "undeleted" files within a Trash Can

- The `.lustre/trash/MDT`*`xxxx`* (where *xxxx* is the hexadecimal MDT index) directory tree is local to each MDT. By this way, users can access the "undeleted" files with readonly mode under the Trash Can directory on a given MDTnnnn via POSIX file system API. However, we can not access these files from fileset sub directory mount. We can perform the following commands from a Lustre namespace (mount point of "`/mnt/lustre`") on a client:

```
# ls /mnt/lustre/.lustre/Trash/MDT0002

0x200034021:0x1:0x0
0x200034021:0x2:0x0
...
# lfs trash ls /mnt/lustre/.lustre/Trash/MDT0002/0x200034021:0x1:0x0
0 0 4096 Nov 14 08:11 [0x200034021:0x1:0x0]->/mnt/lustre/f1
# lfs trash list /mnt/lustre/.lustre/Trash/MDT0002
MDT index: 1
uid gid   size delete time FID                    Fullpath
0 0 4096  Nov 14 08:11     [0x200034021:0x1:0x0]->/mnt/lustre/f1
0 0 32104 Nov 14 08:07     [0x200034021:0x2:0x0]->/mnt/lustre/dir/f2
```

- By default it will list files/directories deleted relative to the current working directory. If `DIR` is provided, then list deleted files/directories relative to that directory, in the same format as `ls`:

```
# lfs trash {list|ls} [DIR|FILE]
MDT index: 1
uid gid size delete time  FID                    Fullpath
0   0   4096 Nov 14 08:11 [0x200034021:0x1:0x0]->/mnt/lustre/f1
0   0  32104 Nov 14 08:07 [0x200034021:0x2:0x0]->/mnt/lustre/dir/f2
...
```

  Internally, the `lfs trash list` command is looking up the FID and MDT of the current directory, or the directory specified by `DIR`, and then listing the respective directory under `$MOUNT/.lustre/trash/MDTxxxx/pFID/` or the directory file descriptor returned via `llapi_open_by_fid()` if the `.lustre/trash` directory is not available. This is mainly for debugging, since users will generally use the virtual `.Trash` directory to interact with the Trash Can and restore files.

1.1.5. Deleting a file or directory in the Trash Can

- To remove the temporary file under "`ROOT/.lustre/Trash`" and free the data space on Lustre OSTs permanently:

```
# lfs trash {delete|rm} [DIR/]FILE ...
```

1.1.6. Empty a Trash Can:

```
# lfs trash clear DIR ...
```

1.1.7. Restore a file from the Trash Can

- on a given MDT. It will restore the file and its content according to the saved full path and then delete the stub on the Trash Can.

```
# lfs trash {restore|unrm} [DIR/]FILE ...
```

- A utility periodically scans the files under Trash Can directory "ROOT/TRASH" and delete the file with grace time expiration.
- Provide the functionality to scan files in the trash on all MDTs that exceed the specified age manually:

```
# lfs trash find -ctime +time [DIR]
```

- Provide the functionality to restore/delete all files within a given directory. This can be achieved by using the command combination of "`lfs trash list`" and "`lfs trash restore`" or "`lfs trash delete`" to filter the files with the full path attribute under a given directory.

1.2. Space and Quota Accounting

In order to separate space and quota accounting for a user, group, or project's files, the original UID, GID, and PROJID of the file cannot be used for files in the Trash Can. Otherwise, there would be confusion on the part of the user when they delete files and their quota usage does not decrease. Similarly, the free and used space and inodes reported by `df` should not contain the space consumed by files in the Trash Can, since users would be confused by the fact that deleting files does not reduce the amount of space used in the filesystem.

Instead, when files are moved into the Trash Can, their UID/GID/PROJID should be changed to the defined `trash_can_uid`, `trash_can_gid`, and `trash_can_projid` values, and the original values preserved in the `trusted.unrm` xattr on the file. This will "hide" the quota usage from the files in the Trash Can, while still allowing the actual usage of files in Trash to be easily determined via "`lfs quota -u/-g/-p`" commands.

The `df` output should add the space used by the `trash_can_projid` quota to the `statfs.st_bfree` and `statfs.st_bavail`, and add the inodes used by `trash_can_projid` to `statfs.st_ffree` returned by the `statfs()` syscall. This will avoid the issue that "`df`" always shows the filesystem as 90% full (or whatever the space usage threshold is configured to be).

A new option "`lfs df --trash`" should show the *actual* space usage for the filesystem, so that it is possible for an administrator to diagnose issues with the Trash Can space usage.

1.3. Clean up files from the Trash Can

A mechanism is needed to automatically clean up files from the trash can when the filesystem becomes full. It cannot be that the user has to delete every file twice, and it cannot be that the filesystem is allowed to get 100% full (or even 90% full) due to files in the trash. There needs to be an automatic mechanism to clean up the trash to ensure that the filesystem performance does not degrade when users though they deleted files.

In our design, it does not depend on a userspace utility for such a critical function to clean up files from the trash when FS is nearly full, since that utility may never be started, or the client is evicted, or similar. If that happens, the filesystem would become full and unusable, **even though** the user had already deleted files from the filesystem. This needs to be bulletproof and run automatically when the OSTs (or MDTs) are getting full.

The MDS is already monitoring the OST fullness every 5s to make object allocation decisions, so it can also make decisions about files to delete. Therefore, the MDT can periodically monitor the space usage of the trash user (quota) and space usage for the entire file system with the additional consideration of the retention period and deleted timestamp for the files, choose the candidates to be deleted permanently to free up the space.

1.4. Per-User Trash Can

A per-user Trash`/MDTxxxx/`*`UID`*`/` directory that is owned by that *UID* and mode 0700 should always be created in the top-level directory to avoid world readable access to deleted files, and to de-conflict files/directories of the same name created by users (e.g. `tmp/` or `data/` or `Documents/` or similar. That avoids exposing files to other users that may be private, and also allows tracking space usage more clearly for each *UID*, so that a user's data can be found and purged more quickly if they are exceeding their quotas.

In some uncommon cases, it may be that a parent directory has files owned by multiple different users (different UIDs). This would likely only happen for top-level directories like `scratch/` or `home/` that contain directories from multiple users. When those directories were deleted they created separate `.lustre/Trash/MDT`*`xxxx/UID/pFID/`* stub directories. In this case, the deleted parent directory should only be created in the *`.../UID/`* directory with the `inode->i_uid` of the parent directory.

1.5. Per-Tenant Trash Can

Files and directories deleted from within a subdirectory mount of a Nodemap should be stored in a `Trash/MDTxxxx/`*`NODEMAP/UID/`* directory to isolate the files/directories from different tenants. The *`NODEMAP/`* directory name is the configured name of the nodemap for that tenant, and can be found from the client export used to perform the final unlink operation. The *`UID`*`/` directory **name** should be the *client* UID of the user, so that the visible directory name matches the user expectation. The *`UID`* directory **ownership** should be the *server* UID of the user, so that proper file access controls can be maintained. By having the multi-level *`NODEMAP/UID/`* naming, it isolates the `UID` directory names from other tenants that may have the same *mapped* UID directory name.

The Nodemap for a tenant should allow configuring the UID/GID/PROJID to which files in the Trash are assigned, via per-nodemap `trash_can_uid`, `trash_can_gid`, and `trash_can_projid` parameters. These IDs should be within the ID offset range of the Tenant (e.g. 99999) so that they can be accessed and mapped correctly, but are unlikely to cause conflicts with other IDs used by the Tenant. This will also allow the Tenant project quota group to account for **all** space used by the tenancy, while still separating Trash Can usage for the regular UID/GID/PROJID of the Tenant users.

In a multi-tenant environment, it would be desirable to have a more sophisticated policy engine to manage Garbage Collection of files within the trash, in order to provide maximum utility to each Tenant. For example, if Tenant 1 has deleted files an hour ago, but Tenant 2 has written and deleted TB of data since that time, the Tenant 1 files may have expired out of the Trash Can. Developing a complex policy engine to manage GC in an MTFL environment is out of scope for the initial TCU implementation. We likely want to leverage and enhance the `lpurge` utility from Hot Pools to actively monitor the space usage of tenants on different OSTs to decide which objects (files) should be removed.

When running the `df` command, the `statfs()` output should add in the space used by the `trash_can_projid` for the nodemap, so that the space and inode usage reported does not reflect the space used by the Trash Can.

1.6. Repeated deletion of same filename

If the same filename is repeatedly created and deleted within the same parent directory, then the deleted files will have conflicts when moved into the `pFID` directory in the Trash Can. This may happen for files that are edited by the user, and a temporary file like `.FILENAME.tmp12345` is created and written by the editor, and then renamed over the original `FILENAME` (causing it to be deleted) so that the file contents are not lost if the new file is only partially written. In such cases, the same `FILENAME` may be deleted many times.

To disambiguate the files in Trash, the conflicting filenames should be disambiguated by appending a timestamp to the filename, like `FILENAME.2025-04-03T00:11:24` (ISO8601 date format), possibly adding `.microseconds` if there is still a conflict. It isn't totally clear whether it would be better to use the timestamp from when the file was deleted, or when the file was created. Both have some value to help users distinguish between the different versions.

In order to avoid overwhelming the Trash Can with files that are rapidly created and deleted (e.g. short-lived temporary files), it would be desirable to impose an upper limit on the number of versions that will be saved in the trash can. Some complexity exists in implementing this, because the MDS shouldn't need to do a full directory listing to determine if there are multiple versions of a file in the trash.

1.7. Avoid preserving temporary files

Files that only exist for a very short time (e.g. temporary files) should not necessarily be preserved in the Trash Can, or they can quickly overwhelm the available capacity of the filesystem, and result in important files being purged from trash and/or filling the trash faster than files can be cleaned up. Files marked with the `I_LINKABLE` flag on the MDS (from `O_TMPFILE`, or Lustre Volatile files, see LU-18844) should **not** be preserved in the Trash Can if they are not linked into a file in the namespace. It would be useful to have a tunable parameter that sets a *minimum* age for files to be preserved in the Trash Can (e.g. 65 minutes?) so that files that are frequently created and deleted are not preserved since they could consume a considerable amount of space.

1.8. JobID of process deleting a file

In LU-13031 the JobID of the process that first creates a file is stored in the `user.job` xattr on the MDT inode, for diagnostic purposes and to allow determining provenance of each file later on. For the Trash Can, LU-17648 describes storing the JobID of the process that is *deleting* the file, for diagnostics such as determining rogue processes that are deleting files in the filesystem. Something like `user.del` would be a reasonable default xattr name. The actual xattr name can be configured with the `mdt.*.job_xattr_del` parameter.

1.9. Backup and Restore of MDT with Trash Can

For filesystem-level/namespace backup and restore, the `.Trash` directory will not be visible to the backup utility during namespace traversal, so deleted files in the `Trash` will not be backed up. However, if a deleted directory is restored from `Trash` then if it has a new FID (== new inode number) the backup utilities may consider this to be a new directory and back it up again. At a minimum, it would be desirable to preserve the inode number. Similarly, HSM integration may depend on the FID of the file, so it may be desirable to preserve the original parent FID of the directory when it is restored from `Trash`. A deleted directory could only be restored to use the same FID if the original directory had been deleted.

For MDT-level/device backup and restore, the `Trash` directory and its contents will be backed up in the same way as any other directory in the filesystem. The stub `pFID` directory is named by the parent directory FID, so it will be backed up and restored as-is (its FID is not really important, but should be treated normally). The FID on the actual parent directory will be preserved in the `trusted.lma` xattr, but the FID lookup in the restored OI will not be possible until after the initial post-restore OI Scrub has completed. Otherwise, the FID lookup in the restored OI will reference the old inode number, which will reference a wrong or unused inode number/generation, and should return `-EINPROGRESS` or possibly `-ESTALE`.

It may be necessary to add special support to OI Scrub/LFSCK to handle the `Trash` directory on each MDT, so that it can do a top-level scan of the stub `pFID` directories to confirm that the parent FID still references a valid parent directory.

1.10. Trash Can and `fscrypt` Files and Directories

When an `fscrypt` directory is deleted, the encryption context on the directory must be copied from the parent directory to the new pFID stub directory **when it is first created**, so that the encrypted filenames and contents can be accessed properly via the `.Trash/` directory, as well as if an encrypted file or directory is undeleted. The `fscrypt` context is partly derived from the parent directory, as well as a unique per-inode Cryptographic nonce value that ensures the encrypted data is unique, even if the same data is encrypted multiple times.

Users can access files that have been deleted and are located in the recycle bin via ".Trash" with read-only mode, regardless of whether the files are encrypted or not.
Since Lustre performs encryption and decryption entirely on the client side, the server stores all data in plaintext. Therefore, when filename encryption is enabled, if a file with the same name is repeatedly created and deleted within a directory, the server cannot simply append a timestamp to the end of the filename to distinguish between different versions of files with the same name. Doing so would violate the design principle that all filename encryption is performed on the client side, potentially leading to garbled characters in decrypted pathnames and possible incorrect conflicts. For the repeated deletion with the same file name within a directory with file name encryption enabled, the server can only use the policy that always keeps the oldest version of the repeated deletion file.

2. Virtual `.Trash` Directory Support

2.1. Virtual `.Trash` directory

A virtual "`.Trash`" subdirectory accessible in each directory in the filesystem would allow users to easily browse deleted files/directories under the current subdirectory in the Trash Can and access them for recovery.

The FID of the "`.Trash`" directory is derived from the FID of the parent directory (`pFID`), by looking up the corresponding "stub" directory with the FID-named directory: "`.lustre/trash/MDTXXXX/`*`UID/pFID`*". Essentially "`.Trash`" under each normal directory is just a virtual shortcut to the stub directory (if the parent is not a striped dir) that is accessible in each directory if specified by name "`.Trash`". The files/directories under "`.Trash`" directory can be access via normal POSIX file system API such as via `readdir()/stat()/getxattr()` so that it can be used by normal tools such as "`ls -l . Trash/`" or "`find .Trash`" to locate files for restoration or permanent removal. If there are no deleted files under a specific directory, then the virtual `.Trash` directory will not be accessible, and will return `-ENOENT` for any lookup.

2.2. `.Trash` pFID name lookup

The FID-based names of stub directories stored as `.lustre/trash/MDT`*`xxxx/UID/pFID`* directory are needed for efficient lookup of the parent FID during unlink. However, these directory names are not very user-friendly when browsing the virtual `.Trash` directory in the filesystem namespace. Rather than showing the *`pFID`* name to users during `readdir()` calls (`ls`, `find`, etc.) it would be better to look up the actual parent directory name via the FID`trusted.link` xattr on the parent and return this to clients. The FID number of the directory entry would be the FID of the stub directory itself, **not** the pFID that is used internally for identification. While copying the `trusted.link` xattr over to the stub directory at creation would simplify this lookup, there is some risk that the name in the `trusted.link` xattr would become stale if the parent directory is renamed. On the other hand, this may also be useful to preserve the original name of the directory in case some automated tool is renaming the original directory to a temporary name before deletion?

2.3. `.Trash` striped directory

There are two solution to handle striped directory for Trash Can:

- Solution 1:

Create a stub dir on trash Can upon the first file deletion of the striped directory. And the stub dir is using the master FID of the striped directory as the name and using the LMV layout of the striped directory as the layout configuration template.

When deleting a file or directory under the striped directory, the MDT first locates the stripe index of the stripe shard under which the file is deleting and then locates the corresponding stub shard of the stub directory and move the deleting file into the stub shard object.

Under this design, the user can directly access the files or directoies on Trash Can via ".Trash" or ".lustre/trash/MDTXXXX/[uid]/pFID". But the first deleting under a striped directory may cause a distributed metadata transcation. And Each deleting under the striped directory needs to locate the stub shard object any maybe need to obtain LMV layout of stub master striped dir remotely.

- Solution 2:

Do not create the stub striped dir on Trash Can upon the first file deletion of the striped directory. The MDT just creates a stub shard dir naming with the corresponding shard FID of the parent directory and move the file into this stub shard dir in Trash Can.

When access trash files via ".Trash", the client will construct a virtual file with the FID same with the parent dir, but the version of FID is set with (`FID_VERSION_TRASH` = 1).

And the shard FIDs are constructed with the corresponding FIDs with same FID names in Trash Can.

Under this design, the user can only access the trash files for striped directories via ".Trash", can not access them via ".lustre/MDTXXXX/pFID" as the master stub directory does not exist yet until the deleting the parent directory.

When deleting the parent directory, the MDT will create a stripe stub directory on Trash Can where the shards' FIDs are FIDs of corresponding stub shard objects.

Under this design, the file deleting is simpler. Just need to create a stub shard object with the name same as the FID of the corresponding shard for the parent striped directory. But the access via ".Trash" and deleting the striped directory is complex.

For a striped directory, its "`.Trash`" directory is also a vitual striped directory with each stripe on the same location (MDTs) where the shard FID is the FID of the corresponding stub directory on that MDT. If the stub directory on a certain MDT does not exist (or not create yet), the client `lookup()` or `readdir()` under "`.Trash`" directory should skip the stripe. The master FID of the virtual "`.Trash`" directory could be same with the FID of the parent directory but with `f_ver` setting with 1 (`FID_VERSION_TRASH` = 1) to distinguish them.

To avoid the inconsistent problem, each access on the virtual striped "`.Trash`" should check and revalidate the virtual stripe LMV EA. For example, It should add the new shards into the stripe EA after a new stub directory on a certain MDT was created.

2.4. `.Trash` directory migration

It should handle the case that a directory was restriped and the LMV layout was changed. In this case, the files under the directory will be migrated to another MDT. To simplify the implementation, we do not migrate the files according to the new LMV layout in the Trash Can. This may result in the `lookup()` operation will be issued to a wrong MDT and return `-ENOENT` wrongly (after files in the trash can are restored). However, the `readdir()` operations will still return all the dir entries in the striped trash even if the parent LMV layout was re-striped and changed, since the parent directory FID (pFID) will remain the same as before restriping. Maybe it needs to migrate the files restored from the trash can to the appropriate shard according to their name hash once the LMV layout has been changed.

2.5. Orphans in Trash Can

For an orphan file, it means the file is still opened (but not closed) by a certain user. Upon its last unlink, it can directly move into the trash can and mark with LUSTRE_ORPHAN_FL | LUSTRE_TRASH_FL. And the orphans file can not be permanently deleted from the trash can until its last close().

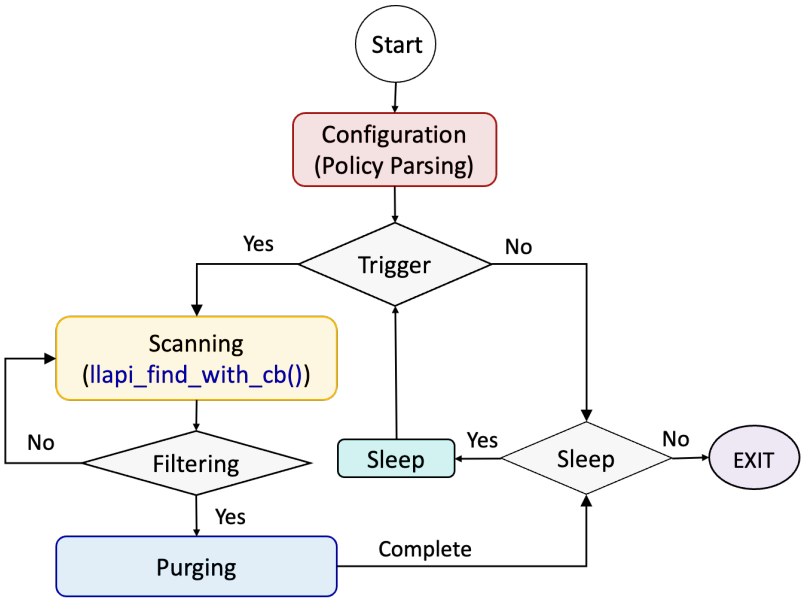
3. Reclaim TCU Space

As introduced above, Lustre Trash Can/Undelete moves deleted files to a special trash directory and provides recovery mechanism, but it still consumes the valuable storage space. To reclaim TCU space, a tool is needed to periodically scan the trash can and purge files based on the configurable policies efficiently.

3.1. ltrash_purge Tool (LU-19598)

ltrash_purge is such a user-space tool designed to automatically manage and clean up files in the Lustre Trash Can from the client side, when the storage becomes full, the files exceed their retention period, or based on user ID, group ID, and proj ID. It periodically scans the trash directories, examines each file, and applies a series of policy checks to determine whether the file should be purged. These policies can be combined, allowing administrators to create sophisticated cleanup strategies that match their organization's specific needs and usage patterns.

The workflow of ltrash_purge is shown in the following figure.



To reuse the existing parallel traversal in Lustre, we define a new pfind API llapi_find_with_cb(). It allows us to decide which to purge with the custom callbacks while scanning. Please see ltrash_purge User Guide (LU-19598) for more details.

Note, ltrash_purge can be used with TCU feature in the following environment to avoid the purging conflicts:

- Multiple-Tenancy environment: the system data is isolated very well by sub-directory mount, in this way, ltrash_purge can be run on each different mount client in parallel, and achieve a good performance.
- TCU reclaim service node: the administrator can control and deploy purging policy on a special node for TCU reclaim service flexibly.

3.2. TCU Reclaim Upcall (LU-19950)

Although ltrash_purge can scan and purge files from the trash can periodically based on the different policies, but it still can't be triggered in time when the system becomes full. To resolve this issue, we define a new TCU reclaim upcall in llite layer to trigger ltrash_purge tool on ENOSPC error in kernel space.

We check ENOSPC error in ll_file_io_generic() for regular write and in ll_page_mkwrite0() for mmap write, then call a handler script defined in the tunable parameter llite.*.tcu_reclaim_upcall to trigger ltrash_purge with the proper configuration.

Note, this design is per-client not filesystem-wide, so there should be a good data access isolation strategy in the system to enable this feature, e.g. multi-tenant environment. Otherwise, when ENOSPC error happens, the applications on the multiple clients will trigger many ltrash_purge instances together, which will cause a lot of conflicts and even overwhelm the system.